

Automated Intrusion Recovery for Web Applications

by

Ramesh Chandra

B.Tech., Computer Science, Indian Institute of Technology, Madras (1998)
M.S., Computer Science, University of Illinois at Urbana-Champaign (2001)
M.S., Computer Science, Stanford University (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

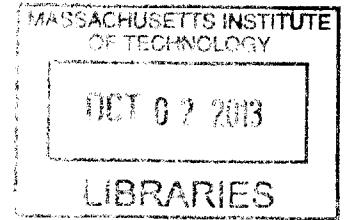
ARCHIVES

Doctor of Philosophy

at the

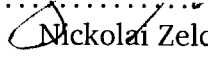
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

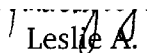
September 2013



© Massachusetts Institute of Technology 2013. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
August 30, 2013

Certified by

Nikolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by

Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Automated Intrusion Recovery for Web Applications

by
Ramesh Chandra

Submitted to the Department of Electrical Engineering and Computer Science
on August 30, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In this dissertation, we develop recovery techniques for web applications and demonstrate that automated recovery from intrusions and user mistakes is practical as well as effective.

Web applications play a critical role in users' lives today, making them an attractive target for attackers. New vulnerabilities are routinely found in web application software, and even if the software is bug-free, administrators may make security mistakes such as misconfiguring permissions; these bugs and mistakes virtually guarantee that every application will eventually be compromised. To clean up after a successful attack, administrators need to find its entry point, track down its effects, and undo the attack's corruptions while preserving legitimate changes. Today this is all done manually, which results in days of wasted effort with no guarantee that all traces of the attack have been found or that no legitimate changes were lost.

To address this problem, we propose that automated intrusion recovery should be an integral part of web application platforms. This work develops several ideas—retroactive patching, automated UI replay, dependency tracking, patch-based auditing, and distributed repair—that together recover from past attacks that exploited a vulnerability, by retroactively fixing the vulnerability and repairing the system state to make it appear as if the vulnerability never existed. Repair tracks down and reverts effects of the attack on other users within the same application and on other applications, while preserving legitimate changes. Using techniques resulting from these ideas, an administrator can easily recover from past attacks that exploited a bug using nothing more than a patch fixing the bug, with no manual effort on her part to find the attack or track its effects. The same techniques can also recover from attacks that exploit past configuration mistakes—the administrator only has to point out the past request that resulted in the mistake.

We built three prototype systems, WARP, POIROT, and AIRE, to explore these ideas. Using these systems, we demonstrate that we can recover from challenging attacks in real distributed web applications with little or no changes to application source code; that recovery time is a fraction of the original execution time for attacks with a few affected requests; and that support for recovery adds modest runtime overhead during the application's normal operation.

Thesis Supervisor: Nickolai Zeldovich

Title: Associate Professor

Acknowledgments

I would like to thank Nikolai Zeldovich for his guidance, for being a great advisor, for encouraging me to think critically, and for being a friend. I have enjoyed working with Nikolai since we were graduate students at Stanford and learnt a lot from him over the years. I thank Hari Balakrishnan, Sam Madden, Frans Kaashoek, and Robert Morris for the helpful research interactions during my graduate study. Special thanks to Hari and Sam for taking the time to be on my committee.

The work in this dissertation resulted from a fruitful collaboration with Taesoo Kim. I have been enriched by working with him and from his friendship. I simply could not have asked for a better collaborator. Thank you, Taesoo.

I thank Eugene Wu for the many helpful discussions on this work, in particular on WARP's time-travel database. I thank Neha Narula and Meelap Shah for helping with WARP's evaluation.

I thank all my friends and colleagues at PDOS, CSAIL, and MIT: Alex, Alvin, Andrea, Austin, Chris, Cody, Emily, Eugene, Frank, Haogang, Hubert, Jacob, Jad, Jayashree, Jonathan, Keith, Kirmani, Lenin, Meelap, Neha, Rahul, Raluca, Shuo, Silas, Taesoo, Xi, and Yandong. They made my stay at MIT an enjoyable experience with their friendly interactions, with the numerous insightful discussions (research or otherwise), and by just being wonderful people.

I thank MokaFive for taking care of my visa during my doctoral studies. Special thanks to Monica Lam and Constantine Sapuntzakis for introducing me to systems research at Stanford. I thank Jad Naous and Dan Steere for getting Compass off the ground while I was wrapping up this dissertation.

No words can suffice to thank my parents for their love and constant support. I would also like to thank Madhu, Latha, Satish, Shirisha, Prahladh, and Priya for always being there.

Contents

1	Introduction	15
1.1	Recovery approach	16
1.2	Recovery challenges	17
1.3	Contributions	17
1.3.1	Identifying attacks	18
1.3.2	Reducing administrator burden	19
1.3.3	Optimizing recovery performance	20
1.3.4	Propagating recovery across web applications	22
1.4	End-to-end recovery example	23
1.5	Related work	24
1.6	Organization	24
2	Recovery for a single web application	25
2.1	Overview	25
2.2	Retroactive patching	27
2.2.1	Normal execution	28
2.2.2	Initiating repair	28
2.2.3	Re-execution	28
2.3	Time-travel database	29
2.3.1	Database rollback	29
2.3.2	Dependency tracking	30
2.3.3	Concurrent repair and normal operation	31
2.3.4	Rewriting SQL queries	31
2.4	DOM-level replay of user input	32
2.4.1	Tracking page dependencies	32
2.4.2	Recording events	33
2.4.3	Server-side re-execution	33
2.4.4	Conflicts	34
2.4.5	Application-specific UI replay	34
2.4.6	User-initiated repair	35
2.5	Implementation	35
2.6	Putting it all together	36
2.7	Evaluation	38
2.7.1	Application changes	38
2.7.2	Recovery from attacks	38
2.7.3	UI repair effectiveness	39
2.7.4	Recovery comparison with prior work	40

2.7.5	Performance evaluation	40
3	Efficient patch-based auditing	45
3.1	Motivating examples	46
3.2	Overview	47
3.2.1	Logging during normal execution	47
3.2.2	Indexing	48
3.2.3	Auditing	48
3.3	Control flow filtering	49
3.3.1	Recording control flow	49
3.3.2	Determining the executed basic blocks	49
3.3.3	Determining the patched basic blocks	50
3.3.4	Indexing	50
3.4	Function-level auditing	50
3.4.1	Comparing results and side-effects	51
3.4.2	Early termination	52
3.5	Memoized re-execution	52
3.5.1	Template generation	53
3.5.2	Dependency tracking	55
3.5.3	Template re-execution	55
3.5.4	Collapsing control flow groups	56
3.6	Implementation	56
3.7	Evaluation	57
3.7.1	Experimental setup	57
3.7.2	Normal execution overheads	58
3.7.3	Detecting attacks	58
3.7.4	Auditing performance	59
3.7.5	Technique effectiveness	60
4	Recovery for distributed web services	63
4.1	Overview	64
4.1.1	Motivating scenarios	65
4.1.2	System model	66
4.1.3	AIRE architecture	66
4.2	Distributed repair	67
4.2.1	Repair protocol	68
4.2.2	Asynchronous repair	69
4.3	Repair access control	70
4.3.1	Delegation of access control	70
4.3.2	Default access control policy	71
4.4	Understanding partially repaired state	72
4.4.1	Modeling repair as API invocations	72
4.4.2	Making service APIs repairable	73
4.5	Implementation	74
4.6	Application case studies	75
4.6.1	Intrusion recovery	75
4.6.2	Partial repair propagation	78
4.6.3	Partial repair in real web services	79

4.6.4	Porting applications to use AIRE	80
4.7	Performance evaluation	80
4.7.1	Overhead during normal operation	80
4.7.2	Repair performance	81
5	Discussion	83
5.1	Assumptions	83
5.2	Limitations	84
6	Related work	87
7	Conclusion	91

List of Figures

1-1	Conceptual model of a general computer system. Users issue commands to the system via the user interface; this could result in execution of the system's code, which may read and write data to persistent storage and communicate with remote computer systems over the network.	15
1-2	Example web applications. Each application has an HTTP server that communicates with browsers; application code runs on a language runtime, stores persistent data in an SQL database, and communicates with other web applications over HTTP; users interact with the applications using browsers. The spreadsheet application stores access control lists for the other applications and periodically pushes the access control lists to the applications.	18
2-1	Overview of WARP's design. Components introduced or modified by WARP are shaded. 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.	26
3-1	Overview of POIROT's design. Components introduced by POIROT are shaded. Solid lines, dotted lines, and dashed lines indicate interactions during normal execution, indexing, and auditing stages, respectively. The Warp / administrator box indicates that POIROT can be used either by WARP to determine the initial set of requests to be repaired, or by an administrator to detect intrusions without invoking WARP for recovery.	47
3-2	Three refinements of request re-execution: (a) naïve, (b) function-level auditing, and (c) early termination. Thick lines indicate execution of unmodified application code, dotted lines indicate execution of the original code for patched functions, and dashed lines indicate execution of new code for patched functions. A question mark indicates comparison of executions for auditing.	51
3-3	Patch for an example application, fixing a cross-site scripting vulnerability that can be exploited by invoking this PHP script as <code>/script.php?q=test&name=<script>..</script></code> . The <code>ucfirst()</code> function makes the first character of its argument uppercase.	53
3-4	URLs of three requests that fall into the same control flow group, based on the code from Figure 3-3.	53
3-5	PHP bytecode instructions for lines 5–12 in Figure 3-3. The line column refers to source lines from Figure 3-3 and the op column refers to bytecode op numbers, used in control transfer instructions. A * indicates instructions that are part of a template for the three requests shown in Figure 3-4 when auditing the patch in Figure 3-3.	54
4-1	Overview of AIRE's design. Components introduced or modified by AIRE are shaded. Circles indicate places where AIRE intercepts requests from the original web service. Not shown are the detailed components for services B and C.	67

4-2	Example scenario demonstrating modeling repair actions as concurrent operations by a repair client. Solid arrows indicate requests during original execution, and dashed arrows indicate eventual repair propagation. The S3 service initiates local repair in between times t_2 and t_3 by deleting the attacker's put. If S3's local repair completes before t_3 , op3 observes value v_0 for X . If A has not yet received the propagated repair from S3, receiving the value v_0 for X at time t_3 is equivalent to a concurrent writer (the hypothetical repair client) doing a concurrent <code>put(x, v0)</code>	73
4-3	Repair in a versioned service. The shaded operation <code>put2</code> from the original history, shown on the left, is deleted during repair, leading to the repaired history of operations shown on the right. The version history exposed by the API is shown in the middle, with two branches: the original chain of versions, shown with solid lines, and the repaired chain of versions, dotted. The mutable "current" pointer moves from one branch to another as part of repair.	74
4-4	Attack scenario in Askbot demonstrating AIRE's repair capabilities. Solid arrows show the requests and responses during normal execution; dotted arrows show the AIRE repair operations invoked during recovery. Request ① is the configuration request that created a vulnerability in the OAuth service, and the attacker's exploit of the vulnerability results in requests ②-⑥. For clarity, requests in the OAuth handshake, other than request ②, have been omitted.	75
4-5	Setup for the spreadsheet application attack scenarios.	77

List of Tables

2.1	Lines of code for different components of the WARP prototype, excluding blank lines and comments.	35
2.2	Security vulnerabilities and corresponding fixes for MediaWiki. Where available, we indicate the revision number of each fix in MediaWiki’s subversion repository, in parentheses. . .	36
2.3	WARP repairs the attack scenarios listed in Table 2.2. The initial repair column indicates the method used to initiate repair.	39
2.4	Effectiveness of WARP’s UI repair. Each entry indicates whether a user-visible conflict was observed during repair. This experiment involved eight victim users and one attacker. . .	40
2.5	Comparison of WARP with Akkuş and Goel’s system [11]. False positives are reported for the <i>best</i> dependency policy in [11] that has no false negatives for these bugs, although there is no single best policy for that system. Akkuş and Goel can also incur false negatives, unlike WARP. The numbers shown before and after the slash are without and with table-level white-listing, respectively.	41
2.6	Overheads for users browsing and editing Wiki pages in MediaWiki. The page visits per second are for MediaWiki without WARP, with WARP installed, and with WARP while repair is concurrently underway. A single page visit in MediaWiki can involve multiple HTTP requests and SQL queries. Data stored per page visit includes all dependency information (compressed) and database checkpoints.	41
2.7	Performance of WARP in repairing attack scenarios described in Table 2.2 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.	42
2.8	Performance of WARP in attack scenarios for workloads of 5,000 users. See Table 2.7 for a description of the columns.	42
3.1	Lines of code for components of the POIROT prototype.	56
3.2	POIROT’s logging and indexing overhead during normal execution for different workloads. The CFG column shows the number of control flow groups. Storage overheads measure the size of compressed logs and indexes. For comparison with the last column, the average request execution time during normal execution is 120 msec.	57

3.3	Detection of exploits and false positives incurred by POIROT for the five MediaWiki vulnerabilities handled by WARP.	59
3.4	POIROT’s auditing performance with 34 patches for MediaWiki vulnerabilities, compared with the performance of the naïve re-execution scheme and with WARP’s estimated repair performance for the same patches using its file-based auditing scheme (WARP’s re-execution of a request during repair is estimated to take 10× the original execution time, based on our evaluation of WARP’s repair performance in §2.7.5). WARP takes less than a second to access its index for file-based auditing. Naïve results are measured only for the top 5 patches; its performance would be similar for the 29 other patches.	59
3.5	POIROT detects information leak vulnerabilities in HotCRP found between April 2011 and April 2012. We exploited each vulnerability and audited it with patches from HotCRP’s git repository (commit hashes for each patch are shown in the “patch” column).	60
3.6	Performance of the POIROT replayer in re-executing all the 100k requests of the Wikipedia 100k workload, for the five patches shown here. The workload has a total of 834 or 844 control flow groups, depending on the MediaWiki version to which the patch was ported. POIROT incurs no false positives for four out of the five patches; it has 100% false positives for the patch 2011-0003, which fixes a clickjacking vulnerability. The “naïve re-exec” column shows the time to audit all requests with full re-execution and the “func-level re-exec” column shows the time to audit all requests with function-level re-execution and early termination. The “early term. ops” column shows the average number of PHP instructions executed up to the last patched function call with early termination (§3.4.2) across all the control flow groups. The “collapsed CF groups” and “collapse time” columns show the number of collapsed control flow groups and the time to perform collapsing of the control flow groups (§3.5.4), respectively. The “template gen. time”, “template ops”, and “memoized re-exec” columns show the time taken to generate templates for all the control flow groups in the workload, the average number of PHP instructions in the generated templates, and the time to re-execute the templates for all the requests, respectively.	61
4.1	AIRE’s repair interface.	68
4.2	AIRE’s access control interface with web services.	71
4.3	Kinds of interfaces provided by popular web service APIs to their clients.	79
4.4	AIRE overheads for creating questions and reading a list of questions in Askbot. The first numbers are requests per second without and with AIRE. The second numbers show the per-request storage required for AIRE’s logs (compressed) and the database checkpoints.	81
4.5	AIRE repair performance. The first two rows show the number of repaired requests and model operations out of the total number of requests and model operations, respectively.	81

Chapter 1

Introduction

Web applications today play an important role in the computing landscape, and consumers and businesses alike rely on them for day-to-day computing; this makes them an attractive target for attackers. Despite a significant amount of past research on prevention of vulnerabilities, new vulnerabilities in web applications are being routinely found. For example, over the past 4 years, an average of 3–4 previously unknown cross-site scripting and SQL injection vulnerabilities were reported *every single day* to the CVE bug database [61]. 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 [33]. As a result, even well-maintained applications can and do get compromised [3, 6, 21, 30, 31, 65, 71, 76].

Though most vulnerabilities discovered until now were fixed before attackers could do serious harm, past attacks like Stuxnet [4] and the compromise of Google’s systems in China [22] show that it is only a matter of time before attackers launch successful attacks that cause serious damage. Every web application will eventually have a successful compromise, and when that happens, administrators of web applications need to recover from the compromise. Due to lack of automated recovery tools, administrators today resort to manual recovery, which, as we shall see, is both tedious and error-prone.

This dissertation’s thesis is that web applications need automated intrusion recovery and that recovery is an important security mechanism that is complementary to, and as important as preventive security measures. In this dissertation, we develop recovery mechanisms which demonstrate that recovery from intrusions and user mistakes is both practical and effective.

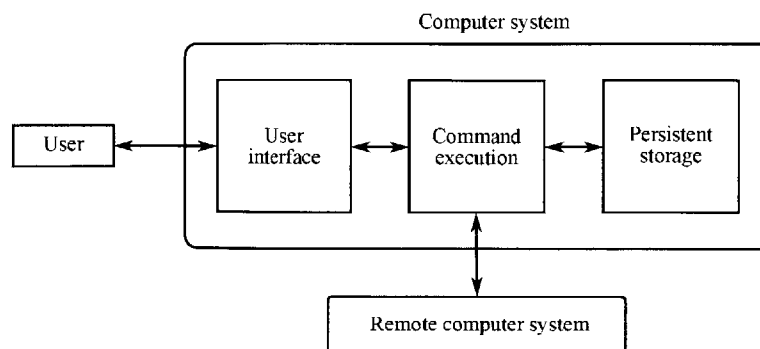


Figure 1-1: Conceptual model of a general computer system. Users issue commands to the system via the user interface; this could result in execution of the system’s code, which may read and write data to persistent storage and communicate with remote computer systems over the network.

1.1 Recovery approach

It is instructive to understand what recovery means in a general setting, before specifically focusing on recovery in web applications. A general computer system can be modeled as shown in Figure 1-1. The system accepts *commands* from users via its user interface (for instance, as keyboard and mouse input), executes the commands, and displays results back to the users. During command execution, the system may read data from, and write data to, persistent storage such as file systems and databases. Furthermore, the system may communicate with remote systems over the network using protocols such as HTTP, SMTP, and SSH. The model in Figure 1-1 is quite general, and systems ranging from desktop operating systems to web-based and mobile applications are conceptually structured in this manner; for example, in a typical web application, the user interface is a browser, network communication is over HTTP, user commands translate to HTTP requests, and a database is used for persistent storage.

Some commands issued to a computer system could be *illegal*, and result in executions that corrupt the system's persistent state and violate one or more of the system's invariants. For example, an attacker may have issued a command that exploited a bug or misconfiguration in the system's software to modify data that she would not otherwise have been able to, or a legitimate user may have issued a command with incorrect data. A correctly functioning system would reject an illegal command outright; however, bugs in the system's code that checks for illegal commands could allow some illegal commands to run. After an illegal command corrupts data in the persistent store, later executions of other commands may use the corrupted data, potentially making them illegal as well.

The goal of recovery then is to restore the system's integrity by undoing corruptions due to past illegal commands, while preserving state changes by legitimate commands. A well-known recovery approach, which we call *rollback-and-redo*, achieves this goal in the following three steps. First, an illegal command that initially corrupted the system state is identified (e.g., a command that was an attack entry point). Second, the persistent state of the system is rolled back to a time before the illegal command, thereby reverting all corruptions to the system state. As this rollback step also reverts legitimate changes, the final step replays all commands after the rollback time except for the illegal command; this reapplies legitimate changes to the persistent state. Rollback-and-redo is a general recovery approach, and many existing recovery systems (such as [15, 45]) are based on this approach; the work in this thesis is based on this approach as well.

The above high-level description of rollback-and-redo omits some important design details of the recovery steps. As we describe later (§1.5), different systems based on rollback-and-redo have different designs for the recovery steps, leading to their differing capabilities. To illustrate the challenges involved in recovery, we use a strawman recovery system that is based on rollback-and-redo, with a design that relies on the administrator to perform the first and third recovery steps—the administrator identifies the initial invalid command (perhaps by inspecting the system logs), and after the recovery system performs rollback, the administrator replays all user commands since the rollback time except for the illegal command (perhaps using a recorded log of the user commands).

When performing recovery using rollback-and-redo, the result of a replayed command may be different from its result during original execution if the command relied on corrupt data that was reverted by recovery. In particular, the command may display a different output to the user or it may change network communication with other systems. This in turn may require changes to subsequent user commands or to the system's subsequent network communications. In the strawman, the administrator has to take this into account when replaying commands during recovery—if the UI output changes, the administrator may have to adapt replay of subsequent commands accordingly; if a network communication changes (e.g., content of an outgoing email changes), the administrator may have to apply a *compensating action* (e.g., notify the email recipient of the change in email contents).

For correctness, recovery assumes that the illegal commands being repaired could not tamper with any logs used for recovery. As a result, system components used for logging are part of the trusted computing base and are assumed to be free of vulnerabilities.

If the system is non-deterministic, there may be many possible repaired states. After recovery completes, the system is guaranteed to be in one of those states, 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, but the repaired state is guaranteed to be free of effects of attack actions.

1.2 Recovery challenges

The strawman illustrates the four challenges that should be addressed to be make recovery practical. First, recovering from an attack requires finding the attacker’s illegal command. This is difficult and tedious for an administrator to do manually: either she has to pore through system logs looking for possible attacks or she has to audit persistent storage for suspicious data and use a tool like Backtracker [46] to identify attacks from that data. Furthermore, despite expending significant manual effort to find attacks, there is no guarantee that the administrator found all attacks; missing an attack’s illegal command results in incomplete recovery and could leave the system corrupt and vulnerable to future attacks.

Second, replaying all users’ commands during recovery demands significant effort from the administrator. If an attack was found a month after it occurred, recovery requires the administrator to replay the commands issued by all users during that month; this can require the administrator to interact with the application’s UI for a month or more, which is not practical.

Third, recovering from a month-old attack by replaying the month’s worth of user commands requires system resources comparable to those needed to originally execute them. However, typically only a few of those commands rely on the attack’s corrupt data and are *affected* by it. The executions of the remaining unaffected commands during recovery have the same result as their original executions. Replaying them during recovery is therefore a waste of time and resources, and the challenge is to avoid replaying them while replaying only the affected commands.

Finally, a compromised system may spread the attack to a remote system by communicating with it; this could corrupt the remote system’s persistent state, which could affect later commands on the remote system. When this happens, recovery should repair the remote system as well; one option is for the administrator to perform manual recovery on the remote system as part of a compensating action. However, performing manual recovery on each affected remote system in turn raises the challenges described in this section, and is therefore often not practical.

The first three challenges described above are in decreasing order of importance, as solving the first challenge is necessary to ensure correctness of recovery by not missing any attacks, solving the second challenge is necessary to minimize user burden, while a solution to the third challenge is purely a performance optimization.

1.3 Contributions

This thesis develops several ideas to solve the above challenges in recovery, as existing work on recovery does not adequately address them (§1.5). These ideas include retroactive patching, automated UI replay, dependency tracking, patch-based auditing, and distributed repair; the rest of this section is devoted to introducing these ideas. To make these ideas concrete, we also designed, implemented and evaluated them in the context of web applications, which are one of the most popular type of applications today. This thesis focuses on web applications that have browser-based UIs, use HTTP for network communication, and store persistent data in a relational database.

