Efficient patch-based auditing for web application vulnerabilities

Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich
MIT CSAIL

Abstract

POIROT is a system that, given a patch for a newly discovered security vulnerability in a web application, helps administrators detect past intrusions that exploited the vulnerability. POIROT records all requests to the server during normal operation, and given a patch, re-executes requests using both patched and unpatched software, and reports to the administrator any request that executes differently in the two cases. A key challenge with this approach is the cost of re-executing all requests, and POIROT introduces several techniques to reduce the time required to audit past requests, including filtering requests based on their control flow and memoization of intermediate results across different requests.

A prototype of POIROT for PHP accurately detects attacks on older versions of MediaWiki and HotCRP, given subsequently released patches. POIROT's techniques allow it to audit past requests 12–51× faster than the time it took to originally execute the same requests, for patches to code executed by every request, under a realistic MediaWiki workload.

1 Introduction

New security vulnerabilities are routinely discovered in many web applications, and web application developers frequently release patches for such bugs in their software. Once an administrator learns about a new vulnerability and applies a patch to prevent new attacks, the administrator may want to check whether anyone exploited the bug before the patch was applied, in order to take any necessary remedial measures. This check is important to ensure that no data was leaked or that the attacker did not leave any back doors for later intrusions, yet today this check remains a mostly manual process.

As one example, consider the HotCRP conference management software [21], which recently had a information disclosure bug that allowed paper authors to view a reviewer's private comments meant only for program committee members [22]. 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, 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 [27], 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 users to re-confirm their keys [13]—an intrusive measure, yet one that was necessary because of the lack of alternatives.

This paper presents POIROT, a system that can audit a web application's past requests and identify requests that potentially exploited a vulnerability, given a patch that fixes the vulnerability. POIROT focuses on vulnerabilities in server-side application code, which includes seven of the top ten web application vulnerabilities [29].

POIROT adopts the record-and-replay approach from previous systems [9, 33], and records each request to the web application during the application's normal execution. When a patch is released, the administrator invokes POIROT, which re-runs past requests on two versions of the application source code—one with and one without the patch—and compares 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 to the administrator as a possible attack, along with a diff of the results with and without the patch.

POIROT's key contribution lies in performance. The closest related work is Warp [9], which undoes the effects of past attacks given a patch by re-executing every request that touched a patched file. In the worst case, a developer may patch code that is executed by every request, in which case auditing several months worth of requests with Warp would take yet another several months on production servers. POIROT shows that it is possible to audit 1–2 orders of magnitude faster than simply re-running every request, even for the challenging patches that modify code executed by *every* request. POIROT's design speeds up auditing by leveraging three techniques, as follows.

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 don't 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 [3], and is often 1–2 orders of magnitude faster than re-executing the entire request.

An evaluation of a POIROT prototype for PHP-based applications with MediaWiki and HotCRP shows that POIROT can accurately and efficiently detect past intrusions given a patch, and that the three techniques mentioned above are important to achieve good performance. Out of 34 real MediaWiki security patches, POIROT takes 1,013 seconds to audit a patch in the worst case (when the patch affects all requests) for a workload that takes 12,116 seconds to complete during normal execution. For a realistic workload based on Wikipedia traces, POIROT imposes 15% CPU overhead during normal execution and requires 5 KB of storage per request, which amounts to 3.3 GB per day for one server. Finally, POIROT has no false negatives, and incurs no false positives for most patches.

The rest of this paper is organized as follows. §2 starts off with an overview of POIROT's design and its workflow. §3, §4, and §5 describe POIROT's three key techniques for minimizing re-execution. §6 discusses our prototype implementation, and §7 evaluates it. §8 touches on some of the limitations of POIROT. §9 compares POIROT with related work, and §10 concludes.

2 Overview

To understand how POIROT helps an administrator automate the auditing process, 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, leaving the administrator to decide on the appropriate remedy. 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 mentioned earlier, 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 report the discrepancy to the administrator.

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, to be further audited by the administrator.

