A Deep Study of the Effects and Fixes of Server-Side Request Races in Web Applications

Zhengyi Qiu
North Carolina State University
Raleigh, North Carolina, USA
zqiu2@ncsu.edu

Shudi Shao
North Carolina State University
Raleigh, North Carolina, USA
sshao@ncsu.edu

Qi Zhao
North Carolina State University
Raleigh, North Carolina, USA
qzhao6@ncsu.edu

Hassan Ali Khan
North Carolina State University
Raleigh, North Carolina, USA
hakhan@ncsu.edu

Xinning Hui
North Carolina State University
Raleigh, North Carolina, USA
xhui@ncsu.edu

Guoliang Jin
North Carolina State University
Raleigh, North Carolina, USA
guoliang_jin@ncsu.edu

ABSTRACT

Server-side web applications are vulnerable to request races. While some previous studies of real-world request races exist, they primarily focus on the root cause of these bugs. To better combat request races in server-side web applications, we need a deep understanding of their characteristics. In this paper, we provide a complementary focus on race effects and fixes with an enlarged set of request races from web applications developed with Object-Relational Mapping (ORM) frameworks. We revisit characterization questions used in previous studies on newly included request races, distinguish the external and internal effects of request races, and relate request-race fixes with concurrency control mechanisms in languages and frameworks for developing server-side web applications.

Our study reveals that: (1) request races from ORM-based web applications share the same characteristics as those from raw-SQL web applications; (2) request races violating application semantics without explicit crashes and error messages externally are common, and latent request races, which only corrupt some shared resource internally but require extra requests to expose the misbehavior, are also common; and (3) various fix strategies other than using synchronization mechanisms are used to fix request races. We expect that our results can help developers better understand request races and guide the design and development of tools for combating request races.

CCS CONCEPTS

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

KEYWORDS

web-application request races, characteristic study, Object-Relational Mapping, external and internal effects, fix strategies

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR '22, May 23–24, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528463

ACM Reference Format:
Zhengyi Qiu, Shudi Shao, Qi Zhao, Hassan Ali Khan, Xinning Hui, and Guoliang Jin. 2022. A Deep Study of the Effects and Fixes of Server-Side Request Races in Web Applications. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524842.3528463

1 INTRODUCTION

Modern web applications, ranging from e-commerce websites to social media platforms, regularly handle a large volume of incoming requests and generate their corresponding responses. The most common architecture used in modern web applications is the 3-tier architecture: i) the presentation tier contains user interface scripts of web applications, ii) the application tier hosts back-end scripts developed using different languages, e.g., Python, Java, Perl, PHP, or Ruby, and these scripts contain *request handlers* to handle HTTP requests from users, and iii) the data tier hosts a database management system for storing and retrieving persistent data.

The presentation tier communicates with the application tier using HTTP requests, which are handled by request-handler scripts on the application layer. The application layer interacts with the data tier either using raw SQL queries, e.g., in the LAMP (Linux, Apache, MySQL, and PHP) stack, or using Object-Relational Mapping (ORM) support from programming frameworks, e.g., Ruby-on-Rails.

Upon receiving HTTP requests, the servers hosting web applications initiate respective request-handler scripts to process the user requests. The initiated request handlers could access shared resources. While handling concurrent requests, request handlers running concurrently could access shared resources in different orders, and *request races* occur when the execution of these concurrent request handlers on the server-side can lead to erroneous behaviors depending on the order of shared-resource accesses [72].

Request races pose threats to the reliability and security [31] of web applications. Some recent web applications failures due to request races have damaged the reputation of famous companies and incurred financial losses, e.g., Starbucks gift-card duplicate balance transfer [46], Flexcoin bankruptcy caused by wallet overdraw [42], and Instacart coupon double redemption [41].

To help the understanding of request races and guide the design of tools for combating request races, we recently studied 157 server-side request races collected from popular open-source web applications that are developed with different languages and frameworks,

i.e., PHP, Perl, Python, C#, Java, Ruby-on-Rails, and Node.js [72]. The results of this study guided the design of a dynamic race detection and inference tool employing novel techniques to model happens-before relationships between HTTP requests handled by web applications. We also pointed out that request races between request handlers are different from races in multi-threaded programs that are used in the 3-tier architecture, e.g., Apache and MySQL, and thus they cannot benefit from the significant research progress on races in multi-threaded programs during the last decade.