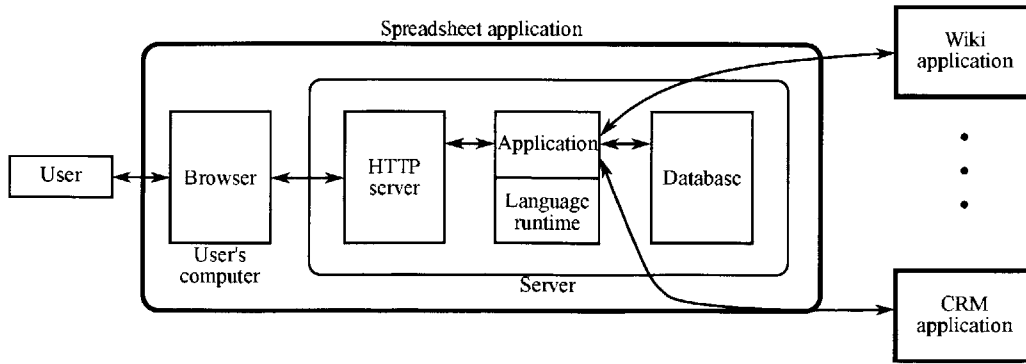


Figure 1-2: Example web applications. Each application has an HTTP server that communicates with browsers; application code runs on a language runtime, stores persistent data in an SQL database, and communicates with other web applications over HTTP; users interact with the applications using browsers. The spreadsheet application stores access control lists for the other applications and periodically pushes the access control lists to the applications.

Though this thesis focuses on making recovery practical for web applications, the ideas developed in this thesis are not limited to web applications. These ideas rely only on the fact that a web application's UI, persistent storage, and network communication have a well-defined structure (for instance, that the HTML UI of a web application has a DOM structure), as will become evident in the subsequent discussion. Therefore, they can also be adapted to other systems that have a well-defined structure to their UI, storage, and network protocols, such as mobile applications, web applications with non-relational databases, and even some traditional desktop applications.

Running example. We use the following running example to introduce the main ideas in this thesis. Businesses today routinely use web applications like Salesforce, Netsuite, and Quickbooks, which integrate with each other to provide workflows suitable for each business. Consider a company that employs several such applications, as shown in Figure 1-2. Each application typically consists of an HTTP server, with application code running on a language runtime, persistent data stored in a relational database, and users interacting with the application using web browsers. User commands on the application's UI can cause browsers to send HTTP requests to the server; applications may communicate with each other using HTTP requests as well.

Imagine that to simplify permission management across the applications, the company's administrators centralize the applications' access control lists in a spreadsheet stored on a service similar to Google Spreadsheets; the administrators use an update script to periodically update the access control lists on the applications, using a capability similar to Google Apps Script. Suppose the spreadsheet application has a cross-site scripting (XSS) vulnerability and an attacker exploits it by tricking a privileged user to visit an attack web page; the attack runs in the background on the attack page and modifies the update script to add the attacker to every application's access control list; later, the attacker logs into the other applications and uses the newly granted privileges to modify data in those applications. Recovering from this attack requires first identifying that the attack took place, and then undoing the attack's modifications to the access control lists as well as the attacker's changes to the other applications' data; the rest of this section shows how the ideas in this thesis recover from this attack.

1.3.1 Identifying attacks

Suppose the developers of the spreadsheet application discover the XSS vulnerability at a later time and issue a patch to fix it. The first challenge faced by recovery is to identify any attacks that exploited

the vulnerability. Previous work on intrusion recovery [15, 24, 29, 37, 45] relied on the spreadsheet application’s administrator to identify intrusions. Administrators typically do this today by manually inspecting the application’s logs or perhaps by using an intrusion analysis tool like Backtracker [46]; however, both these options are tedious for administrators and can miss intrusions. Another option is to require the application’s developer to write vulnerability-specific predicates [40] that test whether each request exploited the vulnerability; however, this adds extra burden for the developer, as the developer has to repeat this process for each discovered vulnerability.

Though the illegal execution in the above example was due to an attack that exploited the XSS vulnerability, in general, illegal executions can arise due to three root causes: (i) a bug in the software that was triggered by an illegal request, possibly with a malicious intent, as was the case in the XSS attack example, (ii) a configuration mistake that was exploited by an illegal request, or (iii) a user mistake that resulted in invalid data being passed to a request. So, the general version of recovery’s first challenge is to identify *all* illegal requests that resulted from a root cause, once the root cause is discovered. This is relatively straightforward for user mistakes, as the user knows which requests she issued incorrectly. However, finding all illegal requests that exploited a bug or misconfiguration is difficult, tedious and error-prone, and missing any illegal requests leads to incorrect recovery.

To solve this challenge, our key insight is that recovery can just fix the root cause instead of identifying and fixing each illegal request that exploited the root cause. The administrator fixes the root cause in the past and replays all requests, *including* illegal requests; because the root cause is fixed, the illegal requests are no longer successful, while the valid requests are successful, thereby recovering from the effects of the illegal executions. For configuration mistakes, fixing the root cause involves correcting the mistake. For software bugs, this involves *retroactive patching*, a novel technique developed in this thesis that fixes a bug in the past by applying a patch at the time the bug was introduced. With these techniques, we fix the system’s execution history to make it appear as if the configuration mistakes or bugs were never present, and relieve the administrator of the tedious and error-prone task of identifying illegal requests.

1.3.2 Reducing administrator burden

The second challenge for recovery is replaying users’ interactions with the spreadsheet application after rolling back the application’s state. The strawman relies on the administrator to manually replay the user interactions; however, this is impractical as it places too much burden on the administrator.

The solution to this problem is to automate replay of user interactions and remove the need for administrator involvement during replay. But, a roadblock to automated replay is that the UI during recovery may be different from what it was during original execution. When this happens, one option is to notify the administrator so that she can manually replay the user’s interaction on the changed page; we call this situation a *conflict*. However, as the UI during recovery is oftentimes slightly different from the UI during normal execution, a naïve replay algorithm that flags a conflict whenever the UI changes can lead to too many conflicts and thereby result in increased burden on the administrator.

We solve this challenge with two ideas. First, we allow each web application to define an application-specific *UI replayer* that is used to replay user interactions during that application’s recovery. An application’s UI replayer better understands the application semantics and hence may be able to automatically handle small changes in UI between original execution and repair. The UI replayer gets as arguments the user’s input during original execution, the UI during original execution, and the UI during recovery. The UI replayer automatically replays the user input if it can; if it cannot perform automatic replay (e.g., because the UI changed considerably), it flags a conflict. On a conflict, recovery stops for the command that caused the conflict and the administrator is notified. Recovery does not wait for the administrator to resolve the conflict, and continues repairing other commands affected by the

attack. When the administrator later resolves the conflict, recovery repairs the command that caused the conflict, thereby reapplying the command's legitimate changes. Until the administrator resolves the conflict, repaired state is missing the command's legitimate changes, but it is still correct in that it is free of the attack's effects.

Second, we record the UI's structure and the user input in a format that is semantically meaningful to application developers and users; in particular, for web applications we record the document object model (DOM) structure of the web pages and the DOM elements on which each user input acts. This helps the UI replayer understand the user's intent during original execution and thereby be more effective at replaying in situations where the UI has changed during repair.

To understand why both the above ideas are necessary, consider the XSS attack described earlier. Suppose that the attack added an invisible `div` element to the victim user's page that changed positioning of other page elements slightly (which the user cannot notice), and uses a handler on the element to issue the background attack request. Also, assume that the user performed a legitimate action on the page by clicking on a link. If the attack `div` element was removed, as it is during recovery, the layout of the page changes but the user does not observe any visible difference in the page and hence would still perform the legitimate action by clicking on the same link.

During recovery, a naïve UI replayer that operates at the pixel level (i.e., the UI is recorded as screenshots and the mouse input is recorded along with the screen coordinates) cannot determine from the recorded information (i.e., the screenshots of the original and the repaired pages, and the mouse click information) that the user clicked on a link and that the click has to be replayed at a different location on the repaired page; this results in the naïve replayer raising a conflict. However, a UI replayer that operates at the DOM level understands that the user's click originally acted on the link element, and so automatically replays the user's click on the same link in the repaired page without raising a conflict.

To alleviate the need for each application developer to develop a UI replayer, we built a default UI replayer that assumes independence of different DOM elements in a page and works as follows: if the user input's DOM element is found in the repaired page and the element's properties did not change, the default UI replayer replays the input on that element; otherwise, the default UI replayer flags a conflict. This default UI replayer suffices for many applications. However, the assumption of independence of DOM elements may not be appropriate for some applications, in which case the application developer can override the default UI replayer with an application-specific UI replayer. For example, an XSS attack may change the description text on a page, as a result of which the user may enter invalid information in a text element on the page. Though recovery reverts the attacker's change to the description, the default UI replayer replays the user's invalid input into the text element, which is incorrect. The application developer understands the link between the description and the text element, and can provide an application-specific UI replayer that flags a conflict in this scenario.

1.3.3 Optimizing recovery performance

The strawman performs recovery by replaying all requests after the time of the root cause. Though automating the replay of these requests reduces administrator burden, replaying them all is still a waste of time and resources, as many requests are not *affected* by the root cause. The third challenge then is to replay only the affected requests, and thereby minimize the resources and time needed for recovery. Though a solution to this challenge is purely a performance optimization and does not affect correctness of recovery, it is nevertheless crucial to making recovery practical.

We say that a request is affected by (or *depends on*) the root cause if the result of the request during recovery, executed on the system with the root cause fixed, is different from its result during original execution. A request can be affected by the root cause in two ways: first, it directly exploits the root cause (e.g., by exploiting a bug in the application code), and second, it relies on persistent data written

by an affected request. To identify affected requests, we track dependencies of requests on a patch fixing a vulnerability, and of requests on each other through the database. We give an overview of how we track these dependencies later in this section.

Once recovery identifies the affected requests, it re-executes each of them, after fixing the root cause. To re-execute a past request, we need to solve two problems. First, we need to make it appear as if the request is re-executing in the past, at the time when it originally executed. We solve this by rolling back database state that the request accesses, to its original execution time (except for the state already fixed by recovery). Our database rollback has to be precise, as rolling back additional database state requires re-executing other requests just to restore the additionally rolled back state. We develop a *time-travel database* that allows us to roll back precisely the database rows accessed by the request to the exact time of the request execution, thereby avoiding unnecessary re-execution for the sole purpose of reconstructing database state.

Second, we need to provide the original inputs to the re-executed request (except for inputs fixed by recovery). We solve this by recording all inputs to the request during original execution and reusing them during recovery. This recorded input also includes results of any non-deterministic function calls made by the request (e.g., calls to system time or the random number generator); the recorded non-deterministic values are fed back during recovery to ensure that a request's results during recovery are not needlessly different from that of its original execution. Returning a different value for a non-deterministic function call (either because it was not recorded during original execution or because the request made different non-deterministic calls during recovery) does not affect recovery correctness; it may only lead to a different final repaired state, increase the amount of re-execution, or increase the number of conflicts that users need to resolve.

We now give an overview of how we use dependency tracking to identify affected requests.

Dependency tracking requirements. We have two requirements for dependency tracking. First, as our goal is to reduce resource usage during recovery, we want dependency tracking itself to be fast and not be resource-intensive. Second, for correctness, dependency tracking should not have any false negatives (i.e., it should not miss any affected requests). However, it may have false positives (i.e., incorrectly identify some unaffected requests as affected), as false positives do not impact correctness of recovery and only lead to more re-execution. Nevertheless, to minimize re-execution and resource usage, we want dependency tracking to be *precise* and have a minimal number of false positives.

To make dependency tracking both fast and precise, our general dependency tracking strategy is as follows. First, we *statically* filter out requests that are not dependent, by using information logged during normal execution. This is fast because it does not require any re-execution, and in practice static filtering often eliminates a large fraction of the non-dependent requests even though it is imprecise. Second, to make dependencies more precise, we perform partial re-execution of the requests remaining after static filtering. We developed techniques to make partial re-execution efficient, which we review below. Only the requests that are not eliminated by these two steps are re-executed to perform recovery.

Dependencies on patch. A request depends on a patch if its result when executed with the patch applied is different from its result during original execution; this follows from our earlier definition of a request's dependency on a root cause. A request that depends on a patch fixing a bug potentially exploited the bug; therefore, we call tracking dependencies on a patch *patch-based auditing*, as it amounts to using the patch to detect intrusions that exploited the bug.

Naïvely re-executing every request to audit a patch requires as much resources as the original execution, which defeats our goal of optimizing resource usage. So, we developed three techniques to speed up patch-based auditing. The first is *control-flow filtering*, which is a fast static filtering technique

that filters out requests that did not execute any patched code and are therefore not dependent on the patch. Though the remaining requests executed patched code, some of them may not be dependent on the patch, because their results when executed with the patched code may be identical to their results during original execution. To make dependencies more precise by eliminating such requests, the remaining requests are re-executed twice—once with the original code and once with the patch applied—and the results are compared. The second technique, *function-level auditing* speeds up this re-execution by eliminating redundant computation between the two executions while auditing a particular request. The third technique, *memoized re-execution*, eliminates redundant computation across auditing of multiple requests. We show that these techniques together speed up patch-based auditing by several orders of magnitude.

Dependencies through database. Suppose that recovery identifies a request A as affected and re-executes it, as part of which it wrote some data to the database. Another request B depends on A through the database if a read query in B, when re-executed, returns different data from the database than it did during original execution. We consider B to be affected and re-execute it as well.

We track dependencies through the database by first statically filtering out unaffected read queries. To do this, we group read queries issued during original execution, by the database column values they used to look up data. During recovery, we use the column values updated by a re-executed write query to efficiently look up the groups of read queries that were potentially affected by that write query; the rest of the reads were unaffected by the write and are filtered out. Once static dependency tracking identifies reads that may be affected, we re-execute each of them and check whether its new result is indeed different from its result during original execution. This makes dependencies precise, and if a read's result was unchanged, avoids the more expensive re-execution of an entire request.

1.3.4 Propagating recovery across web applications

An attack can spread from one web application to another, as in the example scenario, where the XSS attack propagated from the spreadsheet application to the other applications via the update script. The recovery techniques discussed until now can perform local recovery on the spreadsheet application, and during this local recovery, the contents of the update script's requests to the other applications are different from what they were during original execution. One option for the recovery process is to notify the administrator so that she can perform compensating actions to repair the remote applications; this is the only option if remote applications were not running our recovery system. However, if the remote applications are running our recovery system and cooperate, we can automatically propagate repair to those applications as well and perform repair across all the applications. Making this work in practice requires the following four techniques.

First, as there is no strict hierarchy of trust among web applications and no single system can be trusted to orchestrate repair across multiple applications, we automatically extend each application's API to define a repair protocol that allows applications to invoke repair on their past requests and responses to other applications. Second, as some applications affected by an attack may be unavailable during repair and blocking repair until all applications are online is impractical, we perform *asynchronous repair* by decoupling the local repair on a single application from the repair of its interactions with other applications. An application repairs its local state as soon as it is asked to perform a repair, and if any past requests or responses are affected, it queues a repair message for other applications, which can be processed when those applications become available.

Third, our repair protocol should not give attackers new ways to subvert the application. To this end, we enforce access control on every repair invocation. As access control policies can be application-specific, access checks are delegated to applications using an interface designed for this purpose. Finally,

with asynchronous repair, some applications affected by an attack can be already repaired, while others may not yet have received or processed their repair messages. Some applications support such partially repaired state, while others may not; in an application that does not, partially repaired state can appear as corrupted to the application's clients and can lead to unexpected behavior. We develop a model to reason about an application's support for partially repaired state and use the model to show that real, loosely-coupled web applications support partially repaired state with little or no code changes.

1.4 End-to-end recovery example

This section describes how the techniques developed in this thesis come together to help the administrator of the spreadsheet application recover from the example XSS attack; we assume that all applications are running our recovery system. During normal operation, information about each application's execution is continuously recorded to a recovery log on the application's server. This information includes HTTP requests, database queries, and versioned rows in the time-travel database; user input in users' browsers; and communication between servers. During recovery, this recorded information is used both for dependency tracking as well as re-executing affected requests.

When the developers of the spreadsheet application find the XSS vulnerability and fix it, the spreadsheet's administrator initiates recovery using the patch, which proceeds as follows (recovery proceeds similarly for attacks that exploit configuration mistakes or user mistakes).

Recovery begins with patch-based auditing that identifies all requests dependent on the patch; these requests potentially exploited the XSS bug fixed by the patch and are considered attack suspects. Then retroactive patching applies the patch and re-executes each of the suspect requests in the past; this recovers from any corruption caused by them and makes it appear as if these requests executed with the patched code all along. To re-execute a request in the past, recovery uses the time-travel database to roll back the rows that the request either wrote to during original execution or will write to as part of re-execution, to their values at the time the request originally executed; the time-travel database also allows requests to read past values of rows without rolling them back.

Re-execution of a request leads to re-execution of other requests for three reasons. First, the request's response could change, in which case the application's UI replayer is invoked on the browser page rendered by the new response. If there is a conflict, it is queued so that the user can later resolve it; otherwise, the user input is replayed, which can lead to changes in later requests issued by the browser, and those are marked for re-execution as well. In the XSS attack example, during recovery, the response to the attack request does not contain the attack code; after replaying user input on the repaired attack page, the background request modifying the update script is not issued, and so it is canceled (i.e., re-executed with null arguments to make it a no-op).

Second, a request that depends on a re-executed request through the database is also marked for re-execution. In the XSS attack example, assuming that the update script contents are stored in the database, illegal executions of the update script depend on the background request, and they are re-executed as well.

Finally, re-execution of a request can change data communicated to other applications. In the attack example, when the update script is re-executed during recovery, the communicated access control list changes, as the attacker is no longer part of the access control list. In this case, repair is propagated to the remote applications using their repair protocol, and they asynchronously perform local recovery; this reverts the corruptions resulting from the attacker's illegal logins to those applications.

Recovery continues until there are no more requests to re-execute, at which point the state of the system has been updated to undo the effects of attacks that exploited the XSS bug, while preserving effects of legitimate actions.

1.5 Related work

Existing recovery systems do not adequately address the challenges discussed in this chapter. In this section, we review them and discuss how they fall short. We present a detailed review of other related work in Chapter 6.

Industrial backup and restore systems like TimeMachine and Dropbox, and research systems like ReVirt [24] and Polygraph [52], allow the user to roll back selected system state to a past time. But, these systems only provide the rollback step of rollback-and-redo and thereby solve just a small part of the recovery problem; the administrator still has to find the attack, track down modifications made by the attack, and redo legitimate changes after using these systems to revert the attack’s modifications,

Akkuş and Goel’s web application data recovery system [11] uses taint tracking to track every request’s database updates, and future requests that read the updated data; it also uses database logs to generate compensating transactions that revert data modified by a request. This allows an administrator to identify the data that was tainted by an attack request and roll back those changes. However, this system has several limitations: using taint for dependency tracking can lead to the problem of taint explosion, resulting in too many false positives. To deal with this problem, the system allows for false negatives in dependency tracking, as a result of which it may not fully recover from an attack; this makes it suitable only for recovering from accidental corruptions caused due to software bugs. Furthermore, finding the attack and reapplying legitimate changes is still left to the administrator.

Finally, recovery systems like Operator Undo [15] and Retro [45] are closest to, and provide inspiration for, the work in thesis. They perform recovery using the rollback-and-redo approach for certain classes of systems—Operator Undo for Email servers, and Retro for shell-oriented Unix applications on single machine. Once an administrator finds the attack, she can use these systems to roll back the state to before the attack and redo all the later requests, after fixing the root cause manually. However, if directly applied to web application recovery, these systems are similar to the high-level approach discussed in Section 1.1, and they run into the limitations discussed therein. The contributions of this thesis are in solving the challenges resulting from these limitations, as discussed in this chapter.

1.6 Organization

The ideas in this thesis were explored, implemented, and evaluated as part of three systems, each of which solves a part of the recovery problem. The first system is WARP [17], which uses the ideas of retroactive patching, UI replay, time-travel database, and dependency tracking to automate recovery from attacks that target a single web application. The second system, POIROT [43], optimizes WARP by making its patch-based auditing precise and efficient using control-flow filtering, function-level auditing, and memoized re-execution. Finally, the third system, AIRE [18], extends WARP to recover from attacks that spread across multiple applications. In Chapters 2, 3, and 4, we discuss these three systems in more detail along with the ideas explored therein. In Chapter 5, we discuss assumptions underlying our recovery approach and the limitations of our approach. We discuss related work in Chapter 6, and finally conclude in Chapter 7.

Chapter 2

Recovery for a single web application

This chapter describes WARP [17], a system which automates recovery from attacks that target a single web application, including recovery from attacks that run in users' browsers. WARP's goal is to undo all changes made by an attacker to a compromised system, including all effects of the attacker's changes on legitimate actions of other users, and to produce a system state that is as if all the legitimate changes still occurred, but the adversary never compromised the application.

Though limited to a single web application, WARP by itself is a complete recovery system, and its design illustrates the core challenges in using rollback-and-redo to perform automated recovery; the other two systems that comprise this thesis, POIROT and AIRE, build on the foundation laid by WARP. To perform recovery, WARP needs to solve the following three recovery challenges introduced in Chapter 1 (§1.2): identifying intrusions, reducing administrator burden, and optimizing resource usage. WARP uses three ideas that were introduced in Chapter 1 to solve these challenges: retroactive patching (§1.3.1), DOM-level UI replay (§1.3.2), and time-travel database (§1.3.3). This chapter describes these ideas in detail and presents experimental results of evaluating them in a WARP prototype.

2.1 Overview

We begin with an overview of WARP's design, which is illustrated in Figure 2-1 and involves the web browser, the HTTP server, the application code, and the database. Each of these four components corresponds to a *repair manager* in WARP. During the application's normal operation, each repair manager records information that is required to perform rollback and re-execution during repair. As WARP's repair relies on this recorded information, the OS, the HTTP server, the application language runtime, and the database server are in WARP's trusted computing base. WARP also trusts a user's browser when repairing requests issued by that user. WARP can recover from attacks on the web application that do not compromise these trusted components, including common web application attacks such as SQL injection, cross-site scripting, cross-site request forgery, and clickjacking. WARP can also recover from attacks that exploit application configuration mistakes and user mistakes, as long as the trusted components are not compromised.

The information recorded by the repair managers includes information needed for dependency tracking. This dependency tracking information is organized into a dependency graph called an *action history graph*, which captures input and output dependencies of *actions* on *data* (e.g., dependencies of database queries on database rows). The action history graph is used during recovery to identify actions that are affected (and therefore need to be re-run) when some data is repaired. WARP borrowed the concept of action history graph from the Retro intrusion recovery system [45] and adapted it to web application recovery.