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

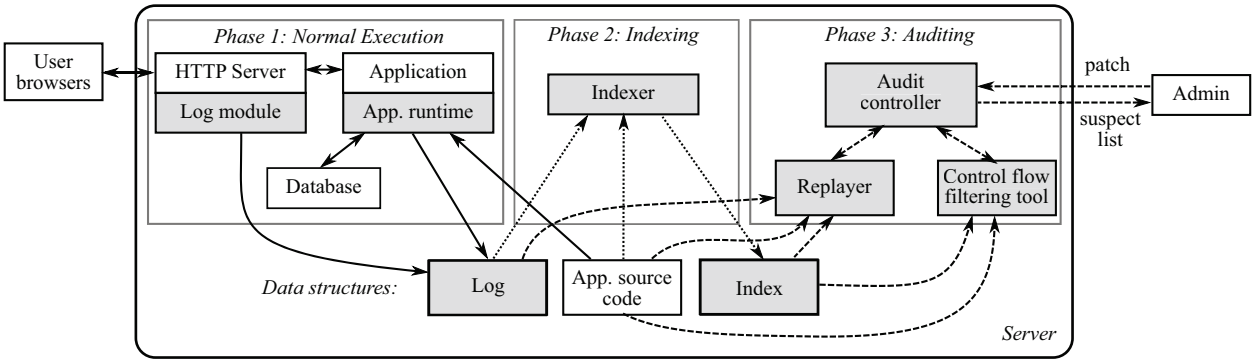

Figure 1: Overview of POIROT's design. Components introduced by POIROT are shaded. Solid lines, dotted lines, and dashed lines indicate interactions during normal execution, indexing, and auditing stages, respectively.

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

2.1 Logging during normal execution

In order to rerun a past request in a web application, 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 [5].

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.

It is up to the administrator to decide on how long to store POIROT's logs. For an application such as HotCRP, it may make sense to store all logs from the time when the conference starts, and audit every request if a vulnerability is discovered. For a larger-scale web site, such as Wikipedia, it may make sense to discard logs of old requests at some point (e.g., after several months), although POIROT would be unable to audit discarded requests for possible attacks.

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.

2.3 Auditing

When a patch for a newly discovered security vulnerability is released, an administrator can invoke POIROT's auditing phase and supply the patch to POIROT. 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 provides the administrator with a list of suspect requests that executed differently with the patched code than they did with the unpatched code, for further examination.

2.4 Limitations and assumptions

POIROT is designed to detect attacks that exploit bugs in a web application's code. Consequently, POIROT assumes that adversaries do not subvert the language interpreter, the web server, or the OS kernel. An adversary that violates this assumption would be able to alter POIROT's logs to hide the attack.

POIROT assumes that the vulnerability being audited is correctly fixed by the security patch used for auditing. Under this assumption, POIROT incurs no false negatives. However, POIROT can incur false positives because it treats any change in application output as an indication of a possible attack. For example, if POIROT failed to record some non-determinism during a request's original execution, re-executing the request could change the request's output and cause POIROT to flag it, even if the request did not exploit the patched vulnerability.

POIROT works best with patches that do not change program behavior aside from fixing a security 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. For example, if a patch issues a new database query, POIROT flags every request executing the patched code as a possible attack. Extending POIROT to snapshot the database state during original execution, and restore it during request re-execution (as in Warp [9]) would likely prevent these false positives.

POIROT's design focuses on high performance auditing. Once POIROT flags a request as suspicious, an administrator familiar with the application must manually inspect that request to determine the appropriate course of action. POIROT can be combined with a system like Warp [9] to undo the effects of suspicious requests. The problem of helping administrators understand the impact of a suspicious request is left to future work.

POIROT's prototype is built for PHP; PHP's single-threaded nature and its higher-level primitives (e.g., string operations) simplified the prototype's implementation. We believe it is straightforward to extend POIROT to other scripting languages such as Python and Ruby; however, extending POIROT to low-level bytecode such as x86 poses some challenges, which we discuss in §8.