Although our previous study [72] is the first to comprehensively explore request races to date in open-source web applications, it still has significant limitations.

First, its coverage of request races in ORM-based web applications is limited, i.e., it only includes 35 request races from one web application developed with Hibernate and three developed with Ruby-on-Rails. It is not clear if request races from a more diverse set of ORM-based web applications share the same characteristics as in raw-SQL web applications.

Secondly, its characterization on the effects of request races does not differentiate external effects, i.e., how errors impact users as failures, and internal effects, i.e., how errors may corrupt internal data and propagate in different layers of the application. In a previous study of concurrency bugs in the multi-threaded MySQL, Fonseca et al. differentiated the external effects and internal effects of the concurrency bugs being studied, the distinction of which led to new findings and insights [45]. However, such a distinction was not made in our previous study of request races [72].

Lastly, its characterization on the fix strategies did not relate to the commonly used concurrency control primitives provided by ORM frameworks. Particularly, Bailis et al. summarized and studied the usage of feral concurrency control primitives of Ruby-on-Rails in web applications [37]. However, it is not clear how common these primitives are used by programmers to fix request races in ORM-based web applications, and it is also not clear whether some fixes in raw-SQL web applications can also be viewed as equivalent to some feral concurrency control primitives from ORM frameworks.

To address the limitations mentioned above, we conduct a study with analyses focusing on the effects and fix strategies of real-world request races on 157 request races that have already been studied by us [72] and 92 request races from ORM-based applications that have not been studied before. Our study primarily focuses on the following three research questions:

- **RQ1:** What are the characteristics of request races in ORM-based web applications? Do they share the same characteristics with raw-SQL web applications on racing-resource types, root-cause patterns, and manifestation conditions?
- **RQ2:** What are the external effects of requests races that affect users using the web applications? What are the internal effects of request races that cause errors in internal data?
- **RQ3:** How often do developers use the feral concurrency control mechanisms provided by ORM frameworks to fix request races? Do developers take similar strategies when fixing request races in raw-SQL web applications?

To answer these research questions, we first pick four popular ORM frameworks in different programming languages, which are Django in Python, Hibernate in Java, Laravel in PHP, and Ruby-on-Rails in Ruby. We then search for open-source real-world web applications on GitHub developed with these ORM frameworks, and we keep those active projects with more than 2K stars. We also include applications from the previous work. To this end, we include 7 web applications using raw-SQL, 7 in Django, 2 in Hibernate, 2 in Laravel, 12 in Ruby-on-Rails, and 11 in Node.js, and we study a total of 249 request races from these applications.

We study this set of request races to answer the three research questions. On **RQ1**, we find that request races from ORM-based web applications share the same characteristics as those from raw-SQL web applications. On **RQ2**, we characterize the external effects of request races into five types, i.e., crash, errors, performance, hang, and semantics. Our results suggest that semantics is the most dominant external effect to users, where the effects are not as easy to notice as crashes or errors. We further categorize request races as latent and non-latent, indicating whether extra requests are needed to make the errors externally visible as failures. We find that 93 of our studied request races are latent, and we further study the internal effects of these latent request races to understand why the semantic assumption is violated. On **RQ3**, we find that only a very small number of studied request races are fixed using feral concurrency control primitives, even in ORM-based web applications.

We expect our results to provide a deep understanding of the characteristics of real-world request races from a diverse set of web applications, which can benefit both application developers and tool developers. For tool developers, our results can guide the design of tools for different purposes, e.g., race detection for ORM-based web applications with our results on **RQ1**, applying the effect-oriented approach to detect a diverse set of request races with our results on **RQ2**, and designing automated request-race fixing tools with our results on **RQ3**.

The remainder of this paper is structured as follows. We first present some background related to our research questions and define some essential terms in Section 2. Then, we introduce the methodology of our study in Section 3. We present our results on the three research questions in Sections 4, 5, and 6, respectively, and we discuss future research opportunities in Section 7. After that, we present related work in Section 8 and discuss threats to validity in Section 9. Finally, we conclude in Section 10. Our dataset can be found at https://github.com/caseqiu213/MSR2022_dataset.