WARP’s workflow begins with the administrator deciding that 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 applies the retroactive fix, uses the action history graph to identify requests affected by the fix, and re-executes them after rolling back the appropriate database state. Re-execution of these requests could in turn affect other requests; WARP uses the action history graph to identify those requests and re-executes them as well. WARP continues this recovery process until all affected requests are repaired. This produces a repaired system state that would have been generated if all of the recorded actions originally happened on a system with the fix already applied to it. 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 their exploits.

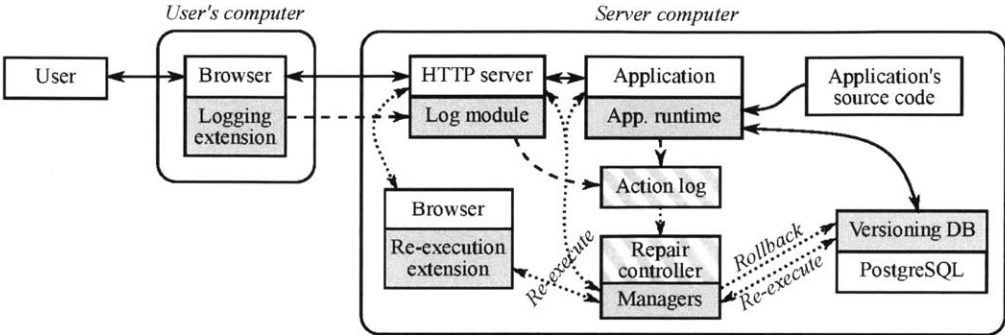


Figure 2-1: Overview of WARP’s design. Components introduced or modified by WARP are shaded. Solid arrows are the original web application interactions that exist without WARP. Dashed lines indicate interactions added by WARP for logging during normal execution, and dotted lines indicate interactions added by WARP during repair.

The rest of this section gives an overview of WARP’s repair using the following worst-case attack example. Imagine that a company has a Wiki site that is used by both employees and customers, and that 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.

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 code¹ may have been affected by a bug in `calendar.php` that was fixed by the patch. 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 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. We call this technique *retroactive patching*, and §2.2 discusses how WARP implements retroactive patching in more detail.

¹A request to a web application results in execution of application code on the server; we call this execution an application run.

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. §2.3 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 runs a UI replayer that 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 recursively undo the effects of those requests as well. §2.4 explains how WARP's browser extension and UI replayer work in more detail.

If a user's actions depend on the attacker's changes, the UI replayer 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, the UI replayer will not be able to re-execute the user's actions, as repair removes the attacker's page. 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 WARP 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. §2.4 further discusses WARP's handling of conflicts and user-initiated undo.

2.2 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.

2.2.1 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. WARP uses input dependencies on source code files to track dependencies on a patch, and an application run is marked dependent on a patch if it loaded a patched file. WARP's patch dependencies can be imprecise and Chapter 3 describes techniques that make them precise.

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.

2.2.2 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.

2.2.3 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).

2.3 Time-travel database

The design of WARP's time-travel database is motivated by three requirements: first, the need to rollback database state to re-execute SQL queries in the past; second, the need to track database dependencies and identify application runs affected by an attack; and finally, the need to repair a web application concurrently with normal operation. WARP aims to minimize the number of SQL queries that are executed to address these requirements; this section discusses how WARP achieves this goal.

2.3.1 Database rollback

The database manager re-executes an SQL query during repair for one of two reasons: to track database dependencies or because it was issued by a re-executed application run. The re-executed query runs at the (past) time at which it originally executed, and so it should see database state as of that time. This is achieved by rolling back database state to that time, before re-executing the query.

When re-executing a write SQL query, one option is to roll back the entire table that the query updates, to the time of the query. However, this requires re-execution of all later write SQL queries on the table to re-apply legitimate changes to the table. This is wasteful, as a write query typically updates only a few rows.

Instead, 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 re-execute a read SQL query, one option is to roll back all the rows that the query reads. However, as read queries often read many rows, this can lead to re-execution of many write queries just to reconstruct legitimate state. To address this issue, WARP performs *continuous versioning* of the database, by keeping track of every value that ever existed for each row. If a re-executed query only reads some rows that were untouched by repair and does not write to them, WARP allows the 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.

Some write SQL queries can update different sets of rows during original execution and repair. To re-execute such 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` = ∞ . 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. As 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.

2.3.2 Dependency tracking

A re-executed application run may have updated the database, which may affect other application runs, requiring them to be re-executed as well. Each affected application run has a read query that when issued on the updated database returns data different from what it returned originally. So, the problem of database dependency tracking is to identify application runs that may be affected, by finding the read queries that return different data on the updated database.

Dependency tracking is complicated by the fact that application runs issue queries over entire tables, and tables often contain data for many independent users or objects of the same type. A naïve approach to track dependencies is to re-execute all read queries after a re-executed application run, and check if they return different data; however, this defeats our goal of minimizing the number of re-executed SQL queries to perform dependency tracking.

Instead, the database manager uses *static dependency tracking* to minimize re-execution of read queries, as follows. It logically splits each table into *partitions*, based on the values of one or more of the table's columns. It then inspects the `WHERE` clause of every read query logged during original execution to statically determine the partitions read by the query. For example, in a Wiki application that stores its Wiki pages in a table with an `editor` column that indicates the user ID of the last editor of a page, a read query with a `WHERE editor='Alice'` clause reads from partition (`editor`, `Alice`). After inspecting the read queries, the database manager creates an index mapping partitions to queries that read those partitions. 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.

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), by identifying all the rows each write query affects (see §2.3.1) and marking partitions corresponding to the column values of these affected rows as modified. The database manager then looks up the partition index to identify the read queries that may be affected. Only these read queries are re-executed to check if their return value has changed.

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.

2.3.3 Concurrent repair and normal operation

Since web applications often serve 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.

WARP implements repair generations by augmenting every table with two additional columns, `start_gen` and `end_gen`; these columns indicate the generations in which a row is valid. Much as with continuous versioning, `end_gen = ∞` indicates that the row has not been superseded in any later generation. During normal execution, queries execute over rows that match `start_gen ≤ current` and `end_gen ≥ current`. During repair, if a row with `start_gen < next` and `end_gen ≥ 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`.

2.3.4 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 ≤ T ≤ end_time AND start_gen ≤ G ≤ 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 ← T`, `end_time ← ∞`, `start_gen ← G`, and `end_gen ← ∞`, 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 $G = next$.

2.4 DOM-level replay of user input

During repair, WARP replays user input in a re-executed browser for two reasons. First, to automatically replay user actions in a changed web page without involving the user, thereby reducing burden on the user. Second, to help users recover from attacks that took place in their browsers; for example, if a repaired 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 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.

When WARP determines that a past HTTP response was incorrect, it re-executes the changed web page in a browser and replays user input, to determine how that page would behave as a result of the change. WARP's browser re-execution uses two ideas. First, as users' browsers may not be online during repair, WARP uses a *cloned browser* on the server for the re-execution. 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.

2.4.1 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, during normal execution, WARP's recording browser extension assigns each browser 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).

2.4.2 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 (e.g., time and event type) and properties of the event's target DOM element (e.g., ID and XPath), which help 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 this could be used to implement an extension-less version of WARP's browser logging by interposing on all events using JavaScript rewriting.

2.4.3 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 browser uses a *re-execution extension* to load the client's HTTP cookies, load the same URL as during original execution, and replay 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.

WARP's re-execution extension uses a *UI replayer* to replay the events originally recorded by the user's browser. For each event, the UI replayer tries to locate the appropriate DOM element using the element's ID or XPath. For keyboard input events into text fields, the UI replayer 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 UI replayer 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.

2.4.4 Conflicts

During repair, WARP's UI replayer 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 attacker may have added an attack link that the user clicked on, or the user may have inadvertently edited a part of the Wiki page that the attacker modified. To deal with such scenarios, WARP's UI replayer replays original user input on a DOM element only if the element is found in the repaired page *and* the element's properties are unchanged (except for the three-way merge on text fields, as described earlier); otherwise 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 that 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.

2.4.5 Application-specific UI replay

WARP's default UI replayer is concerned with replaying input *from* the user on the correct DOM element; hence it replays user input on a DOM element if the element's properties are unchanged, and flags a conflict otherwise. This scheme makes the simplifying assumption that a user action on a DOM element is not influenced by other DOM elements on the page. Though this works well in general, it may not be appropriate for some scenarios, where important information must be correctly displayed *to* the user, because the user's action depends on the displayed information. For example, consider an online money transfer application that displays the transfer amount to the user before she clicks the submit button. Assume an attacker initiates a transfer of \$1,000 from a user but subverts the transfer application to display only \$500 to trick the user into clicking the submit button. During repair, the displayed amount is corrected to \$1,000; however, WARP's default UI replayer clicks on the submit button (because repair did not change the button's properties) and completes the transfer, which is incorrect.

To handle such scenarios, WARP allows the application's programmer to override the default UI replayer with an application-specific UI replayer. The application's UI replayer, given the DOM trees of the original and repaired pages, and the original user actions, can replay the user actions or flag a conflict. In the above example, the application's UI replayer detects that the transfer amount has changed and signals a conflict, so that the user can take corrective action (e.g., cancel the transfer).

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

2.4.6 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 misconfiguration. 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.

2.5 Implementation

We have implemented a prototype of WARP which works with the Firefox browser on the client, and Apache, PostgreSQL, and PHP on the server. Table 2.1 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 UI replayer, we expect that DOM-level events are sufficiently standardized in modern browsers that it would be possible to replay events

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

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.

2.6 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 (§2.4.1). 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 (§2.2.1). Second, for each operation that loads an additional PHP source file, it records the file name (§2.2.1). 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 (§2.3).

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 (§2.4.2). 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 (§2.4.3). 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 (§2.2.3). 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 (§2.3).

After a query re-executes, the repair controller uses dependency tracking 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 UI replayer replays the original user inputs on that page (§2.4.3). If conflicts arise, WARP flags them for manual repair (§2.4.4).

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`.

2.7 Evaluation

In evaluating WARP, we answer several questions. §2.7.1 shows what it takes to port an existing web application to WARP. §2.7.2 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. §2.7.3 shows the effectiveness of WARP’s UI repair in reducing user conflicts. §2.7.4 compares WARP with the state-of-the-art work in data recovery for web applications [11]. Finally, §2.7.5 measures WARP’s runtime cost.

We ported MediaWiki [55], a popular Wiki application that also runs the Wikipedia site, 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.

2.7.1 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.

2.7.2 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 2.2. 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

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 2.3: WARP repairs the attack scenarios listed in Table 2.2. The initial repair column indicates the method used to initiate repair.

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 2.2.

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 2.3 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.

2.7.3 UI repair effectiveness

We evaluated the effectiveness of WARP’s UI repair 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 re-execution browser: First, without WARP’s UI replayer; second, with WARP’s UI replayer but without WARP’s text merging for user input; and third, with WARP’s complete UI replayer. Our experiment had one attacker and eight victims. Each user logged in, visited the attack page to trigger one of the above three attacks, edited Wiki pages, and logged out.

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

Table 2.4: Effectiveness of WARP’s UI repair. Each entry indicates whether a user-visible conflict was observed during repair. This experiment involved eight victim users and one attacker.

Table 2.4 shows the results when WARP is invoked to retroactively patch the XSS vulnerability. Without WARP’s UI replayer, 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 UI replayer 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 UI replayer 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.

2.7.4 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 [11]. 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 2.5 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.

2.7.5 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

Bug causing corruption	Akkuş and Goel [11]		WARP	
	False +ves	User input	False +ves	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 2.5: Comparison of WARP with Akkuş and Goel’s system [11]. False positives are reported for the *best* dependency policy in [11] that has no false negatives for these bugs, although there is no single best policy for that system. Akkuş and Goel can also incur false negatives, unlike WARP. 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 2.6: Overheads for users browsing and editing Wiki pages in MediaWiki. The page visits per second are for MediaWiki without WARP, with WARP installed, and with WARP while repair is concurrently underway. A single page visit in MediaWiki can involve multiple HTTP requests and SQL queries. Data stored per page visit includes all dependency information (compressed) and database checkpoints.

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 2.6 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

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 / 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 2.7: Performance of WARP in repairing attack scenarios described in Table 2.2 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 2.8: Performance of WARP in attack scenarios for workloads of 5,000 users. See Table 2.7 for a description of the columns.

this scenario, we used the XSS, SQL injection, and ACL error workloads from §2.7.2 with 100 users, and victim page visits at the end of the workload. The results are shown in the first four rows of Table 2.7. 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 2.7.

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 2.7. 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 2.8. 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 uses repair generations to allow the web application to continue normal operation. To evaluate repair generations, we measured the performance of MediaWiki for the read and edit workloads from §2.7.5 while repair is underway for the CSRF attack.

The results are shown in the “During repair” column of Table 2.6. 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.

Chapter 3

Efficient patch-based auditing

This chapter presents POIROT [43], a system that performs *precise* and *efficient* patch-based auditing. Given a patch fixing a vulnerability, POIROT identifies *attack suspects*, which are requests that potentially exploited the vulnerability. Patch-based auditing marks a request as suspect if its result when run with the patch applied is different from its result when run with the original code.

Recall from Chapter 2 (§2.2) that WARP’s recovery uses a patch to identify the first set of requests to be re-executed for repair. WARP implemented a simple and efficient auditing scheme that flags as suspect any request that ran a patched source code file during original execution. However, this scheme is imprecise in practice, as patches often update common source code files (e.g., `index.php`) that were run by a large fraction of requests during original execution. This can lead to many false positives in the suspect list and can cause WARP to unnecessarily re-execute many requests during recovery. For example, as we shall see in §3.7, 17 out of 34 recent MediaWiki patches modify common source code files that are executed by every request in a realistic, attack-free workload. For these 17 patches, WARP therefore marks the entire workload as suspect and needlessly re-executes all requests during repair.

POIROT’s precise auditing solves this problem, and WARP can use it to avoid repairing benign requests. POIROT makes patch-based auditing precise by re-executing every request on two versions of the application source code—one with and one without the patch—and comparing the results. If the results are the same (including any side-effects such as modifying files or issuing SQL queries), POIROT concludes that the request did not exploit the vulnerability. Conversely, if the results differ, POIROT reports the request as an attack suspect. Naïvely re-executing every request twice is clearly impractical, as auditing a month’s worth of requests can take several more months. POIROT’s key contribution is in speeding up auditing and making it practical by leveraging the following three key techniques.

First, POIROT performs *control flow filtering* to avoid re-executing requests that did not invoke patched code. To filter out these requests, POIROT records a control flow trace of basic blocks executed by each request during normal execution, and indexes them for efficient lookup. For a given patch, POIROT computes the set of basic blocks modified by the patch, and determines the set of requests that executed those basic blocks. This allows POIROT to skip many requests for patches that modify rarely used code.

Second, POIROT optimizes the two re-executions of each request—one with the patch and one without—by performing *function-level auditing*. Each request is initially re-executed using one process. When a patched function is invoked, POIROT forks the process into two, executes the patched code in one process and the unpatched code in another, and compares the results. If the results do not match, POIROT marks the request as suspect and stops re-executing that request, and if the results are identical, POIROT kills off one of the forked processes and continues re-executing in the other process. Function-level auditing improves performance since forking is often cheaper than re-executing long runs of common application code.

As an extension of function-level auditing, POIROT terminates re-execution of a request if it can determine, based on previously recorded control flow traces, that this request will not invoke any patched functions for the rest of its re-execution. We call this *early termination*.

Third, POIROT eliminates redundant computations—identical instructions processing identical data—that are the same across *different* requests, using a technique we call *memoized re-execution*. POIROT keeps track of intermediate results while re-executing one request, and reuses these results when re-executing subsequent requests, instead of recomputing them. The remaining code re-executed for subsequent requests can be thought of as a dynamic slice for the patched code [8], and is often 1–2 orders of magnitude faster than re-executing the entire request.

Our evaluation of POIROT (§3.7) shows that with the above techniques, POIROT can audit a month’s worth of requests in just hours or days in the worst case. After auditing, the suspect list generated by POIROT can be used by WARP to initiate recovery. Alternately, an administrator can choose to use POIROT for intrusion detection alone, and take remedial measures herself, without using WARP for recovery.

3.1 Motivating examples

The following two examples of recent vulnerabilities illustrate the need for POIROT’s intrusion detection capabilities, even if WARP were not used for recovery. First, consider the HotCRP conference management software [47], which recently had a information disclosure bug that allowed paper authors to view a reviewer’s private comments meant only for program committee members [48]. After applying the patch for this vulnerability, an administrator of a HotCRP site would likely want to check if any comments were leaked as a result of this bug. In order to do so today, the administrator would have to manually examine the patch to understand what kinds of requests can trigger the vulnerability, and then attempt to determine suspect requests by manually poring over logs (such as HotCRP’s application-level log, or Apache’s access log) or by writing a script to search the logs for requests that match a specific pattern. This process is error-prone, as the administrator may miss a subtle way of triggering the vulnerability, and the logs may have insufficient information to determine whether this bug was exploited, making every request potentially suspicious. For example, there is no single pattern that an administrator could search for to find exploits of the HotCRP bug mentioned above.

Manual auditing by the administrator may be an option for HotCRP sites with a small number of users, but it is prohibitively expensive for large-scale web applications. Consider the recent vulnerability in Github—a popular collaborative software development site—where any user was able to overwrite any other user’s SSH public key [65], and thus modify any software repository hosted on Github. After Github administrators learned about and patched the vulnerability, their goal was to determine whether anyone had exploited this vulnerability and possibly altered user data. Although the patch was just a one-line change in source code, it was difficult to determine who may have exploited this vulnerability in the past. As a result, Github administrators disabled all SSH public keys as a precaution, and required all users to re-confirm their keys [28]—an intrusive measure, yet one that was necessary because of the lack of alternatives. With POIROT, patch-based auditing pinpoints the requests that are attack suspects and reports a diff of the results of running these requests with and without the patch; the Github administrators can use this information to identify the few users whose SSH public keys were updated by the attack suspects, and require only those users to re-confirm their keys.

The rest of this chapter is organized as follows. §3.2 presents an overview of POIROT’s design and its workflow. §3.3, §3.4, and §3.5 describe POIROT’s three key techniques for minimizing re-execution. §3.6 discusses our prototype implementation, and §3.7 evaluates it.

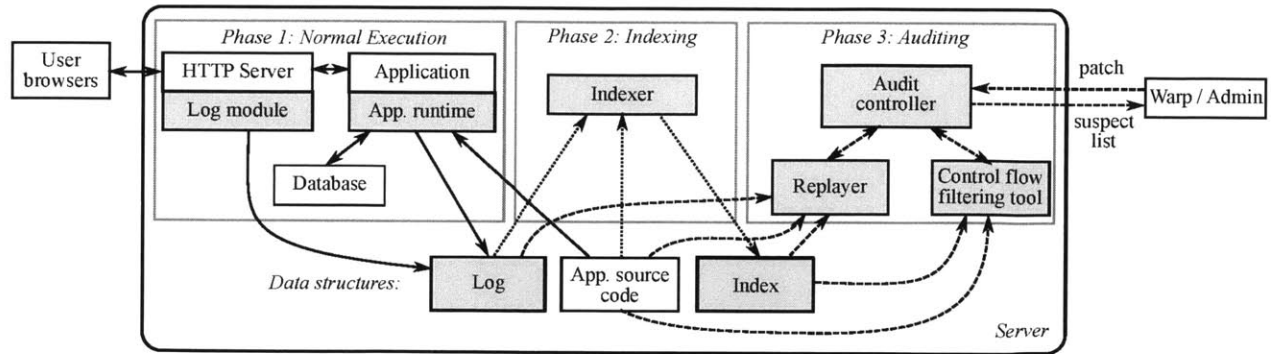


Figure 3-1: Overview of POIROT's design. Components introduced by POIROT are shaded. Solid lines, dotted lines, and dashed lines indicate interactions during normal execution, indexing, and auditing stages, respectively. The Warp / administrator box indicates that POIROT can be used either by WARP to determine the initial set of requests to be repaired, or by an administrator to detect intrusions without invoking WARP for recovery.

3.2 Overview

To understand how POIROT performs auditing, suppose that some request exploited the HotCRP vulnerability mentioned in the previous section, and saw confidential comments. When that request is re-executed by POIROT, the HTTP response with the patch applied will be different from that without the patch (since the response will not contain the comments), and the request will be flagged as suspect. On the other hand, requests that did not exploit the vulnerability will likely generate the same responses, and will not be flagged as suspect. Similarly, in the Github scenario, an attack request that exploited the vulnerability would issue an SQL query to modify the victim's public key. When the attack is re-executed on patched code, the query will not be issued, and POIROT will flag that request as suspect.

More precisely, given a patch fixing a vulnerability in a web application, POIROT's goal is to identify a minimal set of requests that may have exploited the vulnerability. Conceptually, POIROT re-runs each past request to the web application twice—once each with the vulnerable and the patched versions of the application's source code—and compares the results of these runs. If the results are the same, the request is assumed to not exploit the vulnerability; otherwise, POIROT adds the request to a list of requests that may have exploited the vulnerability. These requests can be repaired by WARP or can be further audited by the administrator, based on whether WARP or the administrator invoked POIROT.

A request's result in POIROT logically includes the web server's HTTP response, as well as any side effects of request execution, such as changes to the file system or queries issued to an SQL database. This ensures that POIROT will catch both attacks that altered server state (e.g., modifying rows in a database), as well as attacks that affect the server's HTTP response. POIROT's auditing flags only requests that are possible attack entry points, but does not flag requests that read data written by an attack to a database or requests that were issued by attacks in users' browsers. If WARP was used for recovery, WARP's database dependency tracking and browser re-execution will identify and repair such requests.

POIROT consists of three phases of operation, as illustrated in Figure 3-1: normal execution, indexing, and auditing. The rest of this section describes these three phases.

3.2.1 Logging during normal execution

In order to re-run a past request in a web application, much like WARP, POIROT needs to record the original inputs to the application code that were used to handle the request. Additionally, in order to perform control flow filtering, POIROT must record the original control flow path of each request.

During the normal execution of a web application, POIROT records four pieces of information about each request to a log. First, it records the request's URL, HTTP headers, any POST data, and the CGI parameters (e.g., the client's IP address). Second, it records the results of non-deterministic function calls made during the request's execution, such as calls to functions that return the current date or time, and functions that return a random number. Third, it records the results of calls to functions that return external data, such as calls to database functions. Finally, it records a control flow trace of the application code, at the level of basic blocks [12]. The first three pieces of information are logged by WARP as well, so if POIROT was used in conjunction with WARP, the logs can be shared.

POIROT implements logging by extending the application's language runtime (e.g., PHP in our prototype implementation) and by implementing a logging module in the HTTP server.

3.2.2 Indexing

The second step in POIROT's auditing process is to build an index from the logs recorded during normal execution. The index contains two data structures, as follows.

The first data structure, called the *basic block index*, maps each basic block to the set of requests that executed that basic block, and is generated from the control flow traces recorded during normal execution. This data structure is used by POIROT's *control flow filtering* to efficiently locate candidate requests for re-execution given a set of basic blocks that have been affected by a patch.