POIROT's prototype assumes the application is never upgraded. We discuss this limitation further in §8.

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 (§5).

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

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 $\langle filename, classname, funcname \rangle$.

3.2 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 Determining the patched basic blocks

Once the administrator provides a patch to POIROT in the auditing phase, POIROT must determine the set of requests to re-execute. 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.

3.4 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.

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

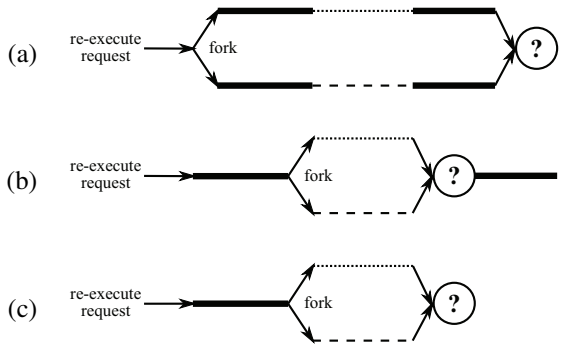

Figure 2: Three refinements of request re-execution: (a) naïve, (b) function-level auditing, and (c) early termination. Thick lines indicate execution of unmodified application code, dotted lines indicate execution of the original code for patched functions, and dashed lines indicate execution of new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

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 the administrator wants 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.

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 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 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 §4.1. If the results and side-effects are 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.

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.

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 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.

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, and suppose the web server received three requests, shown in Figure 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 flow traces, even if their inputs differed, such as the three requests shown in Figure 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, 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 (§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.

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 ev-

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

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 5: PHP bytecode instructions for lines 5–12 in Figure 3. The line column refers to source lines from Figure 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 4 when auditing the patch in Figure 3.

ery 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 [25], 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, 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 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 §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 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 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 §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 5, template generation stops after instruction 16, because there are no subsequent calls to the patched `name()` function.

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 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 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.

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 5) with the values from that request, and invokes the template bytecode. In the example of Figure 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 5), POIROT performs function-level auditing, as described in §4, to audit the execution of this function for this particular request. Once the function returns, POIROT compares the results of the 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.

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 §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.

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 1: Lines of code for components of the POIROT prototype.

6 Implementation

We implemented a prototype of POIROT for PHP. Table 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 [28] 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.

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.

7 Evaluation

Our evaluation aims to support the following hypotheses:

- POIROT incurs low runtime overhead (§7.2).
- POIROT detects exploits of real vulnerabilities with few false positives (§7.3).
- Even for challenging patches that affect every request, POIROT can audit much faster than either naïve re-execution or the closest related system, Warp (§7.4).
- POIROT's techniques are important for performance (§7.5).

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 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.

Using a realistic MediaWiki workload and a synthetic HotCRP workload, we show that POIROT's auditing performance is 24–133× that of naïve re-execution, and an additional factor of $\sim 5\times$ faster than Warp (due to Warp's overheads compared to naïve). 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.

7.1 Experimental setup

The test applications used for these experiments were MediaWiki [24], a popular Wiki application that also runs the Wikipedia site, 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 [31]. 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 [1] 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.

7.2 Normal execution overheads

To illustrate POIROT's overhead during normal execution, we used several workloads; the results are shown in Table 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 [31] would be even smaller, since it has an order of magnitude more common requests than our 100k workload.

Table 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.

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 compare POIROT to Warp, the closest related work, and we used the same five vulnerabilities evaluated by Warp's authors. The real Wikipedia trace [31] 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 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

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: 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	ε

⋆ 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 4: POIROT's auditing performance with 34 patches for MediaWiki vulnerabilities, compared with the performance of the naïve re-execution scheme and Warp's estimated performance for the same patches (estimated to be 10× the original execution time, based on results from [9]). ε for Warp indicates the cost of accessing its index, which was not reported in the Warp paper. Naïve results are measured only for the top 5 patches; its performance would be similar for the 29 other patches.

