Understanding and Detecting Server-Side Request Races in Web Applications

Zhengyi Qiu
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
zqiu2@ncsu.edu

Shudi Shao
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
sshao@ncsu.edu

Qi Zhao
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
qzhao6@ncsu.edu

Guoliang Jin
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
guoliang_jin@ncsu.edu

ABSTRACT

Modern web sites often run web applications on the server to handle HTTP requests from users and generate dynamic responses. Due to their concurrent nature, web applications are vulnerable to server-side request races. The problem becomes more severe with the ever-increasing popularity of web applications.

We first conduct a comprehensive characteristic study of 157 real-world server-side request races collected from different, popular types of web applications. The findings of this study can provide guidance for future development support in combating server-side request races.

Guided by our study results, we develop a dynamic framework, ReqRacer, for detecting and exposing server-side request races in web applications. We propose novel approaches to model happens-before relationships between HTTP requests, which are essential to web applications. Our evaluation shows that ReqRacer can effectively and efficiently detect known and unknown request races.

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis**; **Software reliability**; **Software testing and debugging**; *Concurrency control*; *Organizing principles for web applications*; Consistency.

KEYWORDS

web-application request races, characteristic study, race detection, happens-before relationships

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468594

ACM Reference Format:
Zhengyi Qiu, Shudi Shao, Qi Zhao, and Guoliang Jin. 2021. Understanding and Detecting Server-Side Request Races in Web Applications. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468594

1 INTRODUCTION

Upon receiving HTTP requests, modern web sites dynamically generate responses by running some programs on the server. We refer to these programs as *server-side web applications*, and we refer to the code that handles each HTTP request as *a request handler*. Due to various types of concurrent activities in server-side web applications, request handlers serving HTTP requests could encounter race conditions while accessing shared resources and lead to erroneous behaviors depending on the order of these shared-resource accesses. Since these races happen on the *server side* of web applications while handling HTTP *requests*, we refer to such races as *server-side request races* or shortly as *request races*.

Several high-profile software failures were caused by request races, e.g., Instacart coupon double redemption [26], Starbucks gift-card duplicate balance transfer [36], and Flexcoin bankruptcy caused by wallet overdraw [30].

In the Instacart incident, a user reported that he was able to redeem the same coupon more than once with savings stacked by sending the coupon-redemption request multiple times [26]. Figure 1 shows a high-level explanation based on the incident description. While serving *one* coupon-redemption request, the corresponding request handler will issue *multiple* database queries, i.e., one query will first check whether a given coupon has been redeemed, and if it has not, more queries will mark the coupon as being redeemed and update the user account with the redeemed savings. Figure 1a shows the case when these two coupon-redemption requests are sent synchronously, i.e., sending one request after receiving the response of the previous request. In this case, these two requests are handled one-by-one, where only the first request will add savings to the user account, but the second request will inform the user that the coupon has already been redeemed. However, if a user sends the two requests asynchronously as shown in

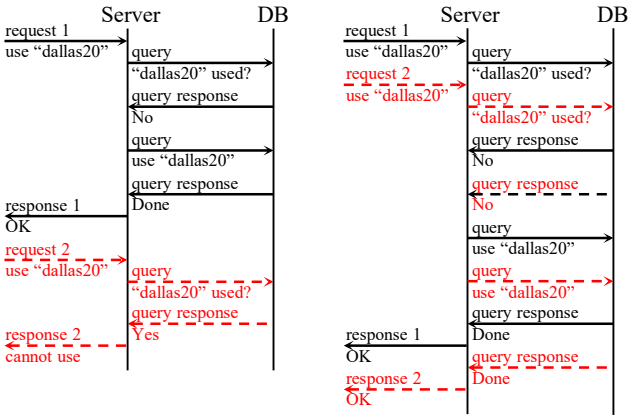

Figure 1: A high-level illustration of the Instacart incident

Figure 1b, i.e., sending one request before receiving the response of the previous request in a different, concurrent client session, the two requests are handled concurrently, and two concurrent instances of the same request handler are racing. Under the interleaving shown in Figure 1b, both requests first see that the coupon has not been used and then both use the coupon, resulting in the reported coupon double-redemption scenario.

As exemplified by these incidents, request races can lead to serious service corruption, severe security vulnerabilities, and huge financial losses. Two recent trends make request races an emerging threat to the reliability and security of web applications. First, the development of cloud platforms greatly eases the deployment of web applications, and the number of web applications increases. Secondly, getting access to web sites is eased by the increasing population of handheld devices, and the chance of encountering request races increases with more concurrent requests. We are in great need of a comprehensive understanding of request races and effective techniques for detecting them.

1.1 Limitations of Existing Work

In this multi-core era, a lot of research efforts have been spent on *thread races* in multi-threaded programs. Researchers have conducted characteristic studies [34, 55] and proposed various techniques for different purposes, e.g., bug detection [25, 33, 39, 67, 77, 78], program testing [49, 62, 68, 71, 74], failure diagnosis [20, 21, 42, 56], and fixing [41, 43, 52, 53]. Since web applications are commonly hosted on top of some multi-threaded programs, e.g., Apache HTTPD and MySQL, one may have the misconception that request races are a solved problem given the research progress on thread races.

However, thread-race detection techniques *cannot* detect request races. The request race shown in Figure 1 happens between two instances of the same request handler racing on database records, and the problematic interleaving on database queries manifests as an atomicity violation, where the violated atomicity assumption is among multiple queries issued by a single request handler. The request race cannot be detected as a thread race in the HTTP server program, as the shared data is stored in a database but not in the shared memory of the HTTP server program. It also cannot be detected as a thread race in the database program, as each database query is atomically processed when the request race happens in Figure 1. Therefore, there is no thread race from the viewpoint of the HTTP server or the database. Essentially, thread races happen in the system layer, which is below the application layer where request races happen. Through the discussion above, we can see that request races can happen even if there are no thread races, and thus thread-race techniques are not effective for request races.

Recent techniques for *process races* on the operating-system level [47] and *distributed concurrency bugs* in distributed and cloud systems [48, 50, 51, 54] are also not effective for request races. The reasons are generally two-folded. First, these techniques also target races in the system layer but not in the application layer, and the races detected by these techniques are not necessarily request races. Secondly, even if these techniques can detect some request races, they could report many false positives, as they do not consider application semantics while modeling happens-before relationships. We will expand the discussion on the second point later.

Not only one cannot directly use these existing techniques to detect request races, but server-side web applications also bring unique challenges to adapt existing approaches to detect request races. Dynamic race detection now commonly follows the tracing-inference-validation architecture, i.e., a tool first traces dynamic executions where races do not manifest, then infers races following common race manifestation characteristics, and finally validates the inferred races during replay runs. However, we face two major challenges to adapt this tracing-inference-validation architecture for request-race detection in web applications.

First, we currently lack a comprehensive understanding of real-world request-race characteristics to guide the three stages in the tracing-inference-validation architecture and make the approach effective and efficient. Only recently researchers have conducted characteristic studies on request races in applications developed with JavaScript on top of the Node.js framework [31, 72]. However, these two studies do not specifically focus on web applications but include many middleware and desktop applications, and their study results cannot provide sufficient coverage for real-world request races in representative server-side web applications.

Secondly, server-side web applications require new techniques to model happens-before relationships and determine which request handlers can run concurrently. For example, one common way to establish a happens-before relationship between two requests is to send the second request by clicking a button on the web page returned for the first request. Such happens-before relationships cannot be modeled by existing techniques that mostly focus on modeling happens-before relationships established with synchronization operations.

Recently, *client-side* races in web applications have drawn much attention from the research community. Researchers have proposed several techniques [18, 24, 40, 58, 63, 64, 76] for client-side race detection. Since they focus on the interaction between the HTML Document Object Model (DOM) and asynchronous event-driven JavaScript executions on the client side, the happens-before relationships modeled by existing techniques do not readily capture

dependencies between different HTTP requests. Although both the client side and the server side are integral parts of web applications, request races on the server side are arguably more critical as they often affect persistent system resources, e.g., the coupon redemption information in the Instacart incident stored in the database.