The second data structure, called the *function call table*, is a count of the number of times each request invoked each function in the application, and is also generated based on the control flow traces recorded during normal execution. This data structure is used to implement the *early termination* optimization.

POIROT's indexing step can be performed on any machine, and simply requires access to the application source code as well as the control flow traces. Performing the indexing step before auditing (described next) both speeds up the auditing step and avoids having to re-generate these data structures for multiple audit operations.

3.2.3 Auditing

When a patch for a newly discovered security vulnerability is released, POIROT uses the patch as input to start the auditing phase. POIROT's auditing code requires access to the original log of requests, as well as to the index. POIROT first performs *control flow filtering* to filter out requests that did not invoke the patched code, and then uses *function-level auditing* and *memoized re-execution* to efficiently re-execute requests that did invoke the patched code. To ensure requests execute in the same way during auditing as they did during the original execution, POIROT uses the log to replay the original inputs (such as the URL and POST data), as well as the results of any non-deterministic functions and external I/O (e.g., SQL queries) that the application invoked. Note that POIROT does not require a past snapshot of the database for re-executing requests: if the application issues a different SQL query during request re-execution—for which POIROT's log does not contain a recorded result—POIROT flags the request as a potential attack and stops re-executing that request. POIROT performs re-execution by modifying the language runtime (e.g., the PHP interpreter in our prototype), as we will describe later.

Once re-execution finishes, POIROT outputs a list of suspect requests that executed differently with the patched code than they did with the unpatched code.

3.3 Control flow filtering

POIROT’s control flow filtering involves three steps. First, during normal execution, POIROT logs a control flow trace of each request to a log file. Second, during indexing, POIROT computes the set of basic blocks executed by each request. Third, when presented with a patch to audit, POIROT computes the set of basic blocks affected by that patch, and filters out requests that did not execute any of the affected basic blocks, since they could not have possibly exploited the vulnerability in the affected basic blocks. As an optimization, POIROT builds an index that maps basic blocks to the set of requests that executed that basic block, which helps speed up the process of locating all requests affected by a patch.

POIROT performs control flow filtering at the granularity of basic blocks because filtering at a coarser granularity (e.g., at function granularity) can result in fewer requests being filtered out, reducing the effectiveness of filtering. Furthermore, control flow traces at the granularity of basic blocks are also needed for memoized re-execution (§3.5).

The rest of this section describes POIROT’s control flow filtering in more detail.

3.3.1 Recording control flow

In order to implement control flow filtering, POIROT needs to know which application code was executed by each request during original execution. POIROT records the request’s *control flow trace*, which is a log of every bytecode instruction that caused a control flow transfer. For example, our prototype implements control flow filtering at the level of instructions in the PHP interpreter (called “oplines”), and our prototype modifies the PHP runtime to record branch instructions, function calls, and returns from function calls. For each instruction that caused a control flow transfer, POIROT records the instruction’s opcode, the address of that instruction, and the address of the jump target.

Recording control flow traces across multiple requests requires a persistent way of referring to bytecode instructions. PHP translates application source code to bytecode instructions at runtime, and does not provide a standard way of naming the instructions. In order to refer to specific instructions in the application, POIROT names each instruction using a $\langle func, count \rangle$ tuple, where *func* identifies the function containing the instruction, and *count* is the position of the instruction from the start of the translated function (in terms of the number of bytecode instructions). Functions, in turn, are named as $\langle filename, classname, funcname \rangle$.

3.3.2 Determining the executed basic blocks

During the indexing phase, POIROT uses the log recorded above to reconstruct the set of basic blocks executed by each request. To reduce overhead during normal execution, POIROT does not log branches that were not taken. As a result, two adjacent control flow transfers in the log may span n basic blocks, where the branches at the end of the first $n - 1$ basic blocks were not taken.

To compute the set of basic blocks executed by a given request, POIROT first computes the sequence of basic blocks within each function, by translating the application’s source code into bytecode instructions and analyzing the control flow graph in that function. Then, for each pair of adjacent control flow transfers A and B in the request’s log, POIROT adds the sequence of basic blocks between the jump target of A ’s instruction and the address of B ’s instruction to the set of basic blocks executed by that request. To consistently name basic blocks across requests, POIROT refers to basic blocks by the first instruction of that basic block.

3.3.3 Determining the patched basic blocks

During the auditing phase POIROT must determine the set of requests to re-execute to audit the patch provided as input. To filter out requests that were not affected by a given patch, POIROT must determine which basic blocks are affected by a change to the application's source code, and which basic blocks are unchanged. In general, deciding program equivalence is a hard problem. POIROT simplifies the problem in two ways. First, POIROT determines which functions were modified by a patch. Second, POIROT generates control flow graphs for the modified functions,¹ with and without the patch, and compares the basic blocks in the control flow graph starting from the function entry point. If the basic blocks differ, POIROT flags the basic block from the unpatched code as "affected." If the basic blocks are the same, POIROT marks the basic block from the unpatched code as "unchanged," and recursively compares any successor basic blocks, avoiding loops in the control flow graph.

3.3.4 Indexing

To avoid re-computing the set of basic blocks executed by each request across multiple audit operations, and to reduce the user latency for auditing, POIROT caches this information in an index for efficient lookup. POIROT's index contains a mapping from basic blocks (named by the first bytecode instruction in the basic block) to the set of requests that executed that basic block. By using the index, POIROT can perform control flow filtering by computing just the set of basic blocks affected by a patch, and looking up these basic blocks in the index.

The index is generated asynchronously, after the control flow trace for a request has been logged, to avoid increasing request processing latency. The index is shared by all subsequent audit operations. In principle, the index (and the recorded control flow traces for past requests) may need to be updated to reflect new execution paths taken by patched code, after each patch is applied in turn, if we want to audit the cumulative effect of executing all of the applied patches. Our current prototype does not update the control flow traces for past requests after auditing.

3.4 Function-level auditing

After POIROT's auditing phase uses control flow filtering to compute the set of requests affected by the patch, it re-executes each of those requests twice—once with and once without the patch applied—in order to compare their outputs. A naïve approach of this technique is shown in Figure 3-2(a). However, the only code that differs between the two executions comes from the patched functions; the rest of the code invoked by the two executions is the same. For example, suppose an application developer patched a bug in an access control function that is invoked by a particular request. All the code executed by that request before the access control function will be the same both with and without the patch applied. Moreover, if the patched function returns the same result and has the same side-effects as the unpatched function, then all the code executed after the function is also going to be the same both with and without the patch.

To avoid executing the common code twice, POIROT implements function-level auditing, as illustrated in Figure 3-2(b). Function-level auditing starts executing each request in a single process. Whenever the application code invokes a function that was modified in the patch, POIROT forks the process, and invokes the patched function in one process and the unpatched function in the other process. Once the functions return in both processes, POIROT terminates the child fork, and compares the results and side-effects of executing the function in the two forks, as we describe in §3.4.1. If the results and side-effects are

¹PHP has no computed jumps within a function, making it possible to statically construct control flow graphs for a function.

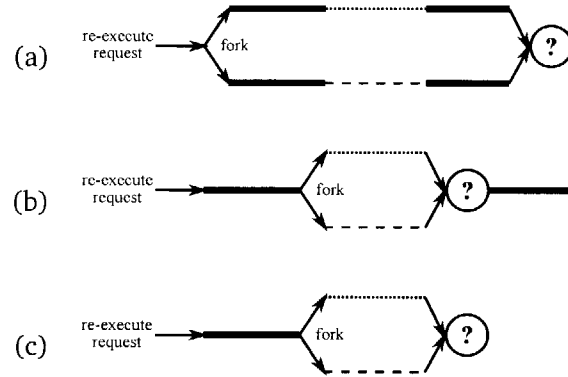


Figure 3-2: Three refinements of request re-execution: (a) naïve, (b) function-level auditing, and (c) early termination. Thick lines indicate execution of unmodified application code, dotted lines indicate execution of the original code for patched functions, and dashed lines indicate execution of new code for patched functions. A question mark indicates comparison of executions for auditing.

identical, POIROT continues executing common application code. Otherwise, POIROT flags the request as suspect, since the request’s execution may have been affected by the patch.

Comparing the results of each patched function invocation, as in POIROT’s function-level auditing, can lead to more false positives than comparing the output of the entire application. This is because the application may produce the same output even if a patched function produces a different return value or has different side-effects with or without the patch. For example, some request may have invoked a patched function, and obtained a different return value from the patched function, but this return value did not affect the eventual HTTP response. These extra false positives can be eliminated by doing full re-execution on the suspect list, and comparing application-level responses, after the faster forked re-execution filters out the benign requests. Our PHP prototype does not implement this additional step, as none of our experiments observed such false positives.

3.4.1 Comparing results and side-effects

A key challenge for function-level auditing is to compare the results and side-effects of invoking an individual function, rather than comparing the final HTTP response of the entire application. To do this, POIROT tracks three kinds of results of a function invocation: HTTP output, calls to external I/O functions (such as invoking an SQL query), and writes to *shared objects*, which are objects not local to the function.

To handle HTTP output, POIROT buffers any output during function execution. When the function returns, POIROT compares the outputs of the two executions.

To handle external I/O functions, POIROT logs the arguments and return values for all external I/O function calls during normal execution. When an external I/O function is invoked during re-execution (in either of the two forks), POIROT checks that the arguments are the same as during the original execution. If so, POIROT supplies the previously recorded return value in response. Otherwise, POIROT declares the request suspect and terminates re-execution.

To handle writes to shared objects, POIROT tracks the set of shared objects that are potentially accessed by the patched function. Initially, the shared object set includes the function’s reference arguments and object arguments. The function’s eventual return value is also added to the shared object set, unless POIROT determines that the caller ignores the function’s return value (by examining the

caller's bytecode instructions). To catch accesses to global variables, POIROT intercepts PHP opcodes for accessing a global variable by name, and adds any such object being accessed to the shared object set.

When the function returns, POIROT serializes all objects in the shared object set, and checks that their serialized representations are the same between the two runs. If not, it flags the request as suspect and terminates re-execution. POIROT recursively serializes objects that point to other objects, and records loops between objects, to ensure that it can compare arbitrary data structures.

3.4.2 Early termination

If a patch just modifies a function that executes early in the application, re-executing the rest of the application code after the patched function has already returned is not necessary. To avoid re-executing such code, POIROT implements an *early termination* optimization, as shown in Figure 3-2(c). Early termination stops re-execution after the last invocation of a patched function returns.

To determine when a request invokes its last patched function, POIROT uses the request's recorded control flow trace to count the number of times each request invoked each function. As an optimization, the indexing phase builds a *function call table* storing these counts.

3.5 Memoized re-execution

Many requests to a web application execute similar code. For example, if two requests access the same Wiki page in Wikipedia, or the same paper in HotCRP, the computations performed by the two requests are likely to be similar. To avoid recomputing the same intermediate results across a group of similar requests, POIROT constructs, at audit time, a *template* that memoizes any intermediate results that are identical across requests in that group. Of course, no two requests are entirely identical: they may differ in some small ways from one another, such as having a different client IP address or a different timestamp in the HTTP headers. To capture the small differences between requests, POIROT's templates have *template variables* which act as template inputs for these differences. POIROT can use a template to quickly re-execute a request by plugging in that request's template variables (i.e., unique parameters) and running the template.

Memoizing identical computations across requests requires addressing two challenges. First, locating identical computations—sequences of identical instructions that process identical data—across requests is a hard problem. Even if two requests invoke the same function with the same arguments, that function may read global variables or shared objects; if these variables or objects differ between the two invocations, the function will perform a different computation, and it would be incorrect to memoize its results. Similarly, a function can have side effects other than its return value. For instance, a function can modify a global variable or modify an object whose reference was passed as an argument. Memoizing the results of a function requires also memoizing side effects.

Second, POIROT's templates must interleave memoized results of identical computations with re-execution of code that depends on template variables. For example, consider the patch for a simple PHP program shown in Figure 3-3, and suppose the web server received three requests, shown in Figure 3-4. The value of `$s` computed on lines 7, 8, and 9 is the same across all three requests, but line 10 generates a different value of `$s` for every request, and thus must be re-executed for each of the three requests. This is complicated by the fact that memoized and non-memoized computations may have control flow dependencies on each other. For instance, what should POIROT do if it also received a request for `/script.php?q=foo`, which does not pass the `if` check on line 5?

POIROT's approach to addressing these two challenges leverages control flow tracing during normal execution. In particular, POIROT builds up templates from groups of requests that had identical control

```

1  function name($nm) {
2  - return $nm;
2  + return htmlspecialchars($nm);
3  }
4
5  if ($_GET['q'] == 'test') {
6    $nm = ucfirst($_GET['name']);
7    $s = "Script ";
8    $s .= $_SERVER['SCRIPT_URL'];
9    $s .= " says hello ";
10   $s .= name($nm);
11   echo $s;
12  }

```

Figure 3-3: Patch for an example application, fixing a cross-site scripting vulnerability that can be exploited by invoking this PHP script as `/script.php?q=test&name=<script>..</script>`. The `ucfirst()` function makes the first character of its argument uppercase.

```

1 /script.php?q=test&name=alice
2 /script.php?q=test&name=bob
3 /script.php?q=test&name=<script>..</script>

```

Figure 3-4: URLs of three requests that fall into the same control flow group, based on the code from Figure 3-3.

flow traces, even if their inputs differed, such as the three requests shown in Figure 3-4. By considering requests with identical control flow, POIROT avoids having to locate identical computations in two arbitrary executions. Instead, POIROT’s task is reduced to finding instructions that processed the same data in all requests with identical control flow traces, in which case their results can be memoized in the template. Moreover, by grouping requests that share control flow, POIROT simplifies the problem of separating memoized computations from computations that depend on template variables, since there can be no control flow dependencies.

More precisely, POIROT’s memoized re-execution first groups requests that have the same control flow trace into a *control flow group*. POIROT then builds up a template for that group of requests, which consists of two parts: first, a sequence of bytecode instructions that produces the same result as the original application, when executing any request from the control flow group, and second, a set of memoized intermediate results that are identical for all requests in the control flow group and are used by the instructions in the template. Due to memoization, the number of instructions in a template is often 1–2 orders of magnitude shorter than the entire application (§3.7).

The rest of this section explains how POIROT generates a template for a group of requests with identical control flow, and how that template is used to efficiently re-execute each request in the group.

3.5.1 Template generation

To generate a template, POIROT needs to locate instructions that processed the same data in all requests, and memoize their results. A naïve approach is to execute every request, and compare the inputs and outputs of every instruction to find ones that are common across all requests. However, this defeats the point of memoized re-execution, since it requires re-executing every request.

To efficiently locate common instruction patterns, POIROT performs a taint-based dependency analysis [62], building on the observation that the computations performed by an application for a given request are typically related to the inputs provided by that request. Specifically, POIROT considers the inputs for all of the requests that share a particular control flow trace: each GET and POST parameter,

Line	Op	Bytecode instruction
5	1	FETCH_R \$0 ← ‘_GET’
5	2	FETCH_DIM_R \$1 ← \$0, ‘q’
5	3	IS_EQUAL ~2 ← \$1, ‘test’
5	4	JMPZ ~2 →20
6	5	FETCH_R \$3 ← ‘_GET’
6	6*	FETCH_DIM_R \$4 ← \$3, ‘name’
6	7*	SEND_VAR \$4
6	8*	DO_FCALL \$5 ← ‘ucfirst’
6	9*	ASSIGN !0 ← \$5
7	10	ASSIGN !1 ← ‘Script ’
8	11	FETCH_R \$8 ← ‘_SERVER’
8	12	FETCH_DIM_R \$9 ← \$8, ‘SCRIPT_URL’
8	13	ASSIGN_CONCAT !1 ← !1, \$9
9	14	ASSIGN_CONCAT !1 ← !1, ‘ says hello ’
10	15*	SEND_VAR !0
10	16*	DO_FCALL \$12 ← ‘name’
10	17	ASSIGN_CONCAT !1 ← !1, \$12
11	18	ECHO !1
12	19	JMP →20
13	20	RETURN 1

Figure 3-5: PHP bytecode instructions for lines 5–12 in Figure 3-3. The line column refers to source lines from Figure 3-3 and the op column refers to bytecode op numbers, used in control transfer instructions. A * indicates instructions that are part of a template for the three requests shown in Figure 3-4 when auditing the patch in Figure 3-3.

CGI parameters (such as requested URL and the client’s IP address), and stored sessions. In PHP, these inputs appear as special variables, called “superglobals”, such as `$_GET` and `$_SERVER`. POIROT then determines which of these inputs are common across all requests in the group (and thus computations depending purely on those inputs can be memoized), and which inputs differ in at least one request (and thus cannot be memoized). Inputs in the latter set are called *template variables*. For instance, for the three requests shown in Figure 3-4, the GET parameter name is a template variable, but the GET parameter q is not.

To generate the template, POIROT chooses an arbitrary request from the group, and executes it while performing dependency analysis at the level of bytecode instructions; we describe the details of POIROT’s dependency tracking mechanism in §3.5.2. POIROT initially marks all template variable values as “tainted”, to help build up the sequence of instructions that depend on the template variables and thus may compute different results for different requests in the group. Any instructions that read tainted inputs are added to the template’s instruction sequence, and their outputs are marked tainted as well. If an instruction is added to the template but some of its input operands are not tainted, the current values of those operands are serialized, and the operand in the instruction is replaced with a reference to the serialized object, such as the \$3 operand of instruction 6 in Figure 3-5. This implements memoization of identical computations. Instructions that have no tainted inputs, as well as any control flow instructions (jumps, calls, and returns), are not added to the template.

For example, consider the PHP bytecode instructions shown in Figure 3-5. Instructions 1–5 do not read any tainted inputs, and do not get added to the template. Instructions 6–9 depend on the tainted `$_GET[‘name’]` template variable, and are added to the template. Instructions 10–14 again do not read any tainted inputs, and do not get added to the template. Finally, instructions 15 and 16 are tainted, and get added to the template, for a total of 6 template instructions.

When POIROT's template generation encounters an invocation of one of the functions being audited, it marks the start and end of the function invocation in the template, to help audit these function invocations later on, as we will describe in §3.5.3. If the recorded control flow trace indicates that there will not be any more invocations of patched functions, template generation stops. Going back to Figure 3-5, template generation stops after instruction 16, because there are no subsequent calls to the patched `name()` function.

3.5.2 Dependency tracking

In order to determine the precise set of instructions that depend on template variables, POIROT performs dependency analysis while generating each template at audit time. In particular, POIROT keeps track of a fine-grained “taint” flag for each distinct memory location in the application. The taint flag indicates whether the current value of that memory location depends on any of the template variables (which are the only memory locations initially marked as tainted). The value of any untainted memory location can be safely memoized, since its value cannot change if the template is used to execute a different request with a different value for one of the template variables. In the PHP runtime, this corresponds to tracking a “taint” flag for every `zval`, including stack locations, temporary variables, individual elements in an array or object, etc.

POIROT computes the taint status of each bytecode instruction executed during template generation. If any of the instruction's operands is flagged as tainted, the instruction is said to be tainted, and is added to the template. The instruction's taint status is used to set the taint flag of all output operands. For example, instruction 6 in Figure 3-5 reads a template variable `$_GET['name']`; as a result, it is added to the template and its output `$4` is marked tainted. On the other hand, instruction 12 reads `$_SERVER['SCRIPT_URL']`, which is not tainted; as a result, its output `$9` is marked as non-tainted.

A template contains only the tainted instructions, which are a subset of the total instructions executed during a request. The output of executing the template instructions for a request is a subset of the output of fully re-executing a request. It is sufficient for POIROT to use the output of template instructions for auditing because the output of non-tainted instructions would be the same in both the patched and unpatched executions.

POIROT's taint tracking code knows the input and output operands for all PHP bytecode instructions. However, PHP also includes several C functions (e.g., string manipulation functions), which appear as a single instruction at the bytecode level (e.g., instruction 8 in Figure 3-5). To avoid having to know the behavior of each of those functions, POIROT assumes that such functions do not access global variables that are not explicitly passed to them as arguments. Given that assumption, POIROT conservatively estimates that each C function depends on all of its input arguments, and writes to its return value, reference arguments, and object arguments. We encountered one function that violates our assumption about not affecting global state: the `header()` function used to set HTTP response headers. POIROT special-cases this function.

3.5.3 Template re-execution

Once a template for a control flow group is generated, POIROT uses the template to execute every request in that control flow group. To invoke the template for a particular request, POIROT assigns the template variables (e.g., `$_GET['name']` in Figure 3-5) with the values from that request, and invokes the template bytecode. In the example of Figure 3-5, this would involve re-executing instructions 6–9 and 15–16. When the template bytecode comes to an invocation of a patched function (e.g., instruction 16 in Figure 3-5), POIROT performs function-level auditing, as described in §3.4, to audit the execution of this function for this particular request. Once the function returns, POIROT compares the results of the

Component	Lines of code
PHP runtime logger / replayer	9,400 lines of C
Indexer	300 lines of Python
Audit controller	1,200 lines of Python
Control flow filter tool	4,800 lines of Python

Table 3.1: Lines of code for components of the POIROT prototype.

function between the two versions (with and without the patch), and assuming no differences appear, POIROT continues executing the template’s bytecode instructions.

In principle, it should be possible to use memoized re-execution to reduce the number of bytecode instructions executed inside the patched function as well. We chose a simpler approach, where the entire patched function is re-executed for auditing, mostly to reduce the complexity of our prototype. Most patched functions are short compared to the number of instructions executed in the entire application, allowing us to gain the bulk of the benefit by focusing on instructions outside of the patched functions.

3.5.4 Collapsing control flow groups

The efficiency of memoized re-execution depends on the number of requests that can be aggregated into a single control flow group. Even though the cost of template generation is higher than the cost of re-executing a single request, that cost is offset by the much shorter re-execution time of all other requests in that control flow group.

Building on the early termination optimization from §3.4.2, we observe that the only part of the control flow trace that matters for grouping is the trace up to the return from the last invocation of a patched function. Instructions executed after that point are not re-executed due to early termination. Thus, two requests whose control flow traces differ only after the last invocation of a patched function can be grouped together for memoized re-execution.

POIROT uses this observation to implement control flow group collapsing. Given a patch, POIROT first locates the last invocation of a patched function in each control flow group, and then coalesces control flow groups that share the same control flow prefix up to the last invocation of a patched function in each trace. This optimization generates larger control flow groups, and thus amortizes the cost of template generation over a larger number of similar requests.

3.6 Implementation

We implemented a prototype of POIROT for PHP. Table 3.1 shows the lines of code for the different components of our prototype. We modified the PHP language runtime to implement POIROT’s logging and re-execution. The rest of the POIROT components are implemented in Python. The indexer and control flow filter tool use the PHP Vulcan Logic Dumper [66] to translate PHP source code into PHP bytecode in an easy-to-process format, and use that to identify executed and patched basic blocks during control flow filtering.