2 BACKGROUND

In this section, we present some background related to each of our research questions and define some necessary terms.

Unserializable interleaving patterns. In our previous study, request races were categorized as either atomicity violations or order violations, and unserializable interleaving patterns in atomicity violations were summarized [72]. In RQ1, we follow the same methodology and label each new bug we study.

Each unserializable pattern consists of three or four operations. Each operation is represented with a single letter, or a group of letters enclosed in a pair of parentheses "()" and separated by '|', indicating multiple possibilities. The first and third operations are from one request. The second and fourth are from the second request and marked with ''.

The pattern of $(\epsilon|R)R'(A|W|D)(A'|W'|D'|R')$ was the most common one from our previous study [72], where ϵ stands for NULL

operations, *R* stands for read operations including database select, file/dir read, and cache read, *A* stands for append operations including database insert, file append, file/dir create, and cache add, *W* stands for write operations including database update, file overwrite, and cache set or replace, and *D* stands for delete operations including database delete, file/dir delete, and cache delete.

Note that unlike unserializable interleaving patterns in multi-threaded programs, where memory operations are usually modeled as read operations or write operations only, it is necessary to separately model append operations and delete operations for shared-resource accesses in request races of web applications [72, 85].

External effects and internal effects. In RQ2, we distinguish the external effects of request races, which are exposed to users, from the internal effects, which impact the internal storage and further propagate across the application code. As the study by Fonseca et al. studied concurrency bugs in MySQL [45], but we will study request races in web applications, our results are different due to the difference of study subjects.

In our previous work [72], database races were categorized as having effects of database error on duplicate data insertion, application error caused by duplicate data, inconsistent or stale view, misleading error message, and program crash or failure; file races have the effects of duplicate file or directory creation, file data corruption, non-existing file or directory error, and misleading error message; cache races have inconsistent or stale view.

However, the effect categorization mixes external effects and internal effects. For example, Moodle 40891 was previously labeled as duplicate directory creation. This is the internal effect happening in the file system, and on the user-side, users are exposed to an error of invalid permission.

ORM feral concurrency control. Modern ORM frameworks provide two types of feral concurrency control primitives on top of transaction and locking [37], and they are:

(1) Application-level validation. Before saving a record to database, the ORM framework runs a set of validations and only saves the record after all validations pass. The validations ensure, for example, the record does not contain a null value for a specific field or the record does not exist and therefore is unique in the database.

(2) Application-level association. The association is a connection between two records, which acts like a foreign key in the database. By checking if a field is present in the database, it is ensured that the association is indeed valid.

Because these concurrency control strategies are in the application level and operate external to the database, they are termed as *feral* concurrency control mechanisms. In RQ3, we explore the relationships between fixes and the feral concurrency control primitives provided by ORM frameworks.

In addition, the previous work [37] also applied the theory of invariant confluence [36] to feral concurrency control mechanisms, where invariant confluence (I-confluence) is a condition that if transactions maintain correct database states regarding an invariant when they execute in isolation, concurrent execution of these transactions can yield another correct state. The authors showed that some feral concurrency control mechanisms, e.g., uniqueness validation, are not I-confluent, which indicates concurrent transaction execution could violate the validation and are still vulnerable to request races.