Lastly, specific to server-side request races, only a few groups of researchers have explored techniques to detect them so far and proposed dynamic [45, 61], static [80], and model-checking [35] approaches. The dynamic and static approaches [45, 61, 80] only detect request races between two instances of the *same* request handler, and one of them [45] further limits itself to one sub-type of request races. Since it is almost always possible to send the same request multiple times asynchronously with command-line tools and invoke multiple concurrent instances of the same request handler, there is no need to model happens-before relationships in the limited scope of request races targeted by these techniques. Although they can detect request races like the one in Figure 1, their design choices lead to major coverage issues, and they cannot detect request races between *distinct* request handlers, leading to false negatives. If one applies these prior techniques also for detecting races between different request handlers, every pair of request handlers will be considered as potentially concurrent, leading to a large number of false positives to be pruned with the costly replay stage. On the other hand, the model-checking approach [35] alleviates the need to model happens-before relationships between distinct request handlers by constructing concurrent test cases manually, limiting the applicability of the approach.

1.2 Contributions

To better understand request races and guide the design of request-race detectors, we first conduct a characteristic study on 157 server-side request races collected from the bug-tracking systems of web applications developed with different languages and frameworks, covering PHP, Perl, Python, C#, Java, Ruby-on-Rails, and Node.js. Our study is the largest and most comprehensive to date for server-side request races. It reveals many general findings on racing-resource types, manifestation conditions, root-cause patterns, the effects of races, and fix strategies, which are useful to guide the development of comprehensive tool support for request races. We also investigate how external factors, such as language/framework differences, affect the characteristics of request races.

Our study confirms several observations we made earlier: (1) request races are numerous and a real threat to the reliability and security of server-side web applications, (2) request races are indeed different from system-layer races that the root causes and fixes of all the studied request races are in the application layer, and (3) it is necessary to design new detection techniques that can handle request races beyond those between multiple instances of the same request handler, as a significant portion of the studied request races are between distinct request handlers.

We then present ReqRacer, a dynamic framework for detecting request races beyond those between two instances of the same request handler. To make it easy to deploy and broadly applicable, ReqRacer traces only on the server side and thus requires no change to browsers or other forms of clients. ReqRacer currently focuses on detecting request races manifest as atomicity violations, as they account for 137 out of the 157 request races we studied.

ReqRacer follows the tracing-inference-validation architecture mentioned earlier, and its design is guided by some key results from our characteristic study. Specifically, our study shows that all the studied races manifesting as atomicity violations can be triggered by two concurrent requests, i.e., the buggy interleavings that can lead to erroneous behaviors only involve shared-resource accesses issued by two concurrent request handlers with one request handler on each racing side. With this result, ReqRacer detects request races by checking whether there are commonly seen unserializable interleaving patterns either between two instances of the same request handler or between two distinct request handlers that can be concurrent. To this end, ReqRacer reports two request handlers as having a true, harmful race if their corresponding requests are concurrent and the inferred unserializable interleavings cause errors or undesired behaviors in replay runs.

Our major contribution lies in ReqRacer's novel approach for constructing a dependency graph to model happens-before relationships between requests, with which ReqRacer recognizes requests that are potentially concurrent. ReqRacer models two types of dependencies that are common in web applications. The first is a *Request-Response-Request (RRR) dependency*, and it exists when one request can only be sent after the response of the previous request has been received by the client. The second is a *Select-by-Primary-Key (SPK) data dependency*, and it exists when a latter SELECT query specifies a primary key and retrieves one row inserted by an earlier query. The RRR dependency is natural for web applications, and the purpose of the SPK-data dependency is to quickly prune request races that would otherwise result in replay divergences during the validation stage, where a replay divergence happens when the request handlers involved in the request race to validate execute significantly different business logic comparing the recorded run and replay run. Similar to the previous work on detecting and validating process races [47], ReqRacer prunes a request race as a false positive when a replay divergence happens, and the SPK-data dependency can greatly reduce the number of false positives that need to be pruned through replay.

We implemented a prototype of ReqRacer for the LAMP (Linux, Apache, MySQL, and PHP) stack. We evaluated ReqRacer with 12 request races that we reproduced from our studied bugs. These 12 bugs cover all the four web applications used in our study that are developed with PHP, i.e., MediaWiki, WordPress, Moodle, and Drupal. Our evaluation results show that ReqRacer can effectively and efficiently detect and expose these known bugs based on traces from recorded runs where request races do not manifest. ReqRacer also found at least four new request races that were unknown to us with the testing workloads for the 12 real-world bugs, and two of them have already been confirmed by developers.

2 SERVER-SIDE REQUEST-RACE STUDY

In our characteristic study, we follow the same methodology as taken by existing studies on concurrency bugs in multi-threaded applications [55], races in Node.js applications [31, 72], and performance bugs in web applications [69, 73]. Below, we first describe the methodology and then detail the results.

Table 1: Web applications and numbers of bugs being studied

Application (Abbreviation)	Server-Side Language	Number on Server Side	Number on Client Side
WordPress (WP) [16]	PHP	18	22
MediaWiki (MW) [8]	PHP	28	10
Drupal (DPL) [5]	PHP	31	7
Moodle (MDL) [9]	PHP	15	8
BugZilla (BZ) [1]	Perl	11	0
Odoo (OD) [12]	Python	2	8
DNN (DNN) [4]	C#	4	1
OpenMRS (MRS) [13]	Java	2	0
Gitlab (GL) [6]	Ruby	26	5
Discourse (DC) [3]	Ruby	7	2
Redmine (RM) [14]	Ruby	2	0
Spree (SPR) [15]	Ruby	0	2
Node.js-based (Node.js)	Javascript	11	1
Total		157	66

2.1 Methodology

Table 1 summarizes the number of studied race-induced bugs and web applications from the bug-tracking systems of which these bugs are collected. We study bugs from two major types of web applications: (1) classical ones that access database by constructing SQL queries directly, and (2) those implemented on top of Ruby-on-Rails, which provides Object-Relational-Mapping (ORM) support. We started with popular open-source web applications that have been previously studied by existing performance-bug studies [69, 73], which have 7 non-ORM applications and 12 ORM applications, respectively. We also included extra non-ORM web applications that we have experience with.

To collect races, we searched across the bug-tracking systems of these applications for closed bugs using keywords related to races, such as "race(s)," "concurrency," "atomic," and "synchronization." Then, we manually filtered out results obviously not related to races, e.g., "braces" or "traces" can be returned while searching for "race." After keyword search and filtering, we obtained around 1400 bug reports. We then manually examined each bug with sufficient information for us to understand the root cause, and we collected bugs with clear root causes that are related to races. Our final selection includes a total of 211 bugs from 12 popular open-source web applications covering six different server-side programming languages. We omit applications without any race-induced bugs.

We also included race-induced bugs in Node.js-based web applications from the two previous studies on Node.js projects [31, 72], the total of which is 12 as other bugs are either in Node.js-based middleware or desktop applications but not web applications. Since these 12 bugs are in 12 different Node.js web applications, we omit their names due to space limit.

After collecting race-induced bugs from the bug-tracking systems of these web applications, two inspectors first independently checked all available resources, e.g., source code, developer discussions, and patches, compared their characterization results, and resolved disagreement if any.

Among the 223 bugs we collected, 157 are induced by server-side races, and 66 are induced by client-side races. Note that we included all race-induced bugs that we could understand regardless of whether they are on the server side or on the client side while collecting the bugs, and the separation of server-side races from client-side races was done after we collected all race-induced bugs. Such a distribution shows that server-side races are indeed under-studied compared with client-side races and warrant more research attention. Our investigation focused on the server-side races, and future work can further study the client-side races.

2.2 Characteristic-Study Results

Table 2 summarizes our findings. Before describing the details, we note that two general observations we made during our characteristic study are aligned with our argument made earlier, i.e., existing system-layer race detection techniques are not effective for request races. First, all the server-side races in web applications we studied are in the application layer but not in the system layer. Secondly, we do not see a case where developers used existing race detection tools to help debugging, which arguably implies that (1) practical thread-race detection tools commonly mentioned in thread-race bug reports are not useful for request races, and (2) tool support for debugging request races is in great need. Below, we detail the results on the following characteristics: racing-resource types, root-cause patterns and manifestation conditions, the effects of races, and fix strategies. We end this section with a discussion on how these results depend on external factors.

Racing-resource types. Databases, files, cache using modules like Redis or Memcached, and shared-memory data structures are the racing resources we found in our studied request races. Four bugs involve consistency issues between database and cache, and their racing resources include both databases and cache. While thread-race techniques cover shared-memory data structure and process-race techniques [47] cover files as a racing resource, tools targeting request races need to further model and analyze accesses to databases and cache. Further, some races involve inconsistency between database and cache, which needs to be accounted for.