In order to perform efficient re-execution, POIROT assumes that all patched code resides in functions. However, PHP also supports “global code,” which does not reside in any function and is executed when a script is loaded. This causes function-level auditing to execute all of the application code twice, since the “patched function”, namely, the global code, returns only at the end of the script. This can be avoided by refactoring the patched global code into a new function that’s invoked once from the global code. We performed this refactoring manually for one patch when evaluating POIROT.

Workload	# CFG	Latency increase	Thruput reduction	Per-request overheads		
				Log space	Index space	Indexing time
Single URL (1k)	5	13.8%	10.3%	4.95 KB	0.06 KB	12.3 msec
Unique URLs (1k)	238	14.9%	20.4%	21.32 KB	1.79 KB	28.9 msec
Wikipedia (10k)	499	14.1%	16.9%	6.72 KB	4.12 KB	3.5 msec
Wikipedia (100k)	834	14.1%	15.3%	5.12 KB	0.23 KB	0.8 msec

Table 3.2: POIROT’s logging and indexing overhead during normal execution for different workloads. The CFG column shows the number of control flow groups. Storage overheads measure the size of compressed logs and indexes. For comparison with the last column, the average request execution time during normal execution is 120 msec.

POIROT’s control flow filtering does not support PHP’s reflection API. For example, if a patch adds a new function that was looked up during the original execution of a request (and did not get executed because it did not exist), control flow filtering would miss that request, and not re-execute it. Supporting reflection would require logging calls to the reflection API, and re-executing requests that reflected on modified functions or classes. We did not find this necessary for the applications we evaluated.

3.7 Evaluation

Our evaluation aims to support the following hypotheses:

- POIROT incurs low runtime overhead (§3.7.2).
- POIROT detects exploits of real vulnerabilities with few false positives (§3.7.3).
- Even for challenging patches that affect every request, POIROT can audit much faster than naïve re-execution and POIROT’s auditing can significantly improve WARP’s repair performance.
- POIROT’s techniques are important for performance (§3.7.5).

Using a realistic MediaWiki workload and a synthetic HotCRP workload, we show that POIROT’s re-execution for auditing is 24–133× faster than that of naïve re-execution, and an additional factor of ~5× faster than WARP’s re-execution for repair. POIROT catches exploits of real vulnerabilities, with only one patch out of 34 in MediaWiki (and none out of four in HotCRP) causing false positives.

3.7.1 Experimental setup

The test applications used for these experiments were MediaWiki, the popular Wiki application we used to evaluate WARP (§2.7), and HotCRP, a popular web-based conference management system. All experiments ran on a 3.07 GHz Intel Core i7-950 machine with 12 GB of RAM. Since the POIROT prototype is currently single-threaded (although in principle the design has lots of parallelism), we used only one core in all experiments.

To obtain a realistic workload, we derived our MediaWiki workload from a real Wikipedia trace [72]. That trace is a 10% sample of the 25.6 billion requests to Wikipedia’s ~20 million unique Wiki pages during a four-month period in 2007. As we did not have time to run the entire four-month trace, we downsampled it to 100k requests. To maintain the same distribution of requests in our workload as in the Wikipedia trace, we chose 1k Wikipedia Wiki pages and synthesized a workload of 100k requests to them, with the same Zipf distribution as in the Wikipedia trace. This new workload has an average of 100 requests per Wiki page, which is more challenging for POIROT than the Wikipedia workload (1k

requests per Wiki page), since memoized re-execution works better when more requests have identical control flow traces.

As the Wikipedia database is several terabytes in size, we used the database of the smaller Wikimedia Labs site [5] for our experiments, and mapped the URLs of Wikipedia Wiki pages in our workload to the URLs of Wikimedia Labs Wiki pages. Finally, for privacy reasons, the trace we used did not contain user-specific information such as client IP addresses; to simulate requests by multiple users in the workload, we assigned random values for the client IP address and the user-agent HTTP headers.

3.7.2 Normal execution overheads

To illustrate POIROT’s overhead during normal execution, we used several workloads; the results are shown in Table 3.2. The single URL workload has 1k requests to the same URL, the unique URLs workload has one request to each of the 1k unique URLs in the Wikipedia workload, and the Wikipedia 10k and 100k workloads contain 10k and 100k requests respectively, synthesized as above.

The results demonstrate that POIROT’s logging increases average request latency by about 14%, reduces the throughput of normal execution by 10–20%, and POIROT logs require 21 KB per request in the worst case, when all URLs are distinct. POIROT’s storage overhead drops considerably for workloads with more common requests, because the log size primarily depends on the number of unique control flow groups. We expect that log sizes for the full Wikipedia trace [72] would be even smaller, since it has an order of magnitude more common requests than our 100k workload. Recall from §2.7.5 that WARP’s throughput overheads for logging are ~25%. As WARP and POIROT log similar information, when used together for recovery their logs can be shared, resulting in a combined overhead that is closer to WARP’s overhead instead of the sum of the overheads.

Table 3.2 additionally reports the time taken by POIROT’s indexing, even though it can be executed at a later time on a separate machine. The indexer takes 1–29 msec per request, and the index file size is 0.06–4.12 KB per request. As with normal execution, indexing time and storage requirements drop for workloads with more common requests. This is because most of the indexing overhead lies in indexing control flow traces, and common requests often have identical control flow traces.

3.7.3 Detecting attacks

We evaluated how well POIROT detects exploits of patched vulnerabilities by using previously discovered vulnerabilities in our two applications, MediaWiki and HotCRP. Using MediaWiki helps us compare POIROT to WARP, and we used the same five vulnerabilities that we used to evaluate WARP. The real Wikipedia trace [72] did not contain any attack requests for these vulnerabilities, so we constructed exploits for all five vulnerabilities, and added these requests to our 100k workload. Table 3.3 shows the results of auditing this workload with POIROT. POIROT can detect all the attacks detected by WARP, and has no false positives for four out of the five attacks. For the clickjacking vulnerability, the patch adds an extra `X-Frame-Options` HTTP response header. This modifies the output of every request, causing POIROT to flag each request as suspect. Extending POIROT’s re-execution during auditing to include the browser would likely prevent these false positives. Additionally, POIROT incurs no false positives for 29 other patches shown in Table 3.4.

To show that POIROT can detect information disclosure vulnerabilities in HotCRP, we constructed exploits for four recent vulnerabilities, including the comment disclosure vulnerability mentioned in §3.1, and interspersed attack requests among a synthetic 200-user workload consisting of user creation, user login, paper submissions, etc. Table 3.5 shows the results. POIROT is able to detect all four attacks with no false positives.

CVE	Description	Detected?	False +ves
2009-4589	Stored XSS	✓	0
2009-0737	Reflected XSS	✓	0
2010-1150	CSRF	✓	0
2004-2186	SQL injection	✓	0
2011-0003	Clickjacking	✓	100%

Table 3.3: Detection of exploits and false positives incurred by POIROT for the five MediaWiki vulnerabilities handled by WARP.

CVE	POIROT		Naïve		WARP	
	# Req	Time (s)	# Req	Time (s)	# Req	Time (s)
2011-4360	100k	267	100k	23,900	100k	~121,000
2011-0537	100k	269	100k	23,700	100k	~121,000
2011-0003	100k	989	100k	25,100	100k	~121,000
2007-1055	100k	1,013	100k	24,300	100k	~121,000
2007-0894	100k	236	100k	31,500	100k	~121,000
12 cases (*)	0	0.03–0.11	100k	~25,000	100k	~121,000
17 cases (†)	0	0.02–0.19	100k	~25,000	0	< 1 second

* 2011-1766, 2010-1647, 2011-1765, 2011-1587, 2011-1580, 2011-1578, 2008-5688, 2008-5249, 2011-1579, 2011-0047, 2010-1189, 2008-4408.

† 2011-4361, 2010-2789, 2010-2788, 2010-2787, 2010-1648, 2010-1190, 2010-1150, 2009-4589, 2009-0737, 2008-5687, 2008-5252, 2008-5250, 2008-1318, 2008-0460, 2007-4828, 2007-0788, 2004-2186.

Table 3.4: POIROT’s auditing performance with 34 patches for MediaWiki vulnerabilities, compared with the performance of the naïve re-execution scheme and with WARP’s estimated repair performance for the same patches using its file-based auditing scheme (WARP’s re-execution of a request during repair is estimated to take 10× the original execution time, based on our evaluation of WARP’s repair performance in §2.7.5). WARP takes less than a second to access its index for file-based auditing. Naïve results are measured only for the top 5 patches; its performance would be similar for the 29 other patches.

3.7.4 Auditing performance

To show POIROT’s auditing performance, we used POIROT to audit the Wikipedia 100k workload for 34 real MediaWiki security patches, released between 2004 and 2011. We ported each patch to one of three major versions of MediaWiki released during this time period. We ran the workload against the three MediaWiki versions, which took an average of 12,116 seconds (3.4 hours) to execute during normal operation. POIROT’s indexing took on average 79 seconds for this workload. We measured the time taken by POIROT to audit all requests for these patches, the time taken by a naïve scheme that simply re-executes every request twice—with and without the patch—and compares the outputs, and the estimated time taken by WARP to repair the requests it considers as attack suspects.

Table 3.4 shows the results. For the bottom 29 out of 34 patches (85% of the vulnerabilities), POIROT’s control flow filtering took less than 0.2 seconds to determine that the patched code was not invoked by the workload requests, thereby completing the audit within that time. This is compared to the more than 6.5 hours needed to audit using the naïve re-execution scheme.

Patch	Description	Detected?	False +ves
f30eb4e5	Capability token lets users see restricted comments.	✓	0
638966eb	Chair can view an anonymous reviewer's identity.	✓	0
3ff7b049	Acceptance decisions visible to all PC members.	✓	0
4fb7ddee	Chair-only comments are exposed through search.	✓	0

Table 3.5: POIROT detects information leak vulnerabilities in HotCRP found between April 2011 and April 2012. We exploited each vulnerability and audited it with patches from HotCRP’s git repository (commit hashes for each patch are shown in the “patch” column).

POIROT audits the remaining five challenging patches, which affect code executed by every request, 24–133× faster than naïve re-execution (top 5 rows in Table 3.4). This means that POIROT can audit 3.4 hours worth of requests in ~17 minutes in the worst case.

To evaluate the improvement in WARP’s repair performance if its file-based auditing were replaced with POIROT’s auditing, we estimated WARP’s repair performance on the 100k workload for the 34 patches; the results are shown in Table 3.4. These estimates are based on our evaluation of WARP’s repair performance (§2.7.5), which showed that WARP’s repair re-executes a request ~10× slower than original execution. WARP’s file-level filtering statically discards all requests for 17 out of the 34 patches, although it is unable to filter out requests for 12 patches that POIROT’s basic-block-level filtering can. For the remaining 17 patches, file-level filtering flags all requests as suspect (even though the workload contained no attack requests) and WARP re-executes all of them, taking 2–3 orders of magnitude more time than POIROT, even for the worst-case patches; so, for our 3.4 hour workload, WARP could take 1.4 days to process the workload for each of the 17 patches, instead of the less than 17 minutes taken by POIROT. These results show that, for real workloads where most requests are benign, replacing WARP’s file-based auditing with POIROT’s fast and precise auditing can significantly improve WARP’s repair performance, as POIROT quickly filters away benign requests and leaves WARP with just a few attack suspects to repair.

3.7.5 Technique effectiveness

Control flow filtering allows POIROT to quickly filter out unaffected requests (in under 0.2 seconds), as illustrated by the bottom 29 patches in Table 3.4. As vulnerabilities typically occur in rarely exercised code, we expect control flow filtering to be highly effective in practice.

For the five challenging patches where re-execution is necessary, function-level re-execution and early termination speed up re-execution, as shown in Table 3.6. The “Func-level re-exec” column shows that it is 1.3–3.4× faster than naïve re-execution, and the “early term. ops” column shows that early termination executes a fraction of the ~200k total instructions. For the CVE-2011-0003 vulnerability, the patched function is invoked towards the end of the request, making early termination less effective.

Memoized re-execution further reduces re-execution time, as shown in Table 3.6. In particular, template collapsing reduces the number of distinct templates from 834–844 to 1–589 (“collapsed CF groups” column), thereby reducing the amount of time spent in template generation (“template gen. time” column). Templates reduce the number of PHP opcodes that must be re-executed by 22–50×, compared to early termination, as illustrated by the “template ops” column. For the CVE-2007-1055 vulnerability, memoized re-execution time is high even though it uses a single template (for its one control flow group); this is because the patched function writes to many global variables, making serialization for comparison expensive.

CVE	Naïve re-exec (s)	Func-level re-exec (s)	# early term. ops	# collapsed CF groups	Collapse time (s)	Template gen. time (s)	# template ops	Memoized re-exec (s)
2011-4360	23,900	8,480	6,437 / ~200k	4 / 844	31.0	2.10	289	234
2011-0537	23,700	18,900	4,801 / ~200k	1 / 834	30.3	1.17	96	238
2011-0003	25,100	19,600	117,045 / ~200k	589 / 834	30.5	395.00	5,427	563
2007-1055	24,300	7,150	5,571 / ~200k	2 / 844	30.1	0.83	177	982
2007-0894	31,500	10,500	24,973 / ~200k	18 / 844	30.4	9.90	1,085	196

Table 3.6: Performance of the POIROT replayer in re-executing all the 100k requests of the Wikipedia 100k workload, for the five patches shown here. The workload has a total of 834 or 844 control flow groups, depending on the MediaWiki version to which the patch was ported. POIROT incurs no false positives for four out of the five patches; it has 100% false positives for the patch 2011-0003, which fixes a clickjacking vulnerability. The “naïve re-exec” column shows the time to audit all requests with full re-execution and the “func-level re-exec” column shows the time to audit all requests with function-level re-execution and early termination. The “early term. ops” column shows the average number of PHP instructions executed up to the last patched function call with early termination (§3.4.2) across all the control flow groups. The “collapsed CF groups” and “collapse time” columns show the number of collapsed control flow groups and the time to perform collapsing of the control flow groups (§3.5.4), respectively. The “template gen. time”, “template ops”, and “memoized re-exec” columns show the time taken to generate templates for all the control flow groups in the workload, the average number of PHP instructions in the generated templates, and the time to re-execute the templates for all the requests, respectively.

Chapter 4

Recovery for distributed web services

This chapter describes AIRE [18], a system that extends WARP’s recovery algorithm to repair distributed attacks that spread between multiple web applications. In the past few years, it has become commonplace for web applications to often interact with one another to exchange data, to simplify user authentication, or to automate tasks that span multiple applications. For example, many web applications use Facebook’s OAuth service to allow users to log in with their Facebook credentials, and to access the user’s Facebook profile or wall postings. Salesforce similarly provides business users with the ability to combine multiple web applications into a single business workflow. Finally, a number of sites run short user-written scripts to integrate user information across many web applications [34, 38, 77, 78].

However, this integration comes at a price: if one of the web services is compromised, an adversary may take advantage of this integration to propagate the attack to other services; manually tracking down the extent of the attack and recovering all of the affected web services would be difficult at best. For example, consider the attack scenario in Chapter 1 (§1.3), where an attacker exploited a vulnerability in a spreadsheet application that stores access control lists for other applications, modified the access control lists to gain access to the other applications, and corrupted their data. Even though WARP can be used to repair each individual application’s local state (once the attack requests that targeted the application are identified), the administrator of each affected application still needs to manually determine whether its communication with other applications changed due to repair. If it did, the attack could have spread to the other applications, and the administrator needs to manually coordinate repair with the other applications’ administrators. This can be tedious, time-consuming, and error-prone, and missing the spread of an attack can lead to incomplete recovery.

Though we do not yet know of a real attack similar to the example above, several recent vulnerabilities in real web services [3, 6, 30–32] could be used to launch such distributed attacks. For example, a recent Facebook OAuth bug [31] allows an attacker to obtain a fully privileged OAuth token for any user, as long as the user mistakenly follows a link supplied by the attacker; the attacker could use this token to corrupt the user’s Facebook data, which other applications may access, spreading the attack further.

AIRE recovers from such distributed attacks by automatically propagating repair to all affected applications. AIRE’s key contribution is in making recovery across web services practical by addressing three challenges, as follows:

Asynchronous repair. Propagating repair across services raises two issues. First, there is no strict hierarchy of trust among web services and services do not expose their internal state to each other; so, there is no single system that can be trusted to orchestrate repair across multiple services. Second, during repair, some services affected by an attack may be down, unreachable, or otherwise unavailable. Waiting for all services to be online in order to perform repair would be impractical, and may unnecessarily

delay recovery in services that are already online. Worse yet, an adversary may purposely add their own server to the list of services affected by an attack, in order to prevent timely recovery.

AIRE solves these issues with two ideas. First, to avoid services having to expose their internal state and to minimize changes to existing services, AIRE's repair of a service is specified using the service's API. AIRE automatically extends each service's API to define a repair protocol that allows services to invoke repair on their past requests and responses to other services. Second, to quickly repair services after an intrusion without waiting for unavailable services, AIRE performs *asynchronous* repair by decoupling the local repair on a single service from the repair of its interactions with other services. A service repairs its local state as soon as it is asked to perform a repair, and if any past requests or responses are affected, it queues a repair message for other services, which can be processed when those services become available.

Repair access control. AIRE must ensure that its repair protocol does not give attackers new ways to subvert web services. To this end, AIRE enforces access control on every repair invocation. As access control policies can be application-specific, AIRE allows applications to define their own policies, and delegates access checks to applications using an interface designed for this purpose. If an application does not define an access control policy, AIRE falls back to a default policy that allows a currently valid user to repair a past request issued by the same user; this default policy makes sense in many applications because the user can anyways manually repair the service in the absence of AIRE by issuing appropriate API requests using her current privileges.

Reasoning about partially repaired state. With asynchronous repair, some services affected by an attack can be already repaired, while others have not yet received or processed their repair messages. In some services, such a partially repaired state can appear corrupted to clients and lead to unexpected application behavior. To address this challenge, we model repair of a service as requests invoked by a *repair client* on the service's API in the current time; if a service's API and its clients support this repair model, then the service's partially repaired state appears valid to its clients. This model reduces the problem of dealing with partially repaired states to the existing problem of dealing with concurrent clients; as web application developers already build web services to handle concurrent clients, this makes it easy for them to reason about partially repaired state. We show that real, loosely-coupled web services support partially repaired state with little or no code changes.

The rest of this chapter is structured as follows. §4.1 provides an overview of AIRE's design. §4.2 describes AIRE's distributed repair, §4.3 discusses AIRE's access control, §4.4 presents a repair model to reason about partial repair state, and §4.5 discusses our prototype implementation. §4.6 provides a case study of using AIRE for intrusion recovery in several attack scenarios, and §4.7 evaluates the performance overheads of AIRE.

4.1 Overview

AIRE's goal is to recover from intrusions or user mistakes that spread across multiple web services. AIRE uses WARP for each affected service's local repair. AIRE assumes that repair starts at the service with the root cause exploited by the attack (e.g., the spreadsheet application with the XSS bug in the attack example described earlier); this service was the entry point of the attack and repair is initiated on it as with WARP (§2.2). If during local repair of a service, AIRE determines that requests or responses to other services may have been affected, AIRE asynchronously sends repair messages to those services informing them of the affected requests and responses, along with a corrected version of the requests and responses.

AIRE on each service that receives a repair message initiates local recovery in turn, after authorizing the repair message. Once repair messages propagate to all affected services, the attack's effects will be removed from the entire system. However, even before repair messages propagate everywhere, applications that are already online can repair their local state.

While repair is propagating through different web services, an application using these services may observe the overall system in a partially repaired state. For arbitrary distributed applications, this could result in the state appearing as invalid from the application's point of view. However, AIRE is designed for recovery in loosely-coupled web services, and we show that most of them can handle partial repair states with little or no source code modifications.

In the rest of this section we describe scenarios motivating the need for AIRE, and present AIRE's system model and architecture.

4.1.1 Motivating scenarios

In the past few years, it has become common for a web application to use well-defined web service APIs to export its functionality, which was previously available only through the application's UI. This has made it possible for services to integrate with each other, and has opened up many use cases. Here we describe a few example scenarios that illustrate the benefits of web service integration and motivate the need for repair in each scenario.

Single sign-on. Many web applications rely on third-party providers like Google, Facebook, Yahoo, and Twitter to authenticate users via protocols such as OpenID or OAuth. This removes the need for users to manage a separate account on each web site, but at the same time, if an adversary compromises the user's primary account, as was recently possible to do with Facebook [30, 31], the attack can now spread to many other sites. AIRE can help track down and recover from attacks that spread across different sites the user accesses.

Scripting services. Several web sites allow users to set up server-side scripts to automate tasks across the user's many web applications [34, 38, 77, 78]. For example, Zapier [78] allows a user to create a script to automatically update a Google Spreadsheet with her new Github issues. Although this is convenient for end users, a significant risk is that an attacker can install a script of his own, thereby retaining access even after the initial entry point of the attack has been repaired. AIRE is able to track down and repair all side effects of the attack, including scripts installed by an adversary.

Shared platforms. A number of web "platforms," such as Facebook, Google Drive, or Dropbox, help third-party applications access, store, and exchange user data. Business users similarly use platforms provided by Salesforce, Workday, and Concur to combine multiple third-party web applications into unified business workflows. This enables interaction between different web applications used by the same user, but also increases the risk of a single compromised application affecting the user's other applications; indeed, checking whether an untrustworthy application has access to the user's shared data is already common security practice [41, 70]. AIRE can help users deal with such scenarios by tracking down what other applications might be affected, and repairing them.

Developer workflow. Developers often set up their development workflows by integrating several web services, perhaps using Github for version control, Jira for project management, FogBugz for bug tracking, and Amazon EC2 for building and testing. AIRE can help developers recover even if one of these services is compromised.

4.1.2 System model

AIRE assumes that the system consists of a set of HTTP web services, with each service exposing a set of operations that can be performed on it, using a well-defined public API (e.g., a REST API [26]). Clients of a service cannot directly access its internal state; they can change the service's state only by performing operations using its public API. This model of web services is commonplace today, and some well-known examples of such services are Amazon S3, Facebook, Google Docs, and Dropbox.

Under this model, an attack is an API operation that exploited a vulnerability or misconfiguration in a service and caused unauthorized changes to the service's state. These state changes could have affected other operations, both on this service as well as on other services, perhaps causing more state changes on the services. Repair should undo all the state changes resulting from the attack.

An important issue during repair is how services propagate repair to each other. One option is for services to expose their internal state to each other so that they can rollback and repair each other's state. However, as services cannot access each other's internal state during normal operation, a key goal of AIRE's design was to ensure that such access is not needed for repair as well. Instead, repair is invoked at the API-level and AIRE is designed to repair past API operations. AIRE automatically extends the public API of each service and defines additional operations to repair past operations, while adhering to the service API's format. This allows services to use each other's APIs to invoke both normal operations and repair, without needing access to their internal states.