Table 1: Web applications and numbers of bugs being studied

Application Type	Application	Framework / Language	Number on Server Side
Raw-SQL	DNN [8]	-/C#	4
	Bugzilla [3]	-/Perl	11
	Drupal [9]	-/PHP	31
	MediaWiki [12]	-/PHP	28
	Moodle [13]	-/PHP	15
	WordPress [29]	-/PHP	18
	Odoo [15]	-/Python	2
ORM-based	Oscar [18]	Django/Python	2
	PostHog [20]	Django/Python	3
	Redash [21]	Django/Python	4
	Saleor [24]	Django/Python	4
	Sentry [25]	Django/Python	8
	Weblate [28]	Django/Python	6
	Zulip [30]	Django/Python	8
	BroadLeaf [2]	Hibernate/Java	2
	OpenMRS [16]	Hibernate/Java	3
	October [14]	Laravel/PHP	1
	Pixelfed [19]	Laravel/PHP	2
	AlchemyCMS [1]	Ruby-on-Rails/Ruby	1
	Canvas LMS [4]	Ruby-on-Rails/Ruby	29
	Danbooru [5]	Ruby-on-Rails/Ruby	3
	diaspora* [6]	Ruby-on-Rails/Ruby	2
	Discourse [7]	Ruby-on-Rails/Ruby	9
	Gitlab [10]	Ruby-on-Rails/Ruby	30
	LinuxFr.org [11]	Ruby-on-Rails/Ruby	1
	OpenProject [17]	Ruby-on-Rails/Ruby	2
	Redmine [22]	Ruby-on-Rails/Ruby	2
	ROR [23]	Ruby-on-Rails/Ruby	1
	Sharetribe [26]	Ruby-on-Rails/Ruby	2
	Spree [27]	Ruby-on-Rails/Ruby	4
Node.js-based	[11 applications][a]	Node.js/Javascript	11
		Total	249

[a] We omit the names of the 11 Node.js-based applications each with one bug, and they can be found in our dataset.

3 METHODOLOGY

In this section, we describe our methodology on how we collect and study request races.

Application and Bug Selection. We start from the bugs in our previous study [72] collected from the bug tracking systems of real-world open-source web applications. Although this bug set covers three different types of paradigms and languages, i.e., (1) classical ones that access databases by constructing raw SQL queries directly, (2) those implemented on top of Object-Relational-Mapping (ORM) frameworks, and (3) those implemented on top of the Node.js framework, the coverage on ORM-based web applications is not extensive. Specifically, only 35 request races from three web applications developed with Ruby-on-Rails and two request races from OpenMRS developed with Hibernate for Java are included. We extend the selection of ORM frameworks to include Laravel for PHP and Django for Python. Note that the previous study miscategorized OpenMRS as an application that directly constructs SQL queries access database.

To find more ORM-based web applications, we search for open-source projects on GitHub that are labeled with the four ORM framework names, and we choose web applications with more than 2K stars. We also exclude applications that have been archived or inactive since 2017. After getting the set of applications, we follow a similar methodology as used by previous studies of concurrency bugs in multi-threaded applications [45, 64], performance

Table 2: Overall results on racing resource types, root cause patterns, manifestation conditions, and unserializable patterns for atomicity violations

	raw-SQL	Django	Hibernate	Laravel	Ruby on Rails	Node.js	Total
Number of bugs studied	109	35	5	3	86	11	249
Racing-Resource Types							
Database	69	31	1	2	71	4	178
File	24	2	2	1	6	0	35
Cache only	10	0	0	0	2	0	12
Cache with database	5	1	0	0	0	0	6
Shared-memory data structure	1	1	2	0	7	7	18
Root Cause Patterns and Manifestation Conditions							
Atomicity violation with two instances of the same request handler	73	19	4	3	53	6	158
Atomicity violation with two different request handlers	35	12	1	0	10	2	60
Order violation within one request handler	1	4	0	0	23	3	31
Unserializable Patterns for Atomicity Violation							
$(\epsilon\|R)R'(A\|W\|D)(A'\|W'\|D'\|R')$	91	28	5	3	60	4	191
$A(A'\|R'\|W')A$	2	0	0	0	0	0	2
$W(A'\|R')(W\|A)$	2	1	0	0	1	0	4
$DD'AA'$	4	0	0	0	0	0	4
$(A\|W)(A'\|D'\|W')R$	9	2	0	0	2	4	17

bugs in web applications [76, 79, 80], and non-deadlock concurrency bugs [43, 72, 78] and deadlock bugs in web applications [71] to collect more request races. Specifically, we first search several keywords that are related to races in the bug tracking system and commit history, e.g., "race(s)," "concurrent/concurrency," and "synchronize." After the keyword search, we get 123, 20, 16, and 101 results from web applications developed using Django, Hibernate, Laravel, and Ruby-on-Rails, respectively. We then manually filter out bugs that are obviously not a race, e.g., results with keywords appearing as substrings or used in a different context. We also exclude bugs that are closed without fixes.