Root-cause patterns and manifestation conditions. In our studied bugs, atomicity violation and order violation are the two root-cause patterns of how the shared-resource accesses in racing request handlers can lead to failures or undesirable effects. All studied races involving only one request, i.e., intra-request races, manifest as order violations, and all studied races involving two requests, i.e., inter-request races, manifest as atomicity violations.

For atomicity violations, all studied races can manifest with two racing request handlers, and the violated atomicity assumption is on accesses within a single request handler but not across multiple request handlers. As a result, we only need to check pairs of concurrent request handlers to detect request races manifesting as atomicity violations on shared-resource accesses. However, not all atomicity violations can manifest with two instances of the same request, which will be handled by the same request handler. One-third of them require two requests that will be handled by different request handlers. Existing dynamic request-race detection techniques in literature [45, 61] only check races between a request handler with itself, and they will miss all races requiring two different request handlers. We are in great need of techniques that can detect request races between distinct request handlers.

For order violations, all the studied cases are due to asynchronous execution. Since modern ORM and Node.js frameworks have built-in support for asynchronous execution, web applications built with such frameworks are more prone to order violations, which is partly

Table 2: Overall characteristic-study results

	WP	MW	DPL	MDL	BZ	OD	DNN	MRS	GL	DC	RM	Node.js	Total
Number of bugs studied	18	28	31	15	11	2	4	2	26	7	2	11	157
Racing-Resource Types													
Database	11	20	18	11	11	1	1	0	19	5	2	5	104
File	3	3	10	4	0	1	3	1	2	2	0	0	29
Cache with modules like Redis or Memcached	5	5	4	1	0	0	0	0	1	0	0	0	16
Shared-memory data structure	0	1	0	0	0	0	0	1	4	0	0	6	12
Root Cause Patterns and Manifestation Conditions													
Atomicity violation with two instances of the same request handler	10	21	22	9	6	2	3	2	6	2	2	6	91
Atomicity violation with two different request handlers	8	7	9	6	4	0	1	0	6	3	0	2	46
Order violation within one request handler	0	0	0	0	1	0	0	0	14	2	0	3	20
Effects of Database Races													
Database error on duplicate data insertion	2	11	11	1	3	0	0	0	3	0	0	0	31
Application error caused by duplicate data	4	0	2	4	1	1	1	0	1	0	1	1	16
Inconsistent or stale view	5	8	5	6	7	0	0	0	11	4	0	4	50
Misleading error message	0	1	0	0	0	0	0	0	0	0	0	0	1
Program crash or failure	0	0	0	0	0	0	0	0	4	1	1	0	6
Fix Strategies for Database Races													
Atomicity violation: Change application logic	4	5	3	2	1	0	0	0	7	1	0	0	23
Atomicity violation: Handle the race properly when it happens	2	6	4	5	1	0	0	0	3	0	0	0	21
Atomicity violation: Guarantee column value uniqueness	2	0	0	3	1	0	1	0	1	0	1	1	10
Atomicity violation: Use duplication-tolerant SQL query	1	5	0	0	0	0	0	0	0	0	0	0	6
Atomicity violation: Refactor query statement	0	1	7	1	0	1	0	0	0	1	0	0	11
Atomicity violation: Add transaction or leverage existing transaction	0	0	0	0	3	0	0	0	0	1	0	0	4
Atomicity violation: Lock the table	0	2	3	0	4	0	0	0	0	0	1	0	10
Atomicity violation: Use atomic API	0	0	0	0	0	0	0	0	0	0	0	3	3
Atomicity violation: Fix in frontend	2	1	1	0	0	0	0	0	0	0	0	0	4
Order violation: Enforce order by callback function	0	0	0	0	0	0	0	0	5	1	0	1	7
Order violation: Use synchronous execution	0	0	0	0	0	0	0	0	2	0	0	0	2
Order violation: Remove transaction	0	0	0	0	1	0	0	0	1	1	0	0	3
Effects of File Races													
Duplicate file/directory creation	1	1	5	1	0	1	1	0	1	0	0	0	11
File data corruption	1	1	3	2	0	0	1	0	0	1	0	0	9
Non-existing file/directory error	1	1	2	1	0	0	1	0	1	0	0	0	7
Misleading error message	0	0	0	0	0	0	0	1	0	1	0	0	2
Fix Strategies for File Races													
Atomicity violation: Change application logic	0	0	1	1	0	0	0	0	0	0	0	0	2
Atomicity violation: Handle the race properly when it happens	0	1	7	2	0	1	0	1	2	1	0	0	15
Atomicity violation: Guarantee file name uniqueness	1	2	1	1	0	0	0	0	0	1	0	0	6
Atomicity violation: Add file lock	2	0	1	0	0	0	3	0	0	0	0	0	6
Effects of Cache Races													
Inconsistent or stale view	5	5	4	1	0	0	0	0	1	0	0	0	16
Fix Strategies for Cache Races													
Atomicity violation: Change application logic	2	4	3	0	0	0	0	0	0	0	0	0	9
Atomicity violation: Handle the race properly when it happens	0	1	1	1	0	0	0	0	0	0	0	0	3
Atomicity violation: Guarantee cache key uniqueness	3	0	0	0	0	0	0	0	0	0	0	0	3
Order violation: Use proper cache API	0	0	0	0	0	0	0	0	1	0	0	0	1
Effects of Shared-Memory Data Structure Races													
Application error or exception	0	1	0	0	0	0	0	1	4	0	0	0	6
Request never be responded to	0	0	0	0	0	0	0	0	0	0	0	2	2
Program crash or failure	0	0	0	0	0	0	0	0	0	0	0	4	4
Fix Strategies for Shared-Memory Data Structure Races													
Atomicity violation: Use database instead	0	1	0	0	0	0	0	0	0	0	0	0	1
Atomicity violation: Add language-provided lock	0	0	0	0	0	0	0	1	0	0	0	0	1
Order violation: Change application logic	0	0	0	0	0	0	0	0	2	0	0	4	6
Order violation: Add condition to enforce order	0	0	0	0	0	0	0	0	0	0	0	2	2
Order violation: Read again after a delay	0	0	0	0	0	0	0	0	2	0	0	0	2

reflected by the fact that only one out of the 20 order violations are in the first eight non-ORM, non-Node.js web applications.

Unserializable interleaving patterns in atomicity violations. To summarize unserializable interleaving patterns in atomicity violations, we follow the arguments raised by Zheng and Zhang [80] in their work of static detection of request races manifesting as atomicity violations. Specifically, operations like file append and database delete cannot be modeled as write operations. For example, a local SELECT query and a local DELETE query interleaved with a remote DELETE query on the same database record is unserializable if they are modeled as $RW'W$. However, this is wrong as it has the same consequence as first performing the local SELECT and DELETE queries and then the remote DELETE query. Therefore, they introduced two new operation categories, i.e., A for append and D for delete, and W is now specifically for overwrite or update. With these four types of operations, $ADRW$, defined, a lot of interleaving patterns are possible. They further argued that a lot of them are unlikely in practice and suggested four unserializable interleaving

Table 3: The patterns of unserializable interleavings and their numbers in our studied request races

	Pattern	Number
1	$(\epsilon\|R)R'(A\|W\|D)(A'\|W'\|D'\|R')$	114
2	$A(A'\|R'\|W')A$	2
3	$W(A'\|R')(W\|A)$	3
4	$DD'AA'$	4
5	$(A\|W)(A'\|D'\|W')R$	14

patterns that could happen in practice: (1) $RR'(A|W|D)(A'|W')$, (2) $A(A'|R'|W')A$, (3) $W(A'|R')(W|A)$, and (4) $DD'AA'$. [1]

With these four unserializable interleaving patterns, we first matched our studied atomicity violations involving a single resource against them. For those without an exact match, we modified a close match or proposed a new pattern. Table 3 summarizes the results. The first four patterns are mostly the same as the ones by Zheng and Zhang [80] with the first pattern modified. These four patterns cannot be further merged, as the first operations of the four patterns are different. Among these four patterns, the first one is the most common in our studied bugs. We also found one new pattern with 14 bugs matching it, where most of these bugs are races on cache and shared data structures. We did not merge Pattern (5) with Pattern (2), as that results in $(A|W)(A'|R'|W'|D')(A|R)$, but $AR'R$ is serializable. All the four races involving both database and cache are categorized as Pattern (5). Specifically, the buggy interleaving is that one request first queries database and caches the queried results. Before the cached results are accessed, another request updates the database without invalidating the cache, and the first request will later access stale data that is not consistent with the database.