Conceptually, the effect of API-level repair is as follows: when a client determines that its past API operation to a service was incorrect and uses AIRE's repair API to fix the mistake, AIRE mutates the state of the system to what it would have been if the mistake had never happened. AIRE performs this mutation with rollback-redo for local repair, and by propagating repair to other affected services using their repair API.

With API-level repair, AIRE can recover from attacks that exploit misconfigurations of a web service or vulnerabilities in a service's code, and spread through HTTP requests between web services. This includes several scenarios such as the ones in §4.1.1. Additionally, AIRE can repair corruptions that result from user mistakes as well.

4.1.3 AIRE architecture

Figure 4-1 provides an overview of AIRE's overall design. Every web service that supports repair through AIRE runs an AIRE repair controller, which extends WARP's repair controller. The repair controller maintains a repair log during normal operation by intercepting the original service's requests, responses, and database accesses. The repair controller also performs repair operations as requested by users, administrators, or other web services, by rolling back affected state and re-executing affected requests.

In order to be able to repair interactions between services, AIRE intercepts all HTTP requests and responses to and from the local system. Repairing requests or responses later on requires being able to name them; to this end, AIRE assigns an identifier to every request and response, and includes that identifier in an HTTP header. The remote system, if it is running AIRE, records this identifier for future use if it needs to repair the corresponding request or response.

During repair, if AIRE determines that the local system sent an incorrect request or response to another service, it computes the correct request or response, and sends it along with the corresponding ID to the other service. AIRE's repair messages are implemented as just another REST API on top of HTTP. AIRE supports four kinds of operations in its repair API. The two most common repair operations involve replacing either a request or response with a different payload. Two other operations arise when AIRE determines that the local service should never have issued a request in the first place, or that it

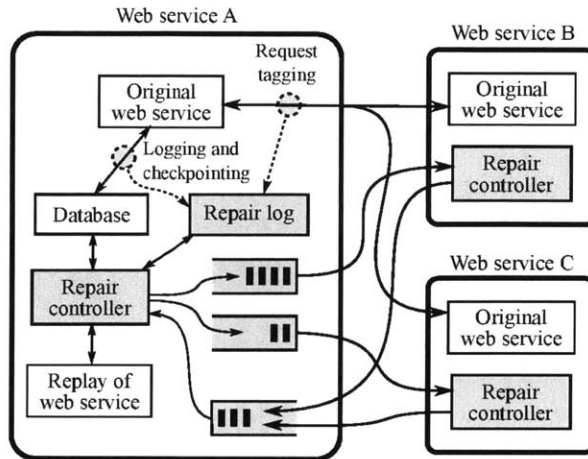


Figure 4-1: Overview of AIRE's design. Components introduced or modified by AIRE are shaded. Circles indicate places where AIRE intercepts requests from the original web service. Not shown are the detailed components for services B and C.

should have issued a request while none was originally performed; in these cases, AIRE asks the remote service to either cancel a past request altogether, or to create a new request.

When repairing a request, AIRE updates its repair log just like it does during normal operation, so that a future repair can perform rollback recovery on an already repaired request. This is important because asynchronous repair can cause a request to be repaired several times as repair propagates through all the affected services.

AIRE must control who can issue repair operations, to ensure that clients or other web services cannot make unauthorized changes via the repair interface. AIRE delegates this access control decision to the original service, as access control policies can be service-specific: for example, a service can require a stronger form of authentication (e.g., Google's two-step authentication) when a client issues a repair operation than when it issues a normal operation; or a platform such as Facebook can block repair requests from a third-party application if the application is attempting to modify the profile of a user that has since uninstalled that application.

To make implementing access control easy for services that use a standard authentication framework (Django in the current prototype), AIRE provides a default policy that allows a past request to be repaired only if repair is issued by the same principal that originally issued the request; services can choose to use or extend this default policy.

In some cases, appropriate credentials for issuing a repair operation on another web service may not be available. For example, AIRE on a service may need to repair a request it previously issued on behalf of a user to a remote service; however, it cannot invoke repair if the remote service requires the user's credentials for repair and the user is not currently logged in. AIRE treats this situation as if the remote service is not available, and queues the repair for later. Once the user logs in, the service can use the user's credentials to propagate repair.

4.2 Distributed repair

This section describes AIRE's design in more detail, focusing on how AIRE achieves asynchronous repair among distributed web services.

Command and parameters	Description
<code>replace</code> (<i>request_id</i> , <i>new_request</i>)	Replaces past request with new data
<code>delete</code> (<i>request_id</i>)	Deletes past request
<code>create</code> (<i>request_data</i> , <i>before_id</i> , <i>after_id</i>)	Executes new request in the past
<code>replace_response</code> (<i>response_id</i> , <i>new_response</i>)	Replaces past response with new data

Table 4.1: AIRE’s repair interface.

4.2.1 Repair protocol

Each AIRE-enabled web service exports a repair interface that its clients (including other AIRE-enabled web services) can use to initiate repair on it. AIRE augments the interface of any existing web service to do this, without requiring any effort on the part of the developer of the service. AIRE’s repair interface is summarized in Table 4.1, and we will now describe its components in more detail.

Request naming. In order to name requests and responses during subsequent repair operations, AIRE must assign a name to every one of them. To do this, AIRE interposes on all HTTP requests and responses during normal operation, and adds headers specifying a unique identifier that will be used to name every request.

To ensure these identifiers uniquely name a request (or response) on a particular server, AIRE on the service handling the request (or receiving the response) assigns the identifier; it becomes the responsibility of the other party to remember this identifier for future repair operations. Specifically, AIRE adds an `Aire-Response-Id:` header to every HTTP request issued *from* a web service; this identifier will name the corresponding response. The server receiving this request will store the response identifier, and will use it later if the response must be repaired. Conversely, AIRE adds an `Aire-Request-Id:` header to every HTTP response produced by a web service; this identifier assigns a name to the HTTP request that triggered this response. A client can use this identifier to refer to the corresponding request during subsequent repair.

Repairing previous requests. AIRE’s repair begins when some client (either a user or administrator, or another web service) determines that there was a problem with a past request or response. To recover, the client uses the AIRE repair API in Table 4.1 to initiate repair on the corresponding web service.

The simplest operation is `replace`, which allows a client to indicate that a past request (named by its *request_id*) was incorrect, and should be replaced with *new_request* instead. The new request contains the corrected version of the arguments that were originally provided to the original request, including the URL, HTTP headers, query parameters, etc. When AIRE’s controller performs a `replace` operation, it repairs the local state to be as if the newly supplied request happened instead of the original request. If any HTTP requests or responses turn out to be affected by this repair, AIRE queues appropriate repair API calls for other web services.

The `delete` operation is similar to `replace`, but it is used when a client determines that it should not have issued some request at all. In this case, `delete` instructs the AIRE repair controller to eliminate all side-effects of the request named by *request_id*.

Finally, `create` allows a client to create a new request in the past, and `replace_response` allows a server to indicate that a past response to a client, named by its *response_id*, was incorrect, and to supply a corrected version of the response in *new_response*. We will discuss both of these operations in more detail shortly.

To make it easier for clients to use AIRE's repair interface, AIRE's repair API encodes the request being repaired (e.g., *new_request* for *replace*) in the same way as the web service would normally encode this request. The type of repair operation being performed (e.g., *replace* or *delete*) is sent in an *Aire-Repair: HTTP* header, and the *request_id* being repaired is sent in an *Aire-Request-Id:* header. Thus, to fix a previous request, the client simply issues the corrected version of the request as it normally would, and adds the *Aire-Repair: replace* and *Aire-Request-Id:* headers to indicate that this request should replace a past operation. In addition to requiring relatively few changes to client code, this also avoids introducing infrastructure changes (e.g., modifying firewall rules to expose a new service).

Creating new requests. Sometimes, repair requires adding a new request “in the past.” For example, if an administrator wants to recover from a mistake where she forgot to remove a user from an access control list when the user should have been removed, the appropriate way to repair from this mistake is to add a new request at the right time in the past to remove the user from the access control list. The *create* call allows for this scenario.

One challenge with *create* is in specifying the time at which the request should execute. Different web services do not share a global timeline, so the client cannot specify a single timestamp that is meaningful to both the client and the service. Instead, the client specifies the time for the newly created request relative to other messages it exchanged with the service in the past. To do this, the client first identifies the local timestamp at which it wishes the created request to execute; then it identifies its last request before this timestamp and the first request after this timestamp that it exchanged with the service, and instructs the service to run the created request at a time between these two requests. The *before_id* and *after_id* parameters to the *create* call name these two requests.

The above scheme is not complete, as it allows the client to specify order of the new request only with respect to past requests it exchanged with the service executing the new request. The client cannot specify precise ordering with respect to arbitrary messages in the system, and so this scheme cannot handle situations where there are causal dependencies across multiple services. Precise ordering would require services to exchange large vector timestamps or dependency chains, which can be costly; as we have not yet found a need for it, we have not incorporated it into AIRE's design.

Repairing responses. In web services, clients initiate communication to the server. However, to invoke a *replace_response* on a client, the service needs to initiate communication to the client. This raises two challenges. First, the server needs to know where to send the *replace_response* call for a client. To address this challenge, AIRE clients add an *Aire-Notifier-URL:* header to every request, along with the *Aire-Response-Id:* header. If the server wants to contact the client to repair the response, it sends a request to the associated notifier URL.

Second, once a client gets a *replace_response* call from a service, it needs to authenticate the service. This is typically done by verifying the server's SSL certificate. To address this challenge, AIRE clients treat any notifications sent to the notifier URL as a hint, which provides a *response repair token*; when a client receives a response repair token, it contacts the server and asks the server to provide the *replace_response* call for a particular response repair token. This way, the client can appropriately authenticate the server, such as by validating its SSL certificate. Once the client fetches the repaired response from the server, it repairs any past requests that originally read this response.

4.2.2 Asynchronous repair

AIRE's repair is *asynchronous*: it performs local repair on a service affected by an attack, without coordinating with other services or waiting for repair on other services to complete. This allows each

service to repair local state at its own pace, without blocking for other slow or currently unavailable services.

As part of local repair, AIRE re-executes operations affected by the attack. It is possible that one of these operations will execute differently due to repair, and issue a new HTTP request that it did not issue during the original execution. In that case, AIRE must issue a `create` repair call to the corresponding web service, in order to create a new request “in the past.” Re-execution can also cause the arguments of a previously issued request to change, in which case, AIRE queues a `replace` message to the remote web service in question. One difficulty with both `create` and `replace` calls is that to complete local repair, the application needs a response to the HTTP requests in these calls. However, AIRE cannot block local repair waiting for the response.

To resolve this tension, AIRE tentatively returns a “timeout” response to the application’s request, which any application must already be prepared to deal with; this allows local repair to proceed. Once the remote web service processes the `create` or `replace` call, it sends back a `replace_response`, which contains the new response. At this point, AIRE will perform another repair to fix up the response.

When re-execution skips a previously issued request altogether, AIRE queues a `delete` message. Finally, if re-execution changes the response of a previously executed request, or computes the response for a newly created request, AIRE queues a `replace_response` message.

AIRE maintains an outgoing queue of repair messages for each remote web service. If multiple repair messages refer to the same request or the same response, AIRE can collapse them, by keeping only the most recent repair message. Sometimes, AIRE may not be able to send a repair message, either because the original request or response did not include the dependency-tracking HTTP headers identifying the web service to send the message to, or because the communication to the remote web service timed out; in either case, AIRE places the repair message in a pending queue that the application can inspect (as discussed later) and notify the administrator, in case she wants to perform some manual compensating action. AIRE also aggregates incoming repair messages in an incoming queue, and can apply the changes requested by multiple repair operations as part of a single local repair.

4.3 Repair access control

Access control is important because AIRE’s repair itself must not enable new ways for adversaries to propagate from one compromised web service to another. For example, a hypothetical design that allows any client with a valid request identifier to issue repair calls for that request is unsuitable, because an adversary that compromises a service storing many past request identifiers would be able to make arbitrary changes to those past requests, affecting many other services; this is something an attacker would not be able to do in the absence of AIRE.

AIRE delegates access control decisions to a handler provided by the web service; if a service does not provide an access control handler, AIRE falls back to a default access control handler. This section describes how AIRE’s access control delegation works and the default access control policy implemented by AIRE.

4.3.1 Delegation of access control

AIRE requires that every repair API call be accompanied with explicit credentials required to authorize the repair operation. AIRE delegates access control decisions to the application because principal types, format of credentials, and access control policies can be application-specific. For example, some applications may use cookies for authentication while others may include an access token as an additional

Function and parameters	Description
Interface exported to AIRE by services receiving repair messages	
<code>authorize (repair_type, original, repaired)</code>	Checks if a repair message should be allowed
Interface exported by AIRE to services sending repair messages	
<code>pending ()</code>	Returns undelivered repair messages
<code>retry (updated_message)</code>	Resends a repair message

Table 4.2: AIRE’s access control interface with web services.

HTTP header; and some applications may allow any user with a currently valid account to repair a past request issued by that user, while others may allow only users with special privileges to invoke repair.

The access control interface between AIRE and the services running it is shown in Table 4.2. Services running AIRE export an `authorize` function that AIRE invokes when it receives a repair message; AIRE passes the function the type of repair operation (`create`, `replace`, `delete`, or `replace_response`), and the original and new versions of the request or response to repair (denoted by the *original* and *repaired* parameters). The `authorize` function’s return value indicates whether the repair should be allowed; if the repair is not allowed, AIRE returns an authorization error for the repair message.

To perform access control checks, the service may need to read old database state (e.g., to look up the principal that issued the original request). For this purpose, AIRE provides the application read-only access to a snapshot of the application’s database at the time when the request originally executed; AIRE can do this because it versions the application’s database in a manner similar to WARP’s time-travel database (§2.3). Once a repair operation is authorized, AIRE re-executes the new request, if any. As part of request re-execution, the application may apply other authorization checks, in the same way it does for any other request during normal operation.

When a client invokes repair, AIRE requires the client to include credentials in the repair message, much like the client needs to during normal execution. In some cases, the repair message may fail the server’s authorization check. For example, a server and a client may use OAuth 2.0 for authorization, which supports an expiration time for tokens that users issue to clients. When the client issues a repair message, it may have a stale OAuth token for the user on whose behalf the original request was issued, causing the server to reject the repair message.

AIRE places repair messages that fail authorization in a pending queue, while it continues performing repair requested by subsequent authorized messages. AIRE provides a `pending` function to the client application to fetch messages in the queue, along with errors returned by the server. Once the application fixes the credentials in a pending repair message, it can use the `retry` function to ask AIRE to resend the message. In the OAuth example above, when a user logs in, the client application could display the repair messages pending due to stale OAuth tokens of the user, and for each repair message, prompt the user for a fresh OAuth token or to cancel the message altogether. If the user supplies a fresh token, the application can ask AIRE to resend the repair message.

4.3.2 Default access control policy

For applications that use a standard authentication framework (e.g., Django’s default authentication system) and do not provide their own `authorize` function, AIRE implements a default access control policy that requires a repair message to provide current credentials for the same principal that issued the original request. This policy requires any potential adversary issuing repair calls to already possess valid credentials for the appropriate user, and thus the damage from an adversary using AIRE’s repair API is no worse than the adversary using the user’s credentials directly. AIRE’s default policy also

allows `replace_response` operations if the server's SSL credentials are valid, and it disallows `create` operations, as there is no corresponding request in the past to check the principal against. For some applications disallowing creates is not a problem, as deleting past attack requests and replacing a mistake in a past request's arguments with the correct arguments are sufficient for many repairs. Applications can use AIRE's default policy as is or can extend it for their purposes.

4.4 Understanding partially repaired state

AIRE's asynchronous repair exposes the state of a service to its clients after its local repair is done, without waiting for repair to complete on other affected services. In principle, for a distributed system composed of arbitrary tightly-coupled services, partially repaired state can appear invalid to clients of the services. For example, if one of the services is a lock service, and during repair it grants a lock to a different application than it did during original execution, then in some partially repaired state both the applications could be holding the lock; this violates the service's invariant that only one application can hold a lock at any time, and can confuse applications that observe this partially repaired state.

However, AIRE is targeted at web services, which are loosely-coupled, perhaps because they are under different administrative domains and cannot rely on each other to be always available. In practice, for such loosely-coupled web service APIs, exposing partially repaired state does not violate their invariants; we call such APIs as *compatible* with partially repaired state. In the rest of this section, we first present a model to reason about partially repaired state and show that many of today's web service APIs are compatible. Then we show how we can extend the design of an API that is not compatible to make it compatible.

4.4.1 Modeling repair as API invocations

Many web services and their clients are designed to deal with concurrent operations, and so web application developers already have to reason about concurrent updates. For example, Amazon S3, a popular web service offering a data storage interface, supports both a simple PUT/GET interface that provides last-writer-wins semantics in the face of concurrency, and an object versioning API that helps clients deal with concurrent writes.

Building on this key observation, we propose modeling repair of a service as a *repair client* performing normal API calls to the service, in the present time. A service whose API and clients support modeling repair in this manner (as another concurrent client) is compatible with partially repaired state. Then application developers need to only reason about concurrent updates, which is a well-understood problem, rather than having to reason about all possible timelines in which concurrent repair operations are happening.

This model is already supported by many web services. For example, consider the scenario in Figure 4-2, illustrating operations on object *X* stored in Amazon S3. Initially, *X* had the value *v0*. At time *t1*, an attacker writes the value *v1* to *X*. At time *t2*, client *A* reads the value of *X* and gets back the value *v1*, and at time *t3* the client reads the value of *X* again. In the absence of repair or any concurrent operations, *A* should receive the value *v1*, but what should happen if, between *t2* and *t3*, Amazon S3 determines the attacker's write was unauthorized and deletes that request?

If repair occurs between *t2* and *t3*, the state of *X* will roll back to *v0*, and two things will happen with *A*: first, it will receive *v0* in response to its second request at *t3*, and second, at some later time it will receive a `replace_response` from S3 that provides the repaired response for the first `get`, also containing *v0*. Client *A* observes partially repaired state during the time between when local repair on S3 completed (which is sometime between *t2* and *t3*) and when *A* finally receives the `replace_response`

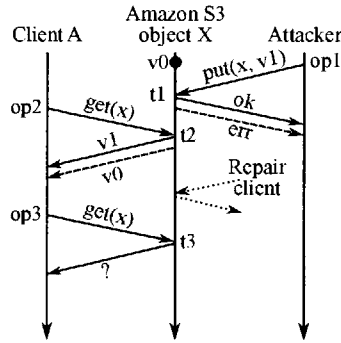


Figure 4-2: Example scenario demonstrating modeling repair actions as concurrent operations by a repair client. Solid arrows indicate requests during original execution, and dashed arrows indicate eventual repair propagation. The S3 service initiates local repair in between times t_2 and t_3 by deleting the attacker’s put. If S3’s local repair completes before t_3 , `op3` observes value v_0 for X . If A has not yet received the propagated repair from S3, receiving the value v_0 for X at time t_3 is equivalent to a concurrent writer (the hypothetical repair client) doing a concurrent `put(x, v0)`.

message; A sees this state as valid (with its first `get(x)` returning v_1 and its second `get(x)` returning v_0), because a hypothetical repair client could have issued a `put(x, v0)` in the meantime.

4.4.2 Making service APIs repairable

A web service that offers a simple PUT/GET interface is trivially compatible with partially repaired state, because clients cannot make any strict assumptions about the state of the service in the face of concurrency. However, some web service APIs may provide stronger invariants that may make them incompatible. For example, several existing web services provide a versioning API that guarantees an immutable history of versions (§4.6.3); these services are incompatible with partially repaired state. Suppose the client A from our earlier example in Figure 4-2 asked the server for a list of all versions of X , instead of a `get` on the latest version. At time t_2 , A would receive the list of versions (v_0, v_1) . If repair simply rolled back the state of X between t_2 and t_3 , A would receive the list of versions (v_0) at time t_3 with v_1 removed from the list, a state that no concurrent writer could have produced using the versioning API.

Incompatible services and clients need to be modified to make them compatible, so that they are repairable using AIRE; the rest of this section describes how we can make the versioning API in today’s web services compatible.

Consider a web service API that provides a single, linear history of versions for an object. Once a client performs a `put(x, v1)`, value v_1 must appear in the history of values for x (until old versions are garbage-collected). If the `put(x, v1)` turns out to be erroneous and needs to be repaired, what partially repaired state can the service expose? Removing v_1 from the version history altogether would be inconsistent if the service does not provide any API to delete versions from history, and might confuse clients that rely on past versions to be immutable. On the other hand, simply appending new versions to the history (i.e., writing a new fixed-up value) prevents AIRE from repairing past responses. In particular, if a past request asked for a list of versions, AIRE would have to send a new list of versions to that client (using `replace_response`) where the effects of v_1 have been removed. However, if AIRE extends that past version history by appending a new version that reverts v_1 , this synthesized history would in turn be inconsistent with the present history.

One way to make the versioning APIs compatible with partial repair is to extend them to support branches, similar to the model used by the git version control system. With an API that supports branches, when a past request needs to be repaired, AIRE can create a new branch that contains a repaired set

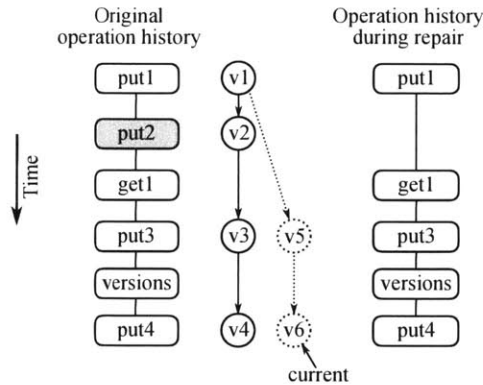


Figure 4-3: Repair in a versioned service. The shaded operation `put2` from the original history, shown on the left, is deleted during repair, leading to the repaired history of operations shown on the right. The version history exposed by the API is shown in the middle, with two branches: the original chain of versions, shown with solid lines, and the repaired chain of versions, dotted. The mutable “current” pointer moves from one branch to another as part of repair.

of changes, and move the “current” pointer to the new branch, while preserving the original branch. This makes the API compatible and has the added benefit of preserving the history of all operations that happened, including mistakes or attacks, instead of erasing mistakes altogether.

For example, consider a simple key-value store that maintains a history of all values for a key, as illustrated in Figure 4-3. During repair, request `put2` is deleted. An API with linear history would not be compatible with partial repair, but a branching API can. With branches, repair creates a new branch (shown in the right half of Figure 4-3), and re-applies legitimate changes to that branch, such as `put3`. These changes will create new versions on the new branch (such as `v5` mirroring the original `v3`); the application must ensure that version numbers themselves are opaque identifiers, even though we use sequential numbers in the figure. At the end of local repair, AIRE exposes the repaired state, with the “current” branch pointer moved to the repaired branch. This change is consistent with concurrent operations performed through the regular web service API.