To this end, we find 92 new server-side request races from applications based on these four ORM frameworks, and all these bugs have sufficient information for us to understand and are in closed status with committed fixes. Note that we also find 37 new client-side races, which we do not investigate further given our focus on server-side request races. Combining with the 157 server-side request races from the previous study, we study a total of 249 request races, where 109 are in classical web applications, 129 in ORM-based web applications, and 11 in Node.js-based web applications. Table 1 lists the names of the applications and the number of bugs being studied.

Report-Study Methodology. In our study, the analysis focuses on formulating characterization questions and characterizing our collected bugs along with these questions. We start with characterization questions used in our previous study [72]. To come up with new characterization questions, we also leverage the existing studies, i.e., one on the external and internal effects in multi-threaded programs [45] and the other on feral concurrency control [37].

To answer **RQ1**, we first double-check the characterization results of the 157 bugs in our previous study, and we then follow the same methodology to characterize the 92 newly collected bugs. To answer **RQ2**, we get inspiration from the previous study on the internal and external effects of concurrency bugs in MySQL [45]. We study the external effects of request races. Then, we categorize request races as latent and non-latent. For latent request races, we further study their internal effects. To answer **RQ3**, we match fix strategies against the types of concurrency control primitives in programming languages and frameworks for web applications, and we further categorize fix strategies that are not just using existing concurrency control primitives. On **RQ2** and **RQ3**, we study both the 92 newly collected bugs and the 157 previously studied bugs.

With the characterization questions, two authors first individually examine available resources for each bug, including the bug report description, reproducing steps if available, developers' discussion, source code, and intermediate and final patches, to characterize each bug. Then, the two authors cross-check their results to reach an agreement on the characterization results. This process helps to further reduce threats to credibility and validity.

4 RQ1: ROOT CAUSE PATTERNS

Following the same characterization methodology in our previous study [72], we characterize request races newly collected from ORM-based web applications along with three aspects, i.e., racing resource types, root cause patterns, and manifestation conditions.

Overall, we find that request races from ORM-based web applications share similar characteristics as those studied in the previous study on these aspects. Table 2 summarizes our findings among applications using four popular ORM frameworks in different programming languages, together with our previous results of raw-SQL and Node.js applications. Note that our previous study [72] included few request races in ORM-based web applications.

Racing resource types. Databases, files, cache using modules like Redis or Memcached, and shared-memory data structures are still the racing resources we find in applications using ORM frameworks as in our previous study [72]. We note that most bugs racing on cache are found in raw-SQL applications. We also note that most bugs racing on shared-memory data structure are found in applications using Node.js or ORM frameworks, while only one such bug is found in raw-SQL applications. The one shared-memory data structure race in raw-SQL is MediaWiki 28179. MediaWiki uses $_SESSION to store records of uploaded files, and when concurrent file uploading happens, users notice that $_SESSION misses records. This problem happens because the application checks if an entry associated with the file name is null and creates an empty array if it is. Concurrent uploading of the same file will be written to the same entry, and one record is overwritten by the other.

Table 3: Overall results on external effects. Numbers in parentheses indicate the number of latent request races.