Although the first four patterns are initially proposed by Zheng and Zhang [80] based on intuition, the value of our results lies in that we confirm that they are indeed the most common ones with our collected real-world bugs and we further refine them. We further summarize one new pattern, i.e., Pattern (5), which Zheng and Zhang [80] suggested as being unlikely, but we observed such a pattern in real-world request races partly due to our inclusion of request races on cache and shared data structures.

The effects of races. 47 out of the 104 studied request races on databases can lead to duplicate data insertion. Among them, 31 result in database errors, and 16 result in application errors. The majority of the remaining database races lead to inconsistent or stale views. File races can lead to various errors, such as duplicate file/directory creation or non-existing file/directory errors. File races can also lead to data corruptions. All cache races lead to inconsistent or stale views. Races on shared-memory data structures can lead to various failures and errors. They can also lead to requests never be responded to, and such effects are not observed on races on other resources. For some of these races, the failure effects are easy to detect, such as those resulting in explicit errors. For others that lead to inconsistent views or data corruptions on shared resources, some types of checkers taking application semantics into account are needed to catch the effects.

Among the 91 races on two instances of the same request handler, 63 of them are racing on databases. Further, the aforementioned 47 races leading to duplicate data insertion with either database or application errors are all among these 63 and can manifest with two instances of the same request handler. These numbers suggest that we can use an effect-oriented approach [37, 77, 78] to find a large portion of database races between two instances of the same request handler by focusing on detecting request races with the duplicate data-insertion effect.

Fix strategies. 21 out of the 137 studied atomicity violations are fixed using database locks, transactions, file locks, or language-provided locks to provide atomicity. The patches for the majority of remaining atomicity violations involve design or application-logic changes. These results are consistent with the conventional wisdom that there are few synchronization operations to use on the server side, and thus some design or logic changes are often needed to fix request races. For order violations, using callback functions and changing application logic are the two major fix strategies.

One interesting finding on fix strategies is that four server-side request races are fixed by changes in client-side code, such as disabling a button on the client side if the browser has not yet received the response for an earlier request, which was sent by clicking the same button, to avoid two concurrent instances of the same request.

Dependency on external factors and discussion. In our study, we paid attention to several external factors, including web-server configuration, database configuration, and development languages and frameworks, and studied the impact of these external factors.

Some web servers, e.g., Apache HTTPD, commonly provide different multi-processing modules, i.e., prefork, worker, and event [10], while web applications built on top of the Node.js framework use the event-based model. In our study, we did not find the underlying multi-processing model affect the way how we define server-side *request* races, i.e., the studied races are on request handlers upon receiving HTTP requests. Such a definition abstracts away whether a request is served by a process or thread and whether a request is served with an event-based model. On the other hand, we found request races whose manifestation condition depends on database configuration. For example, the MyISAM storage engine in MySQL does not support transactions [11], and we have seen MediaWiki races that are caused by the use of transactions, which only manifest if MyISAM but not InnoDB is used in MySQL.

In terms of the dependency on development languages and frameworks, built-in support for asynchronous execution could make web applications more prone to order violations as we discussed earlier. We also note that non-ORM, non-Node.js web application has the least number of races racing on shared-memory data structure, as their development languages provide little support for shared memory natively. On the other hand, some other number differences may not be depending on language/framework. No Node.js request races are on file or cache, which is probably due to the smaller total number of request races being studied. Also, the differences on the total numbers of request races we studied in different types of web applications, i.e., we study the largest number of request races in non-ORM, non-Node.js web applications and the smallest number

[1] Each pattern has four or three operations. The first and third operations are from one request. The second and fourth, which are marked ', are from the second request. If one operation has multiple possibilities, they will be put inside "()" and separated by '|'. *A* stands for append operations, which include database insert, file append, file/dir create, and cache add; *D* stands for delete operations, which include database delete, file/dir delete, and cache delete; *R* stands for read operations, which include database select, file/dir read, cache read; *W* stands for write operations, which include database update, file overwrite, and cache set or replace; and ϵ stands for NULL operations.

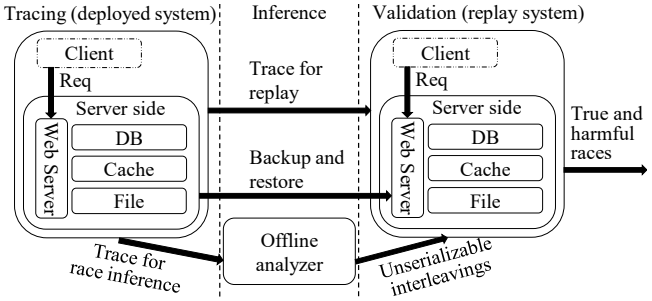

Figure 2: The architecture of ReqRacer

in Node.js web applications, are more related to the development history length of web applications being studied.

Although we included some bugs from Node.js-based web applications, which were studied in earlier studies [31, 72], our focus is different, as we specifically focus on web applications but their studies include other types of applications built on top of Node.js, e.g., middleware and desktop applications. Due to this difference, our characteristic study is essentially targeting a different subject. Moreover, our study includes 146 more request races from non-Node.js web applications, which allows us to both find characteristics that are common to all types of web applications and to understand the impact of development language/framework differences.

3 SERVER-SIDE REQUEST-RACE DETECTION

Based on our characteristic-study results, we design a dynamic framework, ReqRacer, to detect and expose server-side request races that manifest as atomicity violations. Figure 2 illustrates the architecture of ReqRacer, and it has three major stages. The first stage records four types of runtime information, and they are used for determining shared-resource accesses, reasoning about dependencies between requests, and enabling execution replay. The second stage consumes the traces recorded during the first stage and infers potential unserializable interleavings to detect racing requests. The last stage replays the recorded traces, tries to enforce the unserializable interleavings inferred by the second stage, and observes their effects. Racing requests with triggered harmful unserializable interleavings will be reported.

3.1 Illustrating Example

The key contribution of ReqRacer is to model happens-before relationships essential to HTTP requests in web applications with a novel form of dependency graphs, so that the well-established tracing-inference-validation architecture can be applied to detect request races.

We will use WordPress 11073 to illustrate the dependency-graph construction process shown in Figure 3. The request race is between adding a comment for a post and trashing the post. While trashing a post in one request, the IDs and current statuses of its comments are first backed up for possible future restoration, and then the statuses of all existing comments will be marked as trashed. In between these two steps, another concurrent request can add a new comment to the post being trashed. Under this situation, the comment is first successfully added to the post and displayed to

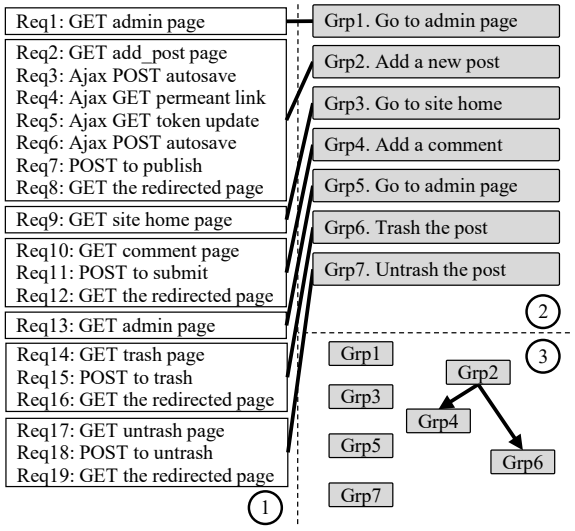

Figure 3: An example of dependency-graph construction

the user, and it is then marked as trashed since the post is being trashed. However, the new comment is not backed up, and this comment cannot be restored if the post is untrashed later. The user who added the new comment will observe inconsistent views, as the comment was first added successfully, but later the comment will be gone if the user refreshes the post that just went through the trash and untrash process.