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 to include the browser (as in Warp) would likely prevent these false positives. Additionally, POIROT incurs no false positives for 29 other patches shown in Table 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 §1, and interspersed attack requests among a synthetic 200-user workload consisting of user creation, user login, paper submissions, etc. Table 5 shows the results. POIROT is able to detect all four attacks with no false positives.

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 time taken by Warp, based on numbers reported by Chandra et al. [9].

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

Table 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). "D?" indicates whether POIROT detects the attack, and "F+" counts false positives.

Table 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.

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 4). This means that POIROT can audit 3.4 hours worth of requests in ∼17 minutes in the worst case.

Our estimate of Warp's performance, based on that paper, is shown in the rightmost columns of Table 4. Warp's file-level filtering allows it to statically discard some requests, although it is unable to filter out requests for 12 patches that POIROT's basic-block-level filtering can. Moreover, when Warp re-executes requests, it is an order of magnitude slower than normal execution, which is a total of 2–3 orders of magnitude slower than POIROT for the worst case patches; for our 3.4 hour workload, Warp could take 1.4 days to audit all of the requests for one patch.

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 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 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

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 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 (§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 (§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.

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 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.

8 Discussion

Our prototype currently 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).

Although our prototype targets PHP web applications, POIROT's techniques should be equally applicable to web applications in other scripting languages such as Python and Ruby. However, when used with low-level bytecode, such as x86 server programs, POIROT's techniques may be less effective due to the following reasons. First, for x86 server applications such as Apache, recording all basic blocks for control flow filtering can impose ∼60% overhead during normal execution [26]; it may be possible to reduce this overhead by profiling the application and recording branches only along the uncommon paths. Second, x86 applications can be multi-threaded, and the non-determinism of thread interleaving can reduce the effectiveness of generating templates for memoized re-execution. Since servers such as Apache typically execute each request in a single thread, independent of other requests, it may be possible to record the execution of each request's thread as a separate control flow trace and use that for memoization. Finally, memoized re-execution in x86 applications may be less effective at finding many requests that share the exact same control flow; for example, string operations in assembly often iterate over all characters in a string, whereas the same operations appear as a single opcode in PHP, Python, and Ruby. One way to apply memoized re-execution at a low level would be to treat string operations as primitives.

Our prototype is currently single-threaded and it was evaluated on a single-core machine. However, the design of the POIROT replayer has lots of parallelism, and it is straightforward to extend it to re-execute requests in parallel. This can be used to significantly reduce auditing time, perhaps taking advantage of cloud computing platforms such as Amazon EC2.

9 Related work

This paper's key contribution over prior work lies in the techniques for achieving high auditing performance, particularly in efficiently re-executing many requests to audit them for exploits of a security vulnerability. The rest of this section explains the relation between POIROT and prior work in more detail.

POIROT's approach to auditing a system for intrusions is based on comparing the execution of past requests us-

ing two versions of the code: one with a patch applied, and one without. This approach is similar to delta execution [30], Rad [33], Warp's retroactive patching [9], and TACHYON [23]. POIROT's contributions lie in techniques to improve the performance of this approach for web applications: control flow filtering, function-level auditing, early termination, and memoized re-execution. Function-level auditing in particular is similar to delta execution and Rad.

Past intrusion recovery systems explored several approaches to identify initial intrusions. Some relied on the user for identification [7, 11, 14, 16, 19, 20], which is both tedious for the user and is error-prone. Others asked developers to specify vulnerability-specific predicates [17] for *each* discovered vulnerability; this imposes significant extra effort for developers. Finally, Warp [9] and Rad [33] used the actual patch fixing a vulnerability to identify intrusions, relieving the users and developers of the burden of intrusion detection. Similar to Warp and Rad, POIROT also uses the patch to identify intrusions.

Warp's retroactive patching [9] used file-level dependency tracking to determine requests that were affected by a patch and required re-execution. However, in practice, file-level dependencies are too coarse-grained for many patches: for example, Warp re-executes all requests from our Wikipedia trace for about half of the patches (§7). POIROT uses finer-grained basic-block-level filtering, which filters out requests for many more patches. POIROT also requires less intrusive changes than Warp: it does not require any changes to the browser or the database, and does not require re-execution to take place on the production system.