	raw-SQL	Django	Hibernate	Laravel	Ruby on Rails	Node.js	Total
Number of bugs studied	109 (43)	35 (15)	5 (1)	3 (2)	86 (27)	11 (5)	249 (93)
Database							
Crash due to unhandled errors	0	13 (0)	1 (0)	0	18 (0)	0	32 (0)
Error from DB	25 (0)	0	0	0	8 (0)	0	33 (0)
Error from application	7 (1)	1 (0)	0	0	5 (0)	0	13 (1)
Performance	1 (0)	0	0	0	0	0	1 (0)
Semantics-Read partially-updated, corrupted data	3(0)	0	0	0	3 (0)	0	6 (0)
Semantics-Return data not matching the current page	0	1 (0)	0	0	3 (0)	0	4 (0)
Semantics-Read after write not returning the just written data	1 (0)	2 (0)	0	0	10 (0)	1 (0)	14 (0)
Semantics-Data being silently overwritten	16 (16)	11 (11)	0	1 (1)	14 (14)	2 (2)	44 (44)
Semantics-Duplicate entries	16 (16)	3 (3)	0	1 (1)	10 (10)	1 (1)	31 (31)
File							
Crash due to unhandled errors	2 (0)	0	0	1 (0)	1 (0)	0	4 (0)
Error from file call	13 (0)	0	0	0	2 (0)	0	15 (0)
Error from application	2 (0)	1 (0)	0	0	0	0	3 (0)
Semantics-Read corrupted file content	6 (4)	1 (1)	2 (1)	0	3 (1)	0	12 (7)
Semantics-Read corrupted directory structure	1 (1)	0	0	0	0	0	1 (1)
Cache only							
Semantics-Page component having wrong data	6 (4)	0	0	0	1 (1)	0	7 (5)
Semantics-Page component having empty data	4 (0)	0	0	0	1 (1)	0	5 (1)
Cache with database							
Semantics-Cache does not load updated DB data	5 (0)	1 (0)	0	0	0	0	6 (0)
Shared-memory data structure							
Crash due to unhandled errors	0	1 (0)	1 (0)	0	1 (0)	2 (0)	5 (0)
Error from application	0	0	1 (0)	0	4 (0)	1 (0)	6 (0)
Hang	0	0	0	0	0	2 (0)	2 (0)
Semantics-Partial or wrong data stored	0	0	0	0	2 (0)	2 (2)	4 (2)
Semantics-Miss records	1 (1)	0	0	0	0	0	1 (1)

Root-cause patterns and manifestation conditions. Similar to our previous study [72], atomicity violation and order violation are still the two root-cause patterns of race bugs in applications using ORM frameworks. For all bugs that manifest as atomicity violations, they involve two request handlers, and they are inter-request races. For all bugs that manifest as order violations, they only require one request handler, and they are intra-request races.

Atomicity violations could either involve two instances of the same request handler or two different request handlers. Some early dynamic request detection techniques [54, 68] can only detect the former case, and more advanced techniques that model happens-before relationships between different request handlers are essential for handling the latter case [72]. In raw-SQL web applications, around one-third of the studied atomicity violations are between two different request handlers, while the ratio is around one-fourth in ORM-based web applications. This shows that it is still important to detect request races between different request handlers in ORM-based web applications, and the happens-before relationship modeling techniques in ReqRacer [72] could be leveraged.

Following the previous study, we also further categorize the patterns of unserializable interleavings in atomicity violations, and the last six rows in Table 2 show the numbers. Our study shows that ORM-based web applications share the same patterns of unserializable interleavings, and similar to raw-SQL applications, most cases fall into pattern one.

All order violation bugs we study are caused by asynchronous execution. Since Node.js and ORM frameworks have built-in support for asynchronous execution, applications based on these frameworks are more prone to order violation bugs. As a result, almost all such bugs are found in those applications, and only one order violation bug is found in raw-SQL applications.

5 RQ2: EFFECTS

In the following sections, we are going to distinguish the external effects of races, which are exposed to users, and the internal effects of races, which impact the internal storage. As we collect more bugs from web applications using ORM frameworks to compare them with races in raw-SQL and Node.js web applications, we will also have a better understanding of requests races in all these three types of web applications. Table 3 shows the overall results, and we discuss the results in detail below.

5.1 External Effects of Races

We define the external effects as the effects that are exposed to the users in a client-side browser.

On the high level, the external effects could be a crash where the execution of a request handler terminates due to unhandled errors and the server hosting the web application generates a generic response corresponding to the errors, which may be difficult for users to understand; errors where web applications catch some underlying errors and generate a response with user-friendly error messages; semantics where the misbehavior is related to specific application logic; a hang where no responses are returned; and performance where the server responds requests slowly.

Each of these types of external effects could manifest differently for different types of racing resources, and some of them can be further categorized into subtypes.

5.1.1 External effects: crashes due to unhandled errors. When a request handler crashes due to unhandled errors, users will see a blank page or generic error messages of internal application details that are difficult to understand, as developers did not foresee such a situation and did not write error-handling code. For request races

resulting in crashes, we usually see words "crash" or "critical failure" being mentioned in the bug report and discussion. In our studied request races, we see that races causing crashes can happen in all three types of web applications.