Now assuming a developer is doing some testing for WordPress. In a browser, the developer first goes to the admin page and adds a new post, then goes to site home and adds a comment for the newly added post, and finally goes back to the admin page to trash and then untrash the post. In this sequence, the harmful race is not triggered. During this process, ReqRacer records a sequential trace as shown in ①, and it then constructs the dependency graph as shown in ③. Note that the boxes surrounding the recorded requests in ① are added for illustration purposes but not part of the trace.

With these recorded requests, one extreme is to consider all recorded requests as totally ordered, under which case we will not be able to infer any potential races. The other extreme is to consider that all requests can potentially be sent concurrently via different browsers or tabs, under which we will be inferring too many false positives. Therefore, we only relax orders between truly independent requests but preserve orders that, once relaxed, can disable subsequent requests and lead to replay divergences. In Figure 3, ReqRacer will go through ①→ ②→ ③ to find potentially concurrent requests from the sequence of recorded requests.

From ① to ②, ReqRacer groups requests that have a Request-Response-Request (RRR) dependency, i.e., one request can only be sent after the response of the previous request has been received by the client. Such an RRR dependency can be established in three ways in Figure 3. First, a POST request could depend on the previous GET request, and this applies to *Req2→Req7*, *Req10→Req11*, *Req14→Req15*, and *Req17→Req18*. Second, a GET request to a redirected page depends on the previous request, and this applies to *Req7→Req8*, *Req11→Req12*, *Req15→Req16*, and *Req18→Req19*. Third, Ajax requests depend on the previous GET request, and this

applies to *Req2*→{*Req3*, *Req4*, *Req5*, *Req6*}. Grouping requests with dependencies together, we get a total of seven groups shown in Figure 3.②. ReqRacer recognizes these RRR dependencies through program modifications to embed tokens into requests and responses. Note that the text description in Figure 3.② is just added to ease understanding but not inferred by ReqRacer.

From ② to ③, ReqRacer adds an edge between two request groups if there is a special type of data dependencies between them, i.e., when a request issues a SELECT query with a primary key, and the database returns a single row that was inserted by a query from an earlier request. We name it as Select-by-Primary-Key (SPK) data dependency. The rationale is that if one considers two requests with an SPK-data dependency as concurrent and forcefully send the latter request before receiving the response of the former request, the SELECT query in the latter request will return zero rows, which could significantly affect the execution of the corresponding request handler compared with the recorded run where it gets one row, and it will result in a replay divergence.

For example, the SPK-data-dependency edge from *Grp2* to *Grp4* in Figure 3.③ is added because one query sent to database while handling *Req10* queries a singleton of result specified with a primary key that was inserted by *Req7*. As it is a singleton inserted by *Req7*, *Req10* will get a NULL result if we relax the order between *Req7* and *Req10*, which can significantly affect the request handlers handling *Req10* and other requests following it in the same group. As a result, we will not be able to add comments to the post in *Grp4* if the post has not been added in *Grp2* yet. The other SPK-data-dependency edge is added for a similar reason. ReqRacer recognizes such data dependencies by recording database queries and selected responses.

Note that our SPK-data-dependency is different from general data dependency where a latter query reads data inserted or updated by an earlier query. This is because the effect of changing the order of two queries with the general data dependency highly depends on the business logic of the corresponding request handlers and can only be observed during replay runs in general. However, if the SELECT query specifies a primary key, the corresponding request handler will have significantly different business logic for cases where the query gets one row or gets no row, and we can infer that there will be replay divergences if the order is not enforced.

With the dependency graph in Figure 3.③ and recorded traces on accesses to shared system resources, two requests are potentially concurrent if they are in different request groups that are not reachable on the dependency graph. In Figure 3, ReqRacer determines that *Grp4* and *Grp6* are potentially concurrent, and they contain *Req11* and *Req15*, which are the racing requests in WordPress 11073. After inferring concurrent requests, ReqRacer further checks unserializable interleavings and validates them, and the race between *Req11* and *Req15* will be reported as a true, harmful request race.

3.2 Tracing and Request-Race Inference

ReqRacer employs server-only tracing but requires no modifications on the client side, making it easy to deploy and broadly applicable. In particular, ReqRacer records (1) HTTP requests received by the HTTP server, (2) accesses to shared resources including database queries and calls to cache APIs, (3) the return value of functions that are used to get the primary key of the row inserted by a previous INSERT query, and (4) unique tokens generated while formulating HTTP responses. Shared-resource accesses and tokens are attributed to the request that initiates them.

With dynamic execution traces, ReqRacer first identifies *conflicting* requests, i.e., a pair of requests whose request handlers both access some shared resource in common and at least one request handler performs logical append, delete, or write operations to the resource. ReqRacer then builds a dependency graph as illustrated in Figure 3, and two request handlers that are in different groups and not reachable from each other are considered as concurrent. Finally, ReqRacer reports two conflicting and concurrent request handlers as racing if the two request handlers can exhibit unserializable interleavings of shared-resource accesses. Below, we describe each of these three steps, including what information is traced and how the traced information is analyzed.

While our framework can be used for all types of racing resources that we have seen in our study, our prototype implementation of ReqRacer handles databases and cache. To extend ReqRacer for files, one needs to incorporate techniques of tracing and analyzing file accesses previously used for process races [46, 47] into our framework. To extend ReqRacer for shared-memory data structures, one needs to further trace and analyze shared-memory accesses. In both cases, happens-before relationships modeled by ReqRacer's dependency graphs could be reused.

3.2.1 Identifying Conflicting Requests. ReqRacer considers two requests as conflicting if their corresponding request handlers contain conflicting accesses to either databases or cache.

On database queries, ReqRacer currently logs SELECT, INSERT, UPDATE, and DELETE. For SELECT, UPDATE, and DELETE queries, we determine the rows of data elements being accessed based on the WHERE clause. An INSERT query adds a new row to a table with the value of each column specified. We extract the column-value pairs from an INSERT query string. These four types of queries are sufficient in our evaluation, and more can be added if necessary.

For two INSERT queries, if they operate on the same unique key, and the values inserted to the unique key are the same, these two INSERT queries are conflicting. Otherwise, if no unique key appears in an INSERT query string, ReqRacer currently considers two INSERT queries conflicting if they insert the same data.

To find conflicts between a query with a WHERE clause and an INSERT query, we compare the WHERE-clause conditions with the column-value pairs from the INSERT query. If the column-value pairs satisfy the conditions, the two queries are conflicting.

To find two conflicting queries both with a WHERE clause, we compare their conditions. If there is some intersection, we conservatively consider these two queries as conflicting.

One special case is a SELECT * query without a WHERE clause. For this type of SELECT query, we consider it as accessing the whole table and conflicting with any query that modifies the table.

On cache accesses, ReqRacer currently logs Add, Delete, Get, Set, and Replace operations. Two cache accesses are conflicting if they have the same key and at least one of them is an Add, Delete, Set, or Replace operation.

3.2.2 Dependency-Graph Construction. As illustrated in Section 3.1, ReqRacer constructs the dependency graph in two steps.

To model the RRR dependency between two requests, where the second request can only be sent after the response of the first request has been received by the client, we modify the web application to include a random token unique to each request when formulating responses to the client, and it leverages the request-response-request chain established by the random token embedded in responses to capture RRR dependencies.

There are four scenarios where this token is embedded. For forms in HTML responses, we add a new hidden field to the forms with the token value. For embedded URLs in HTML responses, we add the token as a URL parameter. For HTTP redirection responses, we also add a URL parameter to the response URL. In these three cases, later requests sent through the token-embedded HTML elements or URLs will automatically carry the random token. For all HTML page responses that could enable Ajax requests, we add the token value as a meta property under the head tag of the response HTML and registers a new function with jQuery's ajaxPrefilter interface, where the function will read and attach the token value to all Ajax requests sent from this page. To avoid potential client-side races on these token values, the random token value is always parsed and loaded as part of the response before subsequent requests that need to carry the token value can be sent. Note that similar mechanisms are used to implement Cross-Site Request Forgery (CSRF) tokens [2]. While CSRF token is unique for every web session, our new token will be unique for every response. Our implementation leverages existing CSRF-token code to embed our new token for RRR dependency.

With these modifications, ReqRacer can establish a request-response-request chain and add an RRR-dependency edge between two requests, when the second request contains a token value that matches the token value embedded into the response of the first request. After identifying pair-wise dependencies based on tokens, these requests are further grouped until no two requests from different groups have an RRR dependency.