POIROT's memoized re-execution is similar to dynamic slicing [3], 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 [4, 6]: 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. [18] 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.

Memoization has been used to speed up re-execution of an application over slightly different inputs [2, 15, 32]. Though POIROT's techniques can be extended to work for that scenario as well, memoized re-execution in the current 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.

POIROT's dependency analysis is similar to taint tracking systems [12, 25]. 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.

Dynamic dataflow analysis [10] and symbolic execution [8] 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.

10 Summary

This paper presented POIROT, a system that can audit past requests in a web application for exploits of a newly patched security vulnerability. POIROT incorporates three techniques—control flow filtering, function-level auditing, and memoized re-execution—to significantly speed up auditing compared to previous systems that audit through re-execution. POIROT is effective at detecting exploits of real vulnerabilities in MediaWiki and HotCRP. POIROT's optimizations allow it to audit challenging patches, which affect every request, 12–51× faster than the original execution time of those requests.

Acknowledgments

We thank David Terei for pointing us at prior work on self-adjusting computation [2]. We also thank Eddie Kohler, Neha Narula, Alex Pesterev, Jacob Strauss, Keith Winstein, Eugene Wu, the anonymous reviewers, and our shepherd, Mike Dahlin, for helping improve this paper. This research was partially supported by the DARPA CRASH program (#N66001-10-2-4089), by NSF award CNS-1053143, by Quanta, and by Google.

References

[1] Wikimedia labs database dump. `http://dumps.wikimedia.org/en_labswikimedia/20111228/`, December 2011.

[2] U. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the 35th ACM Symposium on*

Principles of Programming Languages, San Francisco, CA, January 2008.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[4] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of the IEEE Conference on Software Maintenance*, September 1993.

[5] F. E. Allen. Control flow analysis. In *Proceedings of the Symposium on Compiler Optimization*, 1970.

[6] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. Regression test selection techniques: A survey. *Informatica*, 35(3):289–321, October 2011.

[7] A. B. Brown and D. 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.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[9] R. Chandra, T. Kim, M. Shah, N. Narula, and N. 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.

[10] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. 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.

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

[12] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. 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.

[13] Github. SSH key audit. `https://github.com/settings/ssh/audit`, 2012.

[14] A. Goel, K. Po, K. Farhadi, Z. Li, and E. D. 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.

[15] P. J. Guo and D. Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, July 2011.

[16] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. 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.

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

[18] S. A. Khalek and S. Khurshid. Efficiently running test suites using abstract undo operations. *IEEE International Symposium on Software Reliability Engineering*, pages 110–119, 2011.

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

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

[21] E. Kohler. Hot crap! In *Proceedings of the Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems*, San Francisco, CA, April 2008.

[22] E. Kohler. Correct humiliating information flow exposure of comments. `http://www.read.cs.ucla.edu/gitweb?p=hotcrp;a=commit;h=f30eb4e52e91ab230944eebe8f31bf61e9783d3a`, March 2012.

[23] M. Maurer and D. Brumley. TACHYON: Tandem execution for efficient live patch testing. In *Proceedings of the 21st Usenix Security Symposium*, Bellevue, WA, August 2012.

[24] MediaWiki. MediaWiki. `http://www.mediawiki.org`, 2012.

[25] J. Newsome and D. X. 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.

[26] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[27] T. Preston-Werner. Public key security vulnerability and mitigation. `https://github.com/blog/1068`, March 2012.

[28] D. Rethans. Vulcan logic dumper. `http://derickrethans.nl/vld.php`, 2009.

[29] The Open Web Application Security Project. OWASP top 10. `http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf`, 2010.

[30] J. Tucek, W. Xiong, and Y. 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.

[31] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11): 1830–1845, 2009.

[32] A. Vahdat and T. Anderson. Transparent result caching. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[33] X. Wang, N. Zeldovich, and M. F. Kaashoek. Retroactive auditing. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011. 5 pages.