For requests whose responses changed due to repair, AIRE sends `replace_response` messages that contain the new responses; the new response for a `get` request is the repaired value at the logical execution time of the request, and the new response for a `versions` request contains only the versions created before the logical execution time of the request. In the example of Figure 4-3, the new response for `get1` is `v1`, while the new response for the `versions` call is `(v1, v2, v3, v5)`, and does not contain `v4` and `v6`.

We explore the reparability of common web service APIs in §4.6.3, and describe how AIRE implements repair for the versioned API example above in §4.5.

4.5 Implementation

We implemented a prototype of AIRE for the Django web application framework [2]. AIRE leverages Django’s HTTP request processing layer, its object-relational mapper (ORM), and its user authentication mechanism. The ORM abstracts data stored in an application’s database as Python classes (called “models”) and relations between them; an instance of a model is called a model object.

We modified the Django HTTP request processor and the Python `httplib` library functions to intercept incoming and outgoing HTTP requests, assign IDs to them, and record them to the repair log. AIRE versions model objects in a manner similar to WARP’s time-travel database, so that the model objects can be rolled back during repair. To implement this, we modified the Django ORM to intercept the

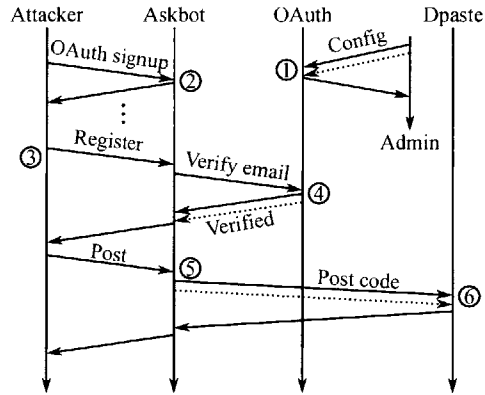


Figure 4-4: Attack scenario in Askbot demonstrating AIRE’s repair capabilities. Solid arrows show the requests and responses during normal execution; dotted arrows show the AIRE repair operations invoked during recovery. Request ① is the configuration request that created a vulnerability in the OAuth service, and the attacker’s exploit of the vulnerability results in requests ②-⑥. For clarity, requests in the OAuth handshake, other than request ②, have been omitted.

application’s reads and writes to model objects. On a write to a model object, AIRE transparently creates a new version of the object, and on a read, it fetches the latest version during normal execution and the correct past version during repair. AIRE rolls back a model object to time t by deleting all versions after t . In addition to tracking dependencies between writes and reads to the same model, AIRE also tracks dependencies between models (such as unique key and foreign key relationships) and uses them to propagate repair. We implemented AIRE’s default access control policy in Django’s default authentication framework. We modified 2953 lines of code in Django to implement the AIRE interceptors; the AIRE repair controller was another 2758 lines of Python code.

Repair for a versioned API. If a service’s API implements versioning, it indicates this to AIRE by making the model class for its versioned data a subclass of AIRE’s `VersionedModel` class. `VersionedModel` objects are not rolled back during repair, and AIRE does not perform versioning for these objects. If other non-versioned model objects store references to `VersionedModel` objects, AIRE rolls them back during repair.

4.6 Application case studies

This section answers the following questions:

- What kinds of attacks can AIRE recover from?
- How much of the system is repaired if some services are offline or authorization fails at a service?
- What happens to the system’s invariants if some services are offline during repair?
- How much effort is required to port an existing application to run with AIRE?

4.6.1 Intrusion recovery

As we do not know of any significant compromises that propagated through distributed web services to date, to evaluate the kinds of attacks that AIRE can handle, we implemented four challenging attack scenarios and attempted to repair from each attack using AIRE. The rest of this subsection describes these scenarios and how AIRE handled the attacks.

Askbot. A common pattern of integration between web services is to use OAuth or OpenID providers like Facebook, Google, or Yahoo, to authenticate users. If an attacker compromises the provider, he can spread an attack to services that depend on the provider. To demonstrate that AIRE can repair from such attacks, we evaluated AIRE using real web applications, with an attack that exploits a vulnerability similar to the ones recently discovered in Facebook [30, 31].

The system for the scenario consists of three large open-source Django web services: Askbot [1], which is an open-source question and answer forum similar to Stack Overflow and used by sites like the Fedora Project; Dpaste, a Django-based pastebin service; and a Django-based OAuth service. These three services together comprise 183,000 lines of Python code, excluding blank lines and comments. Askbot maintains a significant amount of state, including questions and answers, tags, user profiles, ratings, and so on, which AIRE must repair.

For this scenario, we configured Askbot to allow users to sign up using accounts in an external OAuth service that we set up for this purpose. We also modified Askbot to integrate with Dpaste; if a user's Askbot post contains a code snippet, the Askbot service posts this code to the Dpaste service for easy viewing and downloading by other users. Finally, the Askbot service also sends users a daily email summarizing that day's activity. These loosely coupled dependencies between the services mimic the dependencies that real web services have on each other.

The attack in the scenario, shown in Figure 4-4, exploits an OAuth vulnerability during Askbot's user signup. After Askbot gets an OAuth token for a user with the standard OAuth handshake (omitted from the figure), Askbot prompts the user for her email and verifies the email with the OAuth service before allowing signup to proceed. To simulate the recent Facebook vulnerability [31], we added a debug configuration option in OAuth that always allows email verification to succeed. This option is mistakenly turned on in production by the administrator by issuing request ①, thus exposing the vulnerability. The attacker exploits this vulnerability in OAuth to sign up with Askbot as a victim user and post a question with some code, thereby spreading the attack from OAuth to Askbot. Askbot automatically posts this code snippet to Dpaste, spreading the attack further to Dpaste; later, a legitimate user views and downloads this code from Dpaste. At an even later time, Askbot sends a daily email summary containing the attacker's question, creating an external event that depends on the attack. Before, after, and during the attack, other legitimate users continue to use the system, logging in, viewing and posting questions and answers, and downloading code from the Dpaste service. Some actions of these legitimate users, such as posting their own questions, are not dependent on the attack, while others, such as reading the attacker's question, are dependent.

After the attack, we used AIRE to recover from it. The administrator starts repair by invoking a `delete` operation on request ①, which introduced the vulnerability. This is shown by the dotted arrow on request ① in Figure 4-4. This initiates local repair on OAuth, which deletes the misconfiguration, and invokes a `replace_response` operation on request ④ with an error value for the new response. The `replace_response` propagates repair to Askbot: as requests ③ and ⑤ depend on the response to request ④, local repair on Askbot re-executes them using the new error response, undoing the attacker's signup and the attacker's post. Local repair on Askbot also runs a compensating action for the daily summary email, which notifies the Askbot administrator of the new email contents without the attacker's question; it also re-executes all legitimate user requests that depended on the attack requests; and finally it invokes a `delete` operation on Dpaste to cancel request ⑥. Dpaste in turn performs local repair, resulting in the attacker's code being deleted, and a notification being sent to the user who downloaded the code. This completes recovery, which removes all the effects of the attack and does not change past legitimate actions in the system that were not affected by the attack.

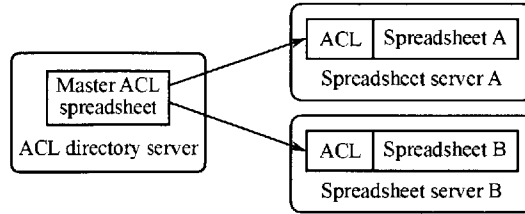


Figure 4-5: Setup for the spreadsheet application attack scenarios.

Lax permissions. A common source of security vulnerabilities comes from setting improper permissions. In a distributed setting, we consider a scenario where one service maintains an authoritative copy of an access control list, and periodically updates permissions on other services based on this list, similar to the attack example presented in §1.3. If a mistake is made in the master list, it is important to not only propagate the change to other services, but to also undo any requests that took advantage of the mistake on those services.

Since Askbot did not natively support such a permission model, we implemented our own spreadsheet service for this scenario. The spreadsheet service has a simple scripting capability similar to Google Apps Script [34]. This allows a user to attach a script to a set of cells, which executes when values in cells change. We use scripting to implement a simple distribution mechanism for access control lists (ACLs). The setup is shown in Figure 4-5. The ACL directory is a spreadsheet service that stores the master copy of the ACL for the other two spreadsheets. A script on the directory updates the ACLs on the other services when master list is modified.

The attack is as follows: an administrator mistakenly adds an unauthorized attacker to the ACL by issuing a request to update the master ACL spreadsheet; the ACL script distributes the new ACL to spreadsheets A and B. Later, the attacker takes advantage of these extra privileges to corrupt some cells in both spreadsheets. All this happens while legitimate users are also using the services.

Once the administrator realizes her mistake, she initiates repair by invoking a `delete` operation on the ACL directory to cancel her update to the ACL spreadsheet. The ACL directory reverts the update, and invokes `delete` on the two requests made by its script to distribute the corrupt ACL to the two services. This causes local repair on each of the two services, which results in rollback of the corrupt ACL. All the requests since the corrupt ACL's distribution are re-executed, as every request to the service checks the ACL. As the attacker is no longer in the ACL, his requests fail, whereas the requests of legitimate users succeed; AIRE's repair thereby cleans up the attacker's corrupt updates while preserving the updates made by legitimate users.

Lax permissions on the configuration server. A more complex form of the above attack could take place if the directory server itself is misconfigured. For example, suppose the administrator does not make any mistakes in the master ACL spreadsheet, but instead accidentally makes the master spreadsheet world-writable. An adversary could then add himself to the master ACL, wait for updates to propagate to A and B, and then modify data in those spreadsheets as above.

Recovery in this case is more complicated, as it needs to repair the ACL server in addition to the spreadsheet servers. Repair is initiated by the administrator invoking a `delete` operation on her request that configured the ACL directory to be world-writable. This initiates local repair on the ACL directory, reverting its permissions to what they were before, and cancels the attacker's request that updated the ACL. This triggers the rest of the repair as in the previous scenario, and fully undoes the attack.

Propagation of corrupt data. Another common pattern of integration between services is synchronization of data, such as notes and documents, between services. If an attack corrupts data on one service, it automatically spreads to the other services that synchronize with it.

To evaluate AIRE’s repair for synchronized services, we reused the spreadsheet application and the setup from the previous scenarios, and added synchronization of a set of cells from spreadsheet service *A* to spreadsheet service *B*. A script on *A* updates the cells on *B* whenever the cells on *A* are modified. As before, the attack is enabled by the administrator mistakenly adding the attacker to the ACL. However, the attacker now corrupts a cell only in service *A*, and the script on *A* automatically propagates the corruption to *B*.

Repair is initiated, as before, with a `delete` operation on the ACL directory. In addition to the repair steps performed in the previous scenario, after service *A* completes its local repair, it invokes a `delete` operation on service *B* to cancel the synchronization script’s update of *B*’s cell. This reverts the updates made by the synchronization, thereby showing that AIRE can track and repair attacks that spread via data synchronization as well.

4.6.2 Partial repair propagation

Repair propagation may be partial if some services are offline during repair or a service rejects a repair message as unauthorized. To evaluate AIRE’s partial repair due to offline services, we re-ran the Askbot repair experiment with Dpaste offline during repair. Local repair runs on both OAuth and Askbot; the vulnerability on OAuth is fixed and the attacker’s post to Askbot is deleted. Clients interacting with OAuth and Askbot see the state with the attacker’s post deleted, which is a valid state, as this could have resulted due to a concurrent operation by another client. Most importantly, this partially repaired state immediately prevents any further attacks using that vulnerability, without having to wait for Dpaste to be online. Once Dpaste comes online, repair propagates to it and deletes the attacker’s post on it as well. When we re-ran the experiment and never brought Dpaste back online, AIRE on Askbot timed out attempting to send the `delete` message to Dpaste, and notified the Askbot administrator, so she can take remedial action.

We also ran the spreadsheet experiments with service *B* offline. In all cases, this results in local repair on service *A*, which repairs the corrupted cells on *A*, and prevents further unauthorized access to *A*. If the attacker can still connect to *B*, he can continue to access it, but once *B* comes online and the directory server or *A* propagate repair to it (depending on the specific scenario), *B* is repaired as well. Similar to the offline scenario in Askbot, clients accessing the services at any time find the services’ state to be valid; all repairs to the services are indistinguishable from concurrent updates.

Finally, we used the spreadsheet experiments to evaluate partial repair due to an authorization failure of a repair message. We use an OAuth-like scheme for spreadsheet services to authenticate each other—when a script in a spreadsheet service communicates with another service, it presents a token supplied by the user who created the script. The spreadsheet services implement an access control policy that allows repair of a past request only if the repair message has a valid token for the same user on whose behalf the request was originally issued.

We ran the spreadsheet experiments and initiated repair after the user tokens for service *B* have expired. This caused service *B* to reject any repair messages, and AIRE effectively treats it as offline; this results in partially repaired state as in the offline experiment described before. On the next login of the user who created the script, the directory service or *A* (depending on the experiment) presents her with the list of pending repair messages. If she refreshes the token for service *B*, AIRE propagates repair to *B*, repairing it as well.

Service	Simple CRUD	Versioned	Conflict check	Description
Amazon S3	✓	✓		Simple file storage
Google Docs	✓	✓		Office applications
Google Drive	✓	✓		File hosting
Dropbox	✓	✓	✓	File hosting
Github	✓	✓		Project hosting
Facebook	✓			Social networking
Twitter	✓			Social microblogging
Flickr	✓			Photo sharing
Salesforce	✓			Web-based CRM
Heroku	✓			Cloud apps platform

Table 4.3: Kinds of interfaces provided by popular web service APIs to their clients.

The results of these three experiments demonstrate that AIRE’s partial repair can repair the subset of services that are online, and propagate repair once offline services or appropriate credentials become available.

4.6.3 Partial repair in real web services

Here we evaluate whether AIRE’s partial repair preserves the invariants that real web service APIs offer their clients, even when some of the services are offline. As real web services like Google Docs are closed-source, we could not inspect their source, and could not run experiments on them to evaluate whether they are compatible with partially repaired state. Instead, we studied the APIs of 10 popular web services to understand the types of guarantees they offer, and implemented one such guarantee—versioning—in our spreadsheet application.

API study. The interfaces exposed by typical web services fall into three categories, as shown in Table 4.3. The simple CRUD (create, read, update, delete) functionality is offered by all services using HTTP PUT, GET, POST, and DELETE operations on the resource objects exported by a service. There is no concurrency control, and if multiple updates happen simultaneously, the last update wins.

Some services provide a versioning API to deal with concurrent updates, which provides a single linear history of immutable versions for each resource. These services’ APIs allow clients to fetch the list of versions of a resource and to restore a resource to a past version (which creates a new version with the contents of the past version). Making these versioned services compatible with partial repair requires supporting branches as in §4.4.2.

Finally, Dropbox offers a conflict check interface in addition to versioning. In versioning, two concurrent writes create two versions, but the later one silently becomes the current version. This default method of resolving conflicts is insufficient for Dropbox’s file synchronization, as the user may see this as her updates being lost; so, Dropbox’s update API allows a client to specify a parent version, and if the parent version does not match, the update is saved as a new conflicted file, and the client is notified.

Based on the API results from Table 4.3, we conclude that most web services are either already compatible with AIRE’s partial repair because they only support simple CRUD functionality, or can be made compatible by extending their versioning API to support branches.

Versioned spreadsheet. To demonstrate extending versioning to support partial repair, we implemented versioning of cells in our spreadsheet application, with support for branching as in Figure 4-3.

With support for branches, conflict check functionality can be implemented as an additional branch, so we do not evaluate it separately, even though it also fits in AIRE's model.

We re-ran the spreadsheet scenarios, with versioning turned on. The difference from the previous run is that the repaired state contained a version tree for each cell that captured all the modifications, including the attacker's corruptions as well as modifications by repair, as immutable versions of the cell. The attacker's corruptions, however, are in a branch separate from the current timeline in the version tree. This preserves the immutability of versions that versioned services guarantee their clients, while still being compatible with partially repaired state resulting from AIRE's asynchronous repair; as before, AIRE was able to recover from the attack. These results demonstrate that real web services can be made compatible with AIRE's partially repaired state so that repair does not violate the guarantees they provide their clients.

4.6.4 Porting applications to use AIRE

Askbot, Dpaste, and Django OAuth are large open source Django applications that are in active use, with 183,000 lines of code in total, excluding comments and blank lines. We did not need to make any changes to run them with AIRE. However, to run the attack scenario in §4.6.1, we needed to modify Askbot to integrate with Django OAuth and Dpaste, which it did not do out-of-the-box. These modifications took 74 and 27 lines of Python code, respectively. We also needed to modify OAuth to add the misconfiguration vulnerability, which took another 13 lines of Python code.

The spreadsheet application without versioning did not need any changes to run with AIRE as well. To evaluate the difficulty of porting the spreadsheet application with versioning, we first implemented a simple linear versioning scheme, where each version is just an incrementing counter. Then we extended it to support version trees and be compatible with AIRE's partial repair. This involved adding parent and timestamp fields to each version, and a pointer to the current version for each cell. This required modifying 44 lines of code in an application with 840 total lines of code. Most versioned services like Dropbox, Amazon S3, and Google Spreadsheets record version creation timestamps and do not use sequential version numbers, so they already maintain enough version information to support AIRE's branched versioning, though implementing branching could involve extra effort.

4.7 Performance evaluation

To evaluate AIRE's performance, this section answers the following questions:

- What is the overhead of AIRE during normal operation, in terms of CPU overhead and disk space?
- How long does repair take on each service, and for the entire system?

We performed experiments on a server with a 2.80 GHz Intel Core i7-860 processor and 8 GB of RAM running Ubuntu 12.10. As our prototype's local repair is currently sequential, we used a single core with hyperthreading turned off to make it easy to reason about overhead.

4.7.1 Overhead during normal operation

To measure AIRE's overhead during normal operation, we ran Askbot with and without AIRE under two workloads: a write-heavy workload that creates new Askbot questions as fast as it can, and a read-heavy workload that repeatedly queries for the list of all the questions. During both workloads, the server experienced 100% CPU load.

Workload	Throughput		Log size per req.	
	No AIRE	AIRE	App.	DB
Reading	21.58 req/s	17.58 req/s	5.52 KB	0.00 KB
Writing	23.26 req/s	16.20 req/s	8.87 KB	0.37 KB

Table 4.4: AIRE overheads for creating questions and reading a list of questions in Askbot. The first numbers are requests per second without and with AIRE. The second numbers show the per-request storage required for AIRE’s logs (compressed) and the database checkpoints.

	Askbot	OAuth	Dpaste
Repaired requests	105 / 2196	2 / 9	1 / 496
Repaired model ops	5444 / 88818	9 / 128	4 / 7937
Repair messages sent	1	1	0
Local repair time	84.06 sec	0.10 sec	3.91 sec
Normal exec time	177.58 sec	0.01 sec	0.02 sec

Table 4.5: AIRE repair performance. The first two rows show the number of repaired requests and model operations out of the total number of requests and model operations, respectively.

Table 4.4 shows the throughput of Askbot in these experiments, and the size of AIRE’s logs. AIRE incurs a CPU overhead of 19% and 30%, and a per-request storage overhead of 5.52 KB and 9.24 KB (or 8 GB and 12 GB per day), respectively. One year’s worth of logs should fit in a 3 TB drive at this worst-case rate, allowing for recovery from attacks during that period.

4.7.2 Repair performance

To evaluate AIRE’s repair performance, we used the Askbot attack scenario from §4.6.1. We constructed a workload with 100 legitimate users and one victim user. The attacker signs up as the victim and performs the attack; during this time, each legitimate user logs in, posts 5 questions, views the list of questions and logs out. Afterwards, we performed repair to recover from the attack.

The results of the experiment are shown in Table 4.5. The two requests repaired in the OAuth service are requests ① and ④ in Figure 4-4, and the one request repaired in Dpaste is request ⑥. The repair messages sent by OAuth and Askbot are the `replace_response` for request ④ and the `delete` for request ⑥, respectively. Askbot does not send `replace_response` for requests ③ and ⑤, as the attacker browser’s requests do not include a `Aire-Notifier-URL:` header.

Local repair on Askbot re-executes 105 out of the 2196 total requests. This is because the attacker posted the question at the beginning of the workload, and subsequent legitimate users’ requests to view the questions page depend on the attacker’s request that posted the question. These requests are re-executed when the attacker’s request is canceled, and their repaired responses do not contain the attacker’s question. AIRE on Askbot does not send `replace_response` messages for these requests as the user’s browsers did not include a `Aire-Notifier-URL:` header, because the current AIRE prototype does not support user’s browsers; AIRE can be integrated into user’s browsers in a manner similar to WARP so that repair can propagate to browsers as well.

Repair takes longest on Askbot, and it is the last to finish local repair. In our unoptimized prototype, repair for each request is $\sim 10\times$ slower than normal execution. This is because the repair controller and the replayed web service are in a separate processes and communicate with each other for every Django model operation; optimizing the communication by co-locating them in the same process should improve repair performance.

Chapter 5

Discussion

5.1 Assumptions

Our design of web application repair makes the following assumptions.

Trusted computing base. Recovery of a web application assumes that the application’s web software stack is in the trusted computing base, as it relies on the repair logs recorded by the software stack during normal operation. The trusted computing base therefore includes the OS, the HTTP server, the database server, the language runtime, and any framework (such as Django or Rails) used by the application; our recovery system cannot recover from an attack that compromises these system components. However, many common web application attacks, such as SQL injection, cross-site scripting, cross-site request forgery, clickjacking, and the scenarios in §4.1.1, do not compromise these system components, and our system can recover from them. Recovery also trusts each user’s browser to correctly record a log of the user’s actions; however, if a user’s browser is compromised, it affects recovery of only that user’s actions and not the actions of other users, as recovery trusts a browser’s log only as much as it trusts the browser’s HTTP requests.

Reducing recovery’s trusted computing base is desirable from a security point of view, and two promising approaches to do so are as follows. First, by borrowing ideas from prior work [45, 58, 63] and tracking dependencies at multiple levels of abstraction, we can use lower layer logs to recover from compromises in system components. For example, OS-level repair logs could be used to repair a compromised database server or language runtime. Second, by inserting proxies in front of the HTTP server and the database, and performing logging in the proxies instead of in the application’s web software stack, we can remove the need to rely on the web software stack for logging. However, this may also lead to more re-execution during repair, as the captured logs are not exact (e.g., the exact database queries made for each HTTP request are not captured).

Correct and minimal patches. We assume that the patch used for patch-based auditing and retroactive patching correctly fixes the vulnerability being repaired. Recovery works best with patches that do not change program behavior aside from fixing the vulnerability. Patches that both fix security bugs and introduce new features, or that significantly modify the application in order to fix a vulnerability, could generate false positives during auditing, and could result in unnecessary re-execution or raise too many conflicts during repair. For example, if the patch upgraded commonly used functionality in the application that was used by legitimate requests, auditing flags those legitimate requests as suspects, and repair needlessly re-executes them.