After grouping requests based on RRR dependencies, ReqRacer next adds edges between request groups with SPK-data dependencies, which happens when one SELECT query with a primary key specified in the second group gets one row inserted by a query in the first group. Specifically, if (1) a request *rqAi* in request group *GrpA* contains an INSERT query and the newly inserted row has value v on a primary key column *ColP*, (2) a request *rqBj* in a different request group *GrpB* which contains a SELECT query with a WHERE clause *ColP*= v, and (3) *GrpA* appears before *GrpB* in the trace, ReqRacer adds an SPK-data dependency edge from *GrpA* to *GrpB*. Note that requests after *rqAi* in *GrpA* will still be considered as potentially concurrent with requests in *GrpB* and will be checked by ReqRacer.

We do not add data-related dependency edges on cache, as a request handler will usually bring the data into cache by itself in cases of cache misses without relying on other request handlers to bring the data in.

3.2.3 Request-Race Inference. ReqRacer currently focuses on detecting request races with atomicity violations as their root causes. With the dependency graph constructed, ReqRacer then checks whether two conflicting, concurrent request handlers have shared-resource accesses that can exhibit unserializable interleavings with the patterns shown in Table 3. ReqRacer identifies potential request races for validation if unserializable patterns are found.

To detect races between two requests that will be served by two instances of the same request handler, ReqRacer duplicates the selected request handler, considers the original request handler and the duplicated request handler as concurrent, and applies the same checking of unserializable interleavings. The duplicated request handler will only be checked against its origin but not other request handlers. We further follow the effect-oriented approach [37, 77, 78] to handle the majority of races between two instances of the same request handler as discussed in Section 2.2, i.e., we focus on finding duplicate data-insertion races by duplicating a request only if the corresponding request handler issues one SELECT query and one INSERT query that are conflicting.

3.3 Replay-Based Validation

While the RRR and SPK-data edges help to reduce the number of false positives to be pruned by the validation stage, replay is still necessary to validate the remaining request races by observing the effect of enforcing specific interleavings to determine whether they lead to errors or replay divergences. ReqRacer reports a true, harmful request race only if it detects failures.

In the replay stage, ReqRacer replays recorded requests and intercepts their responses. In one replay session, ReqRacer replays requests until reaching one request in the request race to validate. While this replay session is paused before the first racing request, ReqRacer replays the other racing request in a different replay session, and all requests that the second racing request has general data dependence, as defined in Section 3.1, on but have not been replayed will be replayed in the order they appear in the trace. To validate duplicated instances of the same request handler, we simply replay the same request twice.

With both replay sessions pausing before racing requests, ReqRacer controls the execution of the racing request handlers to make shared-resource accesses follow the order of unserializable interleavings. To achieve this, we insert delays in the database execution engine and cache-access APIs to control the order of accesses to databases and cache. As not all interleavings are feasible to enforce, ReqRacer gives up an interleaving if one access has been waiting for a pre-defined timeout value but it is not the access to proceed according to the interleaving to enforce. ReqRacer currently sets the timeout value to 10 seconds. ReqRacer also gives up an interleaving if a response indicates an error and reports the request race as a true positive. If an interleaving is successfully enforced, ReqRacer will detect failures by checking whether there are application errors, database errors, or errors emitted by programmer-supplied application-specific checkers. If a failure is detected, ReqRacer reports the request race as a true positive. In all other cases, the inferred request race is pruned as a false positive.

To enable replay-based validation, we need to create backups for persistent system resources so that they can be restored onto the replay systems. For database states, ReqRacer uses the backup and restore functionality provided by databases. As cache is less persistent, cache state is not backed up for restore, and it is populated during replay based on database states.

Table 4: Overall evaluation results. "Reqs" shows the number of requests in the workload. "#Acc." represents the number of database or cache accesses in the trace. "Confl. Reqs" shows the number of conflicting request pairs. "RRR", "SPK", "S", and "R" show the numbers of conflicting request pairs pruned by checking RRR dependency, SPK-data dependency, serializability, and replay, respectively. "TP" and "FP" show the numbers of true positives and false positives. "Likely TP" is for cases that can be detected if application-specific checkers are added. Numbers with an '*' are unknown to us while devising the workload.

Bug information				Between distinct request handlers								Between two instances of the same request handler					
Bug ID	Racing resource	Reqs	#Acc.	Confl. Reqs	RRR	SPK	S	R	TP	Likely TP	FP	Confl. Reqs	S	R	TP	Likely TP	FP
WP 11073	Database	24	533	118	-22	-15	-74	-6	1	0	0	3	0	-2	0	1*	0
WP 11437	Database	9	181	26	-10	-2	-11	-2	0	1*	0	1	0	0	1	0	0
WP 24933	Database	15	277	18	-3	-8	-6	0	1	0	0	2	0	-2	0	0	0
MW 40594	Database	18	1008	14	0	0	-14	0	0	0	0	1	0	0	1	0	0
MW 69815	Database	23	2069	11	0	0	-11	0	0	0	0	1	0	0	1	0	0
MDL 28949	Database	47	2878	407	-37	-51	-297	-17	1	4*	0	11	0	-4	2/4*	1*	0
MDL 43421	Database	31	1141	122	-11	-48	-44	-18	0	1*	0	5	0	-1	1/1*	1*	1
MDL 51707	Database	14	250	21	-4	0	-16	0	1	0	0	3	-3	0	0	0	0
MDL 59854	Database	101	1969	492	-25	-81	-375	-10	0	1*	0	20	-12	-2	2/3*	1*	0
DPL 1484216	Database	11	422	6	0	0	-3	-3	0	0	0	1	0	0	1	0	0
WP 15545	Cache	23	412	24	-2	0	-22	0	0	0	0	2	0	0	1	1*	0
WP 20786	Cache & Database	14	263	10	-2	0	-7	0	1	0	0	9	-7	-2	0	0	0

4 IMPLEMENTATION AND EVALUATION

We implemented a ReqRacer prototype for the popular LAMP stack, i.e., Linux, Apache, MySQL, and PHP. The prototype consists of components for server-side tracing and replay-based validation, which is implemented by modifying PHP, MySQL, and application-specific cache APIs. The inference step is implemented using Python. We use an open-source tool, Gor [7], to capture and replay HTTP requests. To enable token-based dependency tracking, we currently manually modify the applications to embed the tokens leveraging existing CSRF token sites. To embed tokens, we first determine the names of CSRF tokens used by an application. In our experiments, we can get the names effectively by checking the hidden-field names in HTML responses through a browser. Once we get the names of CSRF tokens, we search the application code to find the sites where such CSRF tokens are embedded. We finally embed and log our tokens at these sites following the rule of how CSRF tokens are embedded in the application.

We do not automate cache-API changes and token embedding, as they are application-specific. Fortunately, our experience suggests that places where we made changes are well modularized, and we expect the workload of porting ReqRacer to new applications under the LAMP stack to be small.

To evaluate the effectiveness and efficiency of ReqRacer in detecting request races, we mimic the way how developers may test their web applications, i.e., by clicking buttons on the browser to visit various pages and use various functionalities, and we leverage known real-world bugs to devise bug-triggering workloads. Among all the bugs we have investigated in the characteristic study, we are currently able to reproduce a total number of 12 bugs from WordPress, MediaWiki, Moodle, and Drupal, and we used all these 12 real-world bugs to evaluate ReqRacer, covering all the four PHP web applications we studied. Based on these 12 bugs, we devise a workload that visits all the pages involved in each bug. Note that the workloads we come up with just visit all the pages one-by-one but not concurrently, and the races are not triggered in the recorded runs with limited concurrency. We also visit some pages that are not essential to the bug in our devised testing workloads.

All our experiments were conducted on a machine with an Intel Core i7-4790 CPU and 16GB memory. The software versions are Apache HTTPD 2.4.93, MySQL 5.6.44, and PHP 5.6.40. Cache is set up according to the requirement of each individual case, and we install the cache component only while evaluating with workloads for WP 15545 and WP 20786.

4.1 Effectiveness Results

Table 4 summarizes the race detection results. Note that a request race could manifest under different workloads, and we are reporting the numbers of unique request races. In total, ReqRacer detects and exposes 17 unique request races that are true and harmful, including 13 unique request races that can explain the 12 known bugs and four unique request races that are previously unknown to us while devising the workloads. ReqRacer also detects eight unique request races that are likely true with application-specific checkers added. As discussed in Section 2.2, the effects of request races that lead to inconsistent views or data corruptions can only be caught with some types of checkers taking application semantics into account. To catch these request races, we came up with applicable-specific checkers based on our understanding of applications while manually checking false positives.

For request races between distinct request handlers, Table 4 shows the numbers of conflicting request pairs and false positives pruned by different strategies. The numbers show that after pruning false positives by RRR dependency, SPK-data dependency, and serializability inference, the majority of false positives are pruned. This shows that these three strategies combined are very effective. The remaining false positives are due to either our conservative analysis on WHERE clauses or failure-free executions after enforcing alternative interleavings, and they are pruned by replay.

For request races between two duplicated instances of a request handler with conflicting SELECT and INSERT queries, two-thirds of the false positives are pruned by serializability checking, and the remainings are pruned by replay. For false positives pruned by replay, there is no application error or database error upon duplicate data insertion. Our evaluation results show that the effect-oriented

approach can effectively find request races between two instances of the same request handler.

We identified likely new bugs, which require application-specific checkers, by checking all the request races pruned by replay and adding application-specific checkers designed based on studied request races resulting in inconsistent and stale views. For example, we added checkers to disallow duplicate comments in WordPress and disallow duplicate class-name aliases in Moodle.

For new bugs and likely new bugs, we have verified that they still exist in the latest version, and we are in the process of reporting and confirming with developers. So far, two bugs have been confirmed, including one that requires an application-specific checker, and others are waiting for responses from developers.

Manual checking of all the request races reported by ReqRacer revealed one false positive, i.e., although we are able to trigger an error under the workload for MDL 43421 by duplicating a request, it is not feasible in practice as the client side will disable the button while waiting for the response. We leave it for future work to address this limitation by incorporating more client-side analysis.

4.2 Efficiency Results

We evaluated the overhead of ReqRacer's recording step by measuring the time between sending a request to and receiving the response from the server, and the overhead is between 2% to 6%. Note that our numbers were measured with both the server and client on the same machine, and we expect the overhead to be smaller in a real-world deployment setting after including network latencies. In our evaluation, the inference step can finish within seconds, and the validation time varies from seconds to several minutes depending on the number of unserializable interleaving to prune. On the other hand, if a developer were to manually conduct stress testing by repeating the sequential workload many times, the bugs are unlikely to be triggered. Even if a bug is triggered once, it is difficult for the developer to know exactly how to trigger it, while ReqRacer can reliably trigger a bug once it is detected.

5 THREATS TO VALIDITY

Characteristics studies are subject to validity problems, and our characteristic-study results need to be taken with the methodology and our selection of web applications in mind. One threat is the likely lack of representativeness of the studied applications and request races. To minimize this threat, we choose popular open-source web applications with a significant user base. Our choice of applications also covers several popular development languages and framework for the server side. The other threat is related to the manual inspection of bug reports. To alleviate this threat, two authors first independently study the collected bugs by thoroughly investigating the resources that are available to us. Once they finish, they cross-check their results and draw a conclusion.

The evaluation of ReqRacer is also subject to validity problems. One threat is the correctness of the implementation and the representativeness of bugs used for evaluation. To minimize this threat, we use all known bugs that we can reproduce from all four PHP web applications included in our study. Another threat is the validity of the newly detected bugs by ReqRacer. We mitigate this threat by reporting newly discovered bugs to developers, and two previously unknown bugs, one of which requires an application-specific checker, have been confirmed. Regarding the general applicability of our proposed technique, our current implementation only handles web applications built on top of LAMP. During our evaluation, ReqRacer was implemented before we reproduced any Moodle and Drupal bugs, and our experience suggests that porting ReqRacer to new LAMP applications will be small. We also believe that our key contributions on modeling happens-before relationships could also apply to other types of web applications, e.g., ORM-based and Node.js-based, and we will leave it for future work.

6 RELATED WORK

Section 1 discussed some related work on several different types of races. We next discuss other related work. Server-side web applications have been the subject of a lot of existing research, and many different techniques have been proposed for improving their reliability [19, 22, 32, 59, 60, 66, 75], but none of them handles the concurrency aspect. Some of these existing techniques handle the input generation problem, and ReqRacer complements these techniques by solving the buggy-interleaving exposing problem.

Techniques focusing on the security aspect of web applications have been proposed, e.g., auditing [44, 70], intrusion detection and recovery [27, 28], and identifying information disclosure [29]. Races are considered severe security vulnerabilities [17], and they can enable concurrency attacks [79]. Our proposed techniques can also help improve the security aspect of web applications by detecting and exposing races.

Similar to detecting client-side races and Node.js races, techniques developed for Android applications also focus on the event-driven nature of the mobile platform [23, 37, 38, 57, 65]. Some Android applications also have a client-server structure, and techniques developed in this paper could also be leveraged to handle races in their server-side applications.

7 CONCLUSION

We present the first, to the best of our knowledge, comprehensive characteristic study of real-world server-side request races in web applications. Our results show that request races are indeed understudied and need more research attention. We expect that future research can follow our study results to provide comprehensive support in combating request races. Guided by these results, we develop ReqRacer, a framework for detecting and exposing request races. Our evaluation shows the effectiveness and efficiency of ReqRacer. Future work can adapt ReqRacer for other types of web applications, repurpose it for production-run deployment, or pursue client/server combined approaches for in-house testing.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback and helpful suggestions. The authors would also like to thank Martin Rinard for commenting on an early version of this paper during his visit to NCSU for giving a talk in the Triangle Computer Science Distinguished Lecturer Series. This project was partly supported by the National Science Foundation under the grant CCF-2008056.

REFERENCES

[1] [n.d.]. Bugzilla. https://github.com/bugzilla/bugzilla.

[2] [n.d.]. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

[3] [n.d.]. Discourse. https://github.com/discourse/discourse.

[4] [n.d.]. DNN, howpublished = "https://github.com/dnnsoftware/Dnn.Platform",.

[5] [n.d.]. Drupal. https://git.drupalcode.org/project/drupal.

[6] [n.d.]. Gitlab. https://about.gitlab.com.

[7] [n.d.]. GoReplay - test your system with real data. https://goreplay.org/.

[8] [n.d.]. MediaWiki. https://github.com/wikimedia/mediawiki.

[9] [n.d.]. Moodle. https://github.com/moodle/moodle.

[10] [n.d.]. Multi-Processing Modules (MPMs) - Apache HTTP Server Version 2.4. https://httpd.apache.org/docs/2.4/mpm.html.

[11] [n.d.]. MySQL 8.0 Reference Manual: 16.2 The MyISAM Storage Engine. https://dev.mysql.com/doc/refman/8.0/en/myisam-storage-engine.html.

[12] [n.d.]. Odoo. https://github.com/odoo/odoo.

[13] [n.d.]. OpenMRS. https://github.com/openmrs/openmrs-core.

[14] [n.d.]. Redmine. https://www.redmine.org/.

[15] [n.d.]. Spree. https://github.com/spree/spree.

[16] [n.d.]. WordPress Trac. https://core.trac.wordpress.org/browser/trunk.

[17] Aaron Hnatiw, Security Compass. [n.d.]. Moving Beyond The OWASP Top 10, Part 1: Race Conditions. https://resources.securitycompass.com/blog/moving-beyond-the-owasp-top-10-part-1-race-conditions-2.

[18] Christoffer Quist Adamsen, Anders Møller, and Frank Tip. 2017. Practical Initialization Race Detection for JavaScript Web Applications. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 66 (Oct. 2017), 22 pages. https://doi.org/10.1145/3133890

[19] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. 2010. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Softw. Eng.* 36, 4 (July 2010), 474–494. https://doi.org/10.1109/TSE.2010.31

[20] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run Software Failure Diagnosis via Hardware Performance Counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/2451116.2451128

[21] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. ACM, New York, NY, USA, 207–222. https://doi.org/10.1145/2541940.2541973

[22] Snigdha Athaiya and Raghavan Komondoor. 2017. Testing and Analysis of Web Applications Using Page Models. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. ACM, New York, NY, USA, 181–191. https://doi.org/10.1145/3092703.3092734

[23] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable Race Detection for Android Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 332–348. https://doi.org/10.1145/2814270.2814303

[24] Marina Billes, Anders Møller, and Michael Pradel. 2017. Systematic Black-box Analysis of Collaborative Web Applications. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 171–184. https://doi.org/10.1145/3062341.3062364

[25] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

[26] Jack Cable. [n.d.]. Race Condition in Redeeming Coupons. https://hackerone.com/reports/157996.