Applications are recovery-enabled. Automatic repair propagation across applications assumes that all the applications to which repair needs to be propagated are running our recovery system and have recovery enabled on them. If an affected application is not running our recovery system, repair cannot be automatically propagated to it and recovery loses track of the spread of the attack beyond that point. If repair propagation from an application to a remote application fails, the application's administrator is notified of this fact, along with the repair that cannot be propagated, so that she can take remedial action (perhaps notify the administrator of the remote application, so she can initiate manual recovery).

Application support for repair. Recovery makes three assumptions about each recovery-enabled application. First, that the application defines an appropriate UI replayer that correctly replays user input in a repaired page and correctly flags conflicts. Second, that the application defines an appropriate access control policy that denies access to unauthorized clients requesting repair. Finally, that the application and its clients support partially repaired state. If the first assumption does not hold, recovery may not fully revert an attack's effects; if the second assumption does not hold, attackers would be able to use repair to make unauthorized changes to a service; and, if the third assumption is broken, a service's clients may observe partially repaired state as being corrupt. Though WARP's default UI replayer and AIRE's default access control policy work for many applications, we rely on an application's developer to override them with suitable alternatives if they do not work for that application. Similarly, though most existing web applications support partially repaired state with no modifications, we rely on an application's developer to suitably modify the application if it does not support partially repaired state.

User support for repair. Recovery makes two assumptions about the administrator (or user) who initiates repair. First, recovery assumes that the administrator resolves any conflicts that are flagged during repair. Second, recovery assumes that the administrator takes remedial measures (e.g., by applying a compensating action) for any affected network communications that it cannot automatically repair (perhaps, because the remote system is not recovery-enabled). If the first assumption is broken and the administrator does not resolve a conflict, legitimate effects of the command that caused the conflict would be missing from the repaired state. If the second assumption is broken, repair may not propagate to all the affected machines, and some machines may still have state corrupted by the attack.

5.2 Limitations

This section discusses limitations of our current prototypes along with approaches to address them.

Repairing data leaks. Recovery cannot undo disclosures of private data, such as if an adversary steals sensitive information from the application, or steals a user's password. However, when private data is leaked, recovery can still help track down affected users; if the administrator specifies which data is private, recovery can notify her of reads that returned the private data only during original execution but not during repair, and hence are potential attacks that leaked the data.

In the case of stolen credentials, an attacker can use them to impersonate the user and perform unauthorized actions; recovery cannot distinguish the attacker's actions from legitimate user's actions, as both used the same credentials. However, if the user is willing to identify her legitimate browsers or the administrator can identify the attacker's IP address, recovery can undo the attacker's actions.

Logging sensitive data. The client-side logs, uploaded by WARP's browser 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 the client-side 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 our system, applications might not record or store this information on their own, and this additionally stored information must be safeguarded from unintended disclosure. Similarly, server-side logs can also contain sensitive information in the logged HTTP requests.

One way to avoid logging known-sensitive data, such as passwords, is to modify 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, and recovery can require the administrator to provide the corresponding decryption key to initiate repair.

Garbage collection of repair logs. Repair logs and database versions grow in size over time. It is up to the administrator of the applications to decide how long to store the logs. For a small application such as HotCRP, it may make sense to store all logs from the time when the conference starts, so that the system can recover from any past attack. For a larger-scale web site, such as Wikipedia, it may make sense to discard old logs and old database versions at some point (e.g., after several months), although recovery would not be possible for attack requests whose logs were discarded.

Application source code updates. Our current WARP 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).

Similarly, patch-based auditing in our current POIROT prototype assumes the application source code is static, but in practice, application source code is upgraded over time. In order to audit past requests that were executed on different versions of the software, the patch being audited must be back-ported to each of those software versions; this is already common practice for large software projects such as MediaWiki. From POIROT's point of view, the indexes generated for each version of the software must be kept separate, and POIROT's control flow filter must separately analyze the basic blocks for each version. Finally, re-execution of a request must use the source code originally used to run that request (plus the backported patch for that version).

Need for a browser extension. To recover from intrusions that involve a user's browser, our WARP prototype requires the user to install a browser extension that records client-side events and user input, and uploads them to WARP-enabled servers. If a 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 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 her data from that point onward, and cancel her previous HTTP requests, if unwanted changes are detected. Furthermore, we believe it would be possible to implement the extension's recording functionality in pure JavaScript as well, perhaps by leveraging Caja [57] 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.

Distributed repair involving browsers Our current WARP prototype cannot repair mashup web applications that communicate with each other in a user's browser, since the browser event logs for each application would be uploaded to that application's server. WARP can be extended to repair such multi-origin web applications in two ways. The first approach is to extend the WARP browser extension to support AIRE's distributed repair protocol: users' browsers become participants in distributed repair and propagate repair across the web applications in a mashup. However, in this approach, repair propagation has to wait for each affected user's browser to come online, which can slowdown progress of repair. The second approach is to have the browser 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 the repair controller at the source origin's server to convince the repair controller on the other frame's origin server that it should be allowed to initiate re-execution for that user, and does not require involvement of users' browsers during repair.

Chapter 6

Related work

There has been a significant amount of past research related to the work in this thesis, in areas such as intrusion detection, recovery, taint tracking, databases, and debugging. This chapter places our contributions in the context of prior research.

Intrusion detection. Past intrusion recovery systems explored several approaches to identify initial intrusions. One approach adopted by systems like Retro [45], Taser [29], Back to the Future [37], ReVirt [24] and Operator Undo [15], is to rely on the administrator to identify intrusions. The administrator could manually inspect logs or use intrusion detection tools [36, 49, 51, 75] to look for symptoms of an attack, and use a tool like BackTracker [46] to track down the attack’s initial entry point. However, this approach requires significant manual effort on the administrator’s part and is error-prone. Another approach is to rely on developers to specify vulnerability-specific predicates [40] for *each* discovered vulnerability. Each predicate is run against past system executions to determine attacks that exploited the vulnerability checked by the predicate. However, this approach imposes significant extra effort for developers.

Unlike prior systems, POIROT’s patch-based auditing and WARP’s retroactive patching use just the patch fixing a vulnerability to identify and recover from any attacks that exploited the vulnerability, and therefore do not place additional burden on administrators or developers.

POIROT’s approach to auditing compares the execution of past requests on patched and original code, which is similar to Rad [74]. POIROT’s contributions over Rad lie in its techniques to improve the performance of this approach for web applications, namely, control flow filtering, function-level auditing, and memoized re-execution.

Recovery. The two recovery systems closest to WARP and AIRE are Retro [45] and Dare [44], respectively. Though WARP borrows the action history graph concept from Retro, Retro focuses on shell-oriented Unix applications on a single machine and cannot be directly applied to web application recovery, for the following three reasons. First, Retro requires the administrator to detect attacks, unlike WARP, which just needs a security patch to perform retroactive patching. 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 database table, which WARP’s time-travel database can. Finally, 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 automated UI replay eliminates the need to resolve most conflicts, and presents a meaningful UI for the true conflicts that require user input.

Dare extends Retro to perform intrusion recovery on a cluster of machines. However, Dare’s distributed repair is synchronous and it assumes that all machines are under the same administrative

domain; both these design decisions make it not applicable to distributed web services. AIRE’s asynchronous repair, in contrast, is designed for loosely-coupled distributed services, and AIRE’s repair protocol supports repair of services across administrative domains.

Operator Undo [15] recovers from an operator mistake in an email server by rolling back the server state, repairing the mistake in the past, and replaying a recorded log of operations. This is similar to our recovery approach. However, Operator Undo is limited to recovering from accidental mistakes and, unlike our work, it cannot recover from attacks. Furthermore, in contrast to retroactive patching, it relies on the administrator to repair the past mistake; also, it replays all recorded operations after the mistake whereas we use dependency tracking to replay only requests affected by an attack.

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. Our work, on the other hand, can fully recover from data corruptions due to bugs as well as attacks, with no manual intervention (except when there are conflicts during repair). Our evaluation of WARP (§2.7.4) compared WARP to Akkuş and Goel’s system in more detail.

Polygraph [52] recovers from compromises in a weakly consistent replication system by rolling back corrupted state. Unlike our work, Polygraph does not attempt to preserve legitimate changes to affected files, which can lead to significant data loss. Furthermore, it does not automate detection of compromises. Polygraph works well for applications that do not operate on multiple files at once. In contrast, our work deals with web applications, which frequently access data in a single shared database.

Simmonds et al.’s user-guided recovery system [67] recovers from violations of application-level invariants in a web service and uses compensating actions and user input to resolve these violations. However, it does not recover from attacks or accidental data corruptions.

Provenance and taint tracking Provenance-aware storage systems [59, 60] 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 [53] 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 [63] and PASSv2 [58] 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), ideas from these systems could be used to extend WARP so that it can recover from intrusions that span many layers (e.g., the database server or the language runtime).

POIROT’s dependency analysis is similar to taint tracking systems [25, 62]. A key distinction is that taint tracking systems are prone to “taint explosion” if taint is propagated on all possible information flow paths, including through control flow. As a result, taint tracking systems often trade off precision for fewer false positives (i.e., needlessly tainted objects). POIROT addresses the problem of taint explosion through control flow by *fixing* the control flow path for a group of requests, thereby avoiding the need to consider control flow dependencies.

Databases. Tracking down and reverting malicious actions has been explored in the context of databases [13, 50]. 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 [64, 68]. However, unlike a temporal database, WARP has no need for more complex temporal queries, has support for 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 [20], and in the context of index selection [27, 39]. These techniques might be helpful in choosing a partitioning for database tables in WARP.

Testing and debugging. Mugshot [56] uses record and replay for debugging web applications, whereas our work uses it for intrusion recovery. Mugshot’s recording and replay of JavaScript events is deterministic, and it cannot replay events on a changed web page. WARP, on the other hand, 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.

POIROT’s approach to auditing a system for intrusions is based on comparing the execution of past requests using two versions of the code: one with a patch applied, and one without. This is similar to the approach used by delta execution [69] and TACHYON [54] for patch validation. POIROT’s memoized re-execution is similar to dynamic slicing [8], which computes the set of instructions that indirectly affected a given variable. Program slicing, and dynamic slicing in particular, was proposed in the context of helping developers debug a single program. POIROT shows that similar techniques can be applied to locate and memoize identical computations across multiple invocations of a program.

POIROT’s control flow filtering is similar to the problem of regression test selection [9, 14]: given a set of regression tests and a modification to the program, identifying the regression tests that need to be re-run to test the modified program. POIROT demonstrates that control flow filtering works well for patch-based auditing under a realistic workload, and further introduces additional techniques (function-level auditing and memoized re-execution) which significantly speed up the re-execution of requests beyond static control flow filtering.

Khalek et al. [42] show that eliminating common setup phases of unit tests in Java can speed up test execution, similar to POIROT’s function-level auditing. However, Khalek et al. require the programmer to define undo methods for all operations in a unit test, which places a significant burden on the programmer that POIROT avoids.

Past work on distributed debugging [10] intercepted executions of unmodified applications and tracked causal dependencies for performance debugging, whereas AIRE tracks dependencies to recover from attacks and mistakes.

Other related work. Dynamic dataflow analysis [19] and symbolic execution [16] have been used to generate constraints on a program’s input that elicit a particular program execution. These techniques are complementary to control flow filtering and could be extended to apply to POIROT’s auditing.

Memoization has been used to speed up re-execution of an application over slightly different inputs [7, 35, 73]. Though POIROT’s techniques can be extended to work for that scenario as well, memoized re-execution in the current POIROT design detects identical computations *across* different executions of a program, and separates memoized computations from input-dependent computations, by grouping requests according to their control flow traces.

Heat-ray [23] considers the problem of attackers propagating between machines within a single administrative domain, and suggests ways to reduce trust between machines. On the other hand, AIRE is focused on attackers spreading across web services that do not have a single administrator, and allows

recovery from intrusions. Techniques such as Heat-ray could be helpful in understanding and limiting the ability of an adversary to spread from one service to another.

Chapter 7

Conclusion

This dissertation demonstrated that automated recovery in web applications is practical as well as effective. Our work developed several key ideas for this purpose: retroactive patching, automated UI replay, dependency tracking, patch-based auditing, and distributed repair. With these ideas, we can audit applications for past intrusions and recover them from intrusions and user mistakes, with little to no effort on the part of system administrators. Automated auditing and recovery are important tools in an administrator’s security toolbox, complementary to intrusion prevention, because, despite the best efforts of administrators, every system will eventually have a compromise. Though this thesis focused on automated recovery in web applications, our ideas are more generally applicable, and can be adapted to other systems such as mobile applications and desktop applications.

We built three systems, WARP, POIROT, and AIRE, to explore and evaluate our ideas. The WARP intrusion recovery system introduced three of the key ideas to make recovery practical in a single web application. Retroactive patching allows administrators to recover from past intrusions that exploited a vulnerability by simply supplying a security patch fixing the vulnerability, without having to identify intrusions or even knowing if an attack occurred. The time-travel database allows WARP to perform precise repair of just the affected parts of the system, by tracking the spread of the attack through the database and rolling back just the affected parts of the database during repair. Finally, DOM-level replay of user input allows WARP to replay legitimate users’ input and preserve legitimate changes, with no administrator or user involvement in many cases. Our evaluation of a WARP prototype shows that it can recover from attacks, misconfigurations, and data loss bugs in real applications, without requiring any code changes, and with modest runtime overhead.

POIROT’s patch-based auditing identifies past requests in a web application that potentially exploited a patched security vulnerability. POIROT’s auditing improves WARP’s repair performance, as it is more precise than WARP’s patch-based auditing and returns fewer false positives. POIROT’s auditing is efficient as well, due to three key techniques that POIROT introduced: control-flow filtering, function-level auditing, and memoized re-execution. We evaluated a prototype of POIROT and showed that it is effective at detecting exploits of real vulnerabilities in MediaWiki and HotCRP; that it has false positives on only 1 out of 34 recent MediaWiki patches as compared to WARP’s false positives on 17 out of the same 34 patches; and that its optimizations allow it to audit challenging patches, which affect every request, 12–51× faster than the original execution time of those requests.

AIRE extended WARP’s recovery to repair attacks that spread across multiple web applications. AIRE introduced three key techniques that make distributed repair practical. First, an asynchronous repair protocol to propagate repair across services that span administrative domains, even when not all the services are available. Second, an access control mechanism that delegates access checks to the application to ensure that the repair protocol does not give attackers new ways to subvert web

applications. Finally, a repair model to reason about partially repaired states resulting from asynchronous repair. We demonstrated, using a prototype of AIRE, that porting existing applications to AIRE requires little effort, that AIRE can recover from realistic distributed attack scenarios, and that typical web service APIs can support partially repaired states resulting from AIRE's asynchronous repair.

Bibliography

- [1] Askbot – create your Q&A forum. <http://www.askbot.com>.
- [2] Django: the Web framework for perfectionists with deadlines. <http://www.djangoproject.com>.
- [3] OAuth security advisory: 2009.1. <http://oauth.net/advisories/2009-1/>, April 2009.
- [4] W32.stuxnet. http://www.symantec.com/security_response/writeup.jsp?docid=2010-071400-3123-99, July 2010.
- [5] Wikimedia labs database dump. http://dumps.wikimedia.org/en_labswikimedia/20111228/, December 2011.
- [6] Twitter OAuth API keys leaked. http://threatpost.com/en_us/blogs/twitter-oauth-api-keys-leaked-030713, March 2013.
- [7] Umut Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 2008.
- [8] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.
- [9] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul A. London. Incremental regression testing. In *Proceedings of the IEEE Conference on Software Maintenance*, September 1993.
- [10] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.
- [11] İ. E. Akkuş and Ashvin Goel. Data recovery for web applications. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Chicago, IL, June–July 2010.
- [12] Frances E. Allen. Control flow analysis. In *Proceedings of the Symposium on Compiler Optimization*, 1970.
- [13] Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from malicious transactions. *Transactions on Knowledge and Data Engineering*, 14:1167–1185, 2002.
- [14] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. Regression test selection techniques: A survey. *Informatica*, 35(3):289–321, October 2011.

- [15] Aaron B. Brown and David A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 1–14, San Antonio, TX, June 2003.
- [16] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.
- [17] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 101–114, Cascais, Portugal, October 2011.
- [18] Ramesh Chandra, Taesoo Kim, and Nickolai Zeldovich. Asynchronous intrusion recovery for distributed web services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013. To appear.
- [19] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, October 2005.
- [20] 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.
- [21] Damon Cortesi. Twitter StalkDaily worm postmortem. <http://dcortesi.com/2009/04/11/twitter-stalkdaily-worm-postmortem/>.
- [22] David Drummond. A new approach to China. <http://googleblog.blogspot.com/2010/01/new-approach-to-china.html>, January 2010.
- [23] John Dunagan, Alice X. Zheng, and Daniel R. Simon. Heat-ray: Combating identity snowball attacks using machine learning, combinatorial optimization and attack graphs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.
- [24] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, Boston, MA, December 2002.
- [25] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.
- [26] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.
- [27] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91–128, 1988.
- [28] Github. SSH key audit. <https://github.com/settings/ssh/audit>, 2012.
- [29] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. The Taser intrusion recovery system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 163–176, Brighton, UK, October 2005.

- [30] Nir Goldshlager. How I hacked any Facebook account...again! <http://www.nirgoldshlager.com/2013/03/how-i-hacked-any-facebook-accountagain.html>, March 2013.
- [31] Nir Goldshlager. How I hacked Facebook OAuth to get full permission on any Facebook account. <http://www.nirgoldshlager.com/2013/02/how-i-hacked-facebook-oauth-to-get-full.html>, February 2013.
- [32] Nir Goldshlager. How i hacked instagram accounts. <http://www.breaksec.com/?p=6164>, May 2013.
- [33] Dan Goodin. Surfing Google may be harmful to your security. *The Register*, August 2008.
- [34] Google, Inc. Google apps script. <https://script.google.com>, 2013.
- [35] Philip J. Guo and Dawson Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, July 2011.
- [36] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.
- [37] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. In *22nd Annual Computer Security Applications Conference (ACSAC)*, pages 257–268, December 2006.
- [38] ifttt, Inc. Put the internet to work for you. <https://ifttt.com>, 2013.
- [39] 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.
- [40] Ashlesha Joshi, Sam King, George Dunlap, and Peter Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 91–104, Brighton, UK, October 2005.
- [41] Nishant Kaushik. Protecting yourself while using cloud services. <http://blog.talkingidentity.com/2011/11/protecting-yourself-while-using-cloud-services.html>, November 2011.
- [42] Shadi Abdul Khalek and Sarfraz Khurshid. Efficiently running test suites using abstract undo operations. *IEEE International Symposium on Software Reliability Engineering*, pages 110–119, 2011.
- [43] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 193–206, Hollywood, CA, October 2012.
- [44] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Recovering from intrusions in distributed systems with Dare. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems*, Seoul, South Korea, July 2012.
- [45] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 89–104, Vancouver, Canada, October 2010.

- [46] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, February 2005.
- [47] Eddie Kohler. Hot crap! In *Proceedings of the Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems*, San Francisco, CA, April 2008.
- [48] Eddie Kohler. Correct humiliating information flow exposure of comments. <http://www.read.cs.ucla.edu/gitweb?p=hotcrp;a=commit;h=f30eb4e52e91ab230944eebe8f31bf61e9783d3a>, March 2012.
- [49] Wenke Lee, Salvatore Stolfo, and Patrick Chan. Learning patterns from Unix process execution traces for intrusion detection. In *Proceedings of the AAAI Workshop on AI Methods in Fraud and Risk Management*, pages 50–56, July 1997.
- [50] Peng Liu, Paul Ammann, and Sushil Jajodia. Rewriting histories: Recovering from malicious transactions. *Journal of Distributed and Parallel Databases*, 8:7–40, 2000.
- [51] Benjamin Livshits and Weidong Cui. Spectator: Detection and containment of JavaScript worms. In *Proceedings of the 2008 USENIX Annual Technical Conference*, Boston, MA, June 2008.
- [52] Prince Mahajan, Ramakrishna Kotla, Catherine C. Marshall, Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, and Ted Wobber. Effective and efficient compromise recovery for weakly consistent replication. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, March 2009.
- [53] Daniel W. Margo and Margo Seltzer. The case for browser provenance. In *Proceedings of the 1st Workshop on the Theory and Practice of Provenance*, San Francisco, CA, February 2009.
- [54] Matthew Maurer and David Brumley. TACHYON: Tandem execution for efficient live patch testing. In *Proceedings of the 21st Usenix Security Symposium*, Bellevue, WA, August 2012.
- [55] MediaWiki. MediaWiki. <http://www.mediawiki.org>, 2012.
- [56] James Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for JavaScript applications. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, April 2010.
- [57] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript, 2008. <http://code.google.com/p/google-caja/downloads/list>.
- [58] Kira-Kumar Muniswamy-Reddy, U. Braun, D. Holland, P. Macko, D. Maclean, Daniel W. Margo, Margo Seltzer, and R. Smogor. Layering in provenance systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, June 2009.
- [59] Kira-Kumar 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–June 2006.
- [60] Kira-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proceedings of the 8th Conference on File and Storage Technologies*, San Jose, CA, February 2010.
- [61] National Vulnerability Database. CVE statistics. <http://web.nvd.nist.gov/view/vuln/statistics>, May 2013.

- [62] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.
- [63] Christopher Olston and Anish Das Sarma. Ibis: A provenance manager for multi-layer systems. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, Pacific Grove, CA, January 2011.
- [64] Oracle Corporation. Oracle flashback technology. <http://www.oracle.com/technetwork/database/features/availability/flashback-overview-082751.html>.
- [65] Tom Preston-Werner. Public key security vulnerability and mitigation. <https://github.com/blog/1068>, March 2012.
- [66] Derick Rethans. Vulcan logic dumper. <http://derickrethans.nl/vld.php>, 2009.
- [67] Jocelyn Simmonds, Shoham Ben-David, and Marsha Chechik. Guided recovery for web service applications. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010.
- [68] Richard T. Snodgrass and Ilsoo Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.
- [69] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. Efficient online validation with delta execution. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, March 2009.
- [70] Twitter, Inc. My account has been compromised. <https://support.twitter.com/articles/31796-my-account-has-been-compromised>, 2013.
- [71] Joey Tyson. Recent Facebook XSS attacks show increasing sophistication. <http://theharmonyguy.com/2011/04/21/recent-facebook-xss-attacks/>, April 2011.
- [72] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.
- [73] Amin Vahdat and Thomas Anderson. Transparent result caching.
- [74] Xi Wang, Nikolai Zeldovich, and M. Frans Kaashoek. Retroactive auditing. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011.
- [75] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 20th IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.
- [76] Karen Wickre. About that fake post. <http://googleblog.blogspot.com/2006/10/about-that-fake-post.html>.
- [77] Yahoo, Inc. Pipes: Rewire the web. <http://pipes.yahoo.com>, 2013.
- [78] Zapier, Inc. Automate the web. <https://zapier.com>, 2013.