[27] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. 2011. Intrusion Recovery for Database-backed Web Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. ACM, New York, NY, USA, 101–114. https://doi.org/10.1145/2043556.2043567

[28] Ramesh Chandra, Taesoo Kim, and Nickolai Zeldovich. 2013. Asynchronous Intrusion Recovery for Interconnected Web Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 213–227. https://doi.org/10.1145/2517349.2522725

[29] Haogang Chen, Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2014. Identifying Information Disclosure in Web Applications with Retroactive Auditing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, Berkeley, CA, USA, 555–569. http://dl.acm.org/citation.cfm?id=2685048.2685092

[30] Lucian Constantin. [n.d.]. Withdrawal vulnerabilities enabled bitcoin theft from Flexcoin and Poloniex. https://www.pcworld.com/article/2104940/withdrawal-vulnerabilities-enabled-bitcoin-theft-from-flexcoin-and-poloniex.html.

[31] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.Fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 145–160. https://doi.org/10.1145/3064176.3064188

[32] Michael Emmi, Rupak Majumdar, and Koushik Sen. 2007. Dynamic Test Input Generation for Database Applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/1273463.1273484

[33] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[34] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. 2010. A Study of the Internal and External Effects of Concurrency Bugs. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. 221–230. https://doi.org/10.1109/DSN.2010.5544315

[35] Milos Gligoric and Rupak Majumdar. 2013. Model Checking Database Applications. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Rome, Italy) *(TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 549–564. https://doi.org/10.1007/978-3-642-36742-7_40

[36] Egor Homakov. [n.d.]. Hacking Starbucks for unlimited coffee. https://sakurity.com/blog/2015/05/21/starbucks.html.

[37] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race Detection for Event-driven Mobile Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/2594291.2594330

[38] Yongjian Hu and Iulian Neamtiu. 2018. Static Detection of Event-based Races in Android Apps. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 257–270. https://doi.org/10.1145/3173162.3173173

[39] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

[40] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. 2015. Stateless Model Checking of Event-driven Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 57–73. https://doi.org/10.1145/2814270.2814282

[41] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/1993498.1993544

[42] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 241–255. https://doi.org/10.1145/1869459.1869481

[43] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated Concurrency-bug Fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, Berkeley, CA, USA, 221–236. http://dl.acm.org/citation.cfm?id=2387880.2387902

[44] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. 2012. Efficient Patch-based Auditing for Web Application Vulnerabilities. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, Berkeley, CA, USA, 193–206. http://dl.acm.org/citation.cfm?id=2387880.2387899

[45] Simon Koch, Tim Sauer, Martin Johns, and Giancarlo Pellegrino. 2020. Raccoon: Automated Verification of Guarded Race Conditions in Web Applications. ACM.

[46] Oren Laadan, Nicolas Viennot, and Jason Nieh. 2010. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, New York, USA) *(SIGMETRICS '10)*. ACM, New York, NY, USA, 155–166. https://doi.org/10.1145/1811039.1811057

[47] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. 2011. Pervasive Detection of Process Races in Deployed Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*

(Cascais, Portugal) *(SOSP '11)*. ACM, New York, NY, USA, 353–367. https://doi.org/10.1145/2043556.2043589

[48] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

[49] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient Scalable Thread-safety-violation Detection: Finding Thousands of Concurrency Bugs During Testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. ACM, New York, NY, USA, 162–180. https://doi.org/10.1145/3341301.3359638

[50] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 677–691. https://doi.org/10.1145/3037697.3037735

[51] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 419–431. https://doi.org/10.1145/3173162.3177161

[52] Peng Liu, Omer Tripp, and Charles Zhang. 2014. Grail: Context-aware Fixing of Concurrency Bugs. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 318–329. https://doi.org/10.1145/2635868.2635881

[53] Peng Liu and Charles Zhang. 2012. Axis: Automatically Fixing Atomicity Violations Through Solving Control Constraints. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 299–309. http://dl.acm.org/citation.cfm?id=2337223.2337259

[54] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. CloudRaid: Hunting Concurrency Bugs in the Cloud via Log-Mining. ACM.

[55] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[56] Brandon Lucia, Benjamin P. Wood, and Luis Ceze. 2011. Isolating and Understanding Concurrency Errors Using Reconstructed Execution Fragments. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 378–388. https://doi.org/10.1145/1993498.1993543

[57] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race Detection for Android Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 316–325. https://doi.org/10.1145/2594291.2594311

[58] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races That Matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 381–392. https://doi.org/10.1145/2786805.2786820

[59] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2013. Dangling References in Multi-configuration and Dynamic PHP-based Web Applications. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) *(ASE'13)*. IEEE Press, Piscataway, NJ, USA, 399–409. https://doi.org/10.1109/ASE.2013.6693098

[60] Hung Viet Nguyen, Hung Dang Phan, Christian Kästner, and Tien N. Nguyen. 2019. Exploring Output-based Coverage for Testing PHP Web Applications. *Automated Software Engg.* 26, 1 (March 2019), 59–85. https://doi.org/10.1007/s10515-018-0246-5

[61] Roberto Paleari, Davide Marrone, Danilo Bruschi, and Mattia Monga. 2008. On Race Vulnerabilities in Web Applications. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Paris, France) *(DIMVA '08)*. Springer-Verlag, Berlin, Heidelberg, 126–142. https://doi.org/10.1007/978-3-540-70542-0_7

[62] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[63] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 251–262. https://doi.org/10.1145/2254064.2254095

[64] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, New York, NY, USA, 151–166. https://doi.org/10.1145/2509136.2509538

[65] Gholamreza Safi, Arman Shahbazian, William G. J. Halfond, and Nenad Medvidovic. 2015. Detecting Event Anomalies in Event-based Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 25–37. https://doi.org/10.1145/2786805.2786836

[66] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. 2012. Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 277–287. http://dl.acm.org/citation.cfm?id=2337223.2337257

[67] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[68] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 11–21. https://doi.org/10.1145/1375581.1375584

[69] Shudi Shao, Zhengyi Qiu, Xiao Yu, Wei Yang, Guoliang Jin, Tao Xie, and Xintao Wu. 2020. Database-Access Performance Antipatterns in Database-Backed Web Applications. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 58–69. https://doi.org/10.1109/ICSME46990.2020.00016

[70] Cheng Tan, Lingfan Yu, Joshua B. Leners, and Michael Walfish. 2017. The Efficient Server Audit Problem, Deduplicated Re-execution, and the Web. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 546–564. https://doi.org/10.1145/3132747.3132760

[71] Chao Wang, Mahmoud Said, and Aarti Gupta. 2011. Coverage Guided Systematic Concurrency Testing. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. ACM, New York, NY, USA, 221–230. https://doi.org/10.1145/1985793.1985824

[72] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.Js. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 520–531. http://dl.acm.org/citation.cfm?id=3155562.3155628

[73] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How Not to Structure Your Database-Backed Web Applications: A Study of Performance Bugs in the Wild. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 800–810. https://doi.org/10.1145/3180155.3180194

[74] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. 2014. SimRT: An Automated Framework to Support Regression Testing for Data Races. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/2568225.2568294

[75] Xiao Yu and Guoliang Jin. 2018. Dataflow Tunneling: Mining Inter-request Data Dependencies for Request-based Applications. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 586–597. https://doi.org/10.1145/3180155.3180171

[76] Lu Zhang and Chao Wang. 2017. RClassify: Classifying Race Conditions in Web Applications via Deterministic Replay. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 278–288. https://doi.org/10.1109/ICSE.2017.33

[77] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 251–264. https://doi.org/10.1145/1950365.1950395

[78] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. ACM, New York, NY, USA, 179–192. https://doi.org/10.1145/1736020.1736041

[79] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. 2018. OWL: Understanding and Detecting Concurrency Attacks. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 219–230. https://doi.org/10.1109/DSN.2018.00033

[80] Yunhui Zheng and Xiangyu Zhang. 2012. Static Detection of Resource Contention Problems in Server-side Scripts. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 584–594. http://dl.acm.org/citation.cfm?id=2337223.2337292