All the 32 studied request races that lead to crashes and race on database are in ORM-based web applications, but none of them are in raw-SQL web applications. The reason for the above phenomenon is that raw-SQL application developers write their own APIs interacting with the backend database, and the APIs check the return values of queries being invoked, which includes information of whether errors happen. These APIs have the mechanism to catch and handle errors that happen during query execution. In contrast, the ORM frameworks do not catch and handle such errors, even though they provide various database APIs, and developers sometimes miss the requirement of catching and handling errors.

ORM frameworks provide various APIs to issue queries from the applications to the database. However, such APIs are not concurrency-safe. For example, two or more INSERT queries with the same data to insert could be sent at the same time. One could succeed, and others would cause the database to raise duplicate entry errors if there are unique constraints on the table schema. The error is returned to the web application. If the error is not handled, the request handler execution terminates, and the server returns a response indicating internal server errors happened. When such a problem happens in raw-SQL applications, the error is wrapped in the return value to the interface method. The error is caught, and a user-friendly message is rendered and returned to the users instead of error messages with internal application information that is difficult for users to understand.

The crashes on files happen when a concurrent remote request deletes the files that the local request is going to use. The request handler finds that the files do not exist. When such errors are not handled, they lead to crashes.

The crash on shared-memory data structure is similar to the file cases. The data structure may yet to be created or have been deleted when it is accessed. The request handler finds the data structure NULL, which raises an unhandled error leading to a crash.

5.1.2 External effects: explicit errors. To return an explicit error to a client, an error will first be reported, either from the underlying system resource or through application-specific checking, then error-handling code in the web application will be invoked, and error messages are finally returned to the users. The errors could root from database, file system, or application logic.

For explicit database errors, they are caused by concurrent requests violating database constraints. Because the errors are handled properly by the application, they do not cause crashes, and instead, application-rendered error messages are returned to the users. The database error could be caused by duplicate entries in a table with unique constraints, unlocking a table twice, invalid transactions, and entries not being found.

For explicit file system errors, they could be caused by failures while performing various file system operations, e.g., creating or removing a directory, opening a stream, and serializing data from files. Request handlers checking for such failures will return a page telling users the file operation fails and may also put more details about the error into a log.

For application errors, they could be caused by failures during application-specific validation. Examples of such validations include presence validations that check if a data entry exists, null validations that check if an object is valid, and permission checks that check if a particular user is allowed to access some page.

5.1.3 External effects: semantics violations. Request races leading to semantics violations account for 54% of the effects of request races we study. In these bugs, results violating the intended semantics of the web applications are delivered to clients. Next, we discuss them based on the type of racing resources.

For request races involving database only, we categorize the violated semantics into five types.

First, a request may return a page with content violating some semantics assumption, which could happen due to reading partially written, and thus corrupted, data of a racing request. In Bugzilla 292544, a user not in the security group, which means that she cannot access security-related bugs, can access a security bug but the bug disappears upon a page refresh. These two inconsistent results are returned because the query adding a security-related bug and the query updating its flags are not wrapped in a transaction in the request handler. If a user tries to access a list of recently created bugs in between these two queries, some newly added bugs may have their flags not being updated yet and are thus visible to users not in the security group.

Secondly, a request may return a page that is obviously wrong, the expected result of which could be determined by the user based on the current page. In Gitlab 22946, a user cherry-picks a commit by clicking on a link on the current page, but a commit that is different from the one chosen by the user is returned. It is due to a write happening between the time of cherry-picking and the time of result rendering, and a different commit is returned and rendered instead of the one cherry-picked by the user.

Thirdly, if a request includes a read query of some data just being written, users often expect to see the just written data on the returned page, but when the request race happens, the just written data does not show up due to the request race. In Moodle 24678, the user adds a chat message and expects the returned latest chat message to be the just added one. However, if a concurrent request adds a chat message having the same timestamp, the user will see this message instead of the one from herself.

Fourthly, if a request does data modification and just tells users it succeeds without verifying the results, the data could be overwritten by the racing request without being noticed. As a result, any request assuming successful data modification is holding a wrong assumption. In Bugzilla 926952, a user sends a request to rename a milestone. As there are 100 bugs associated with the milestone, this request also changes the milestone field of these bugs. The returned page indicates that all the rename operations succeed. However, a concurrent request overwrites the changes on the milestone field of bug entries back to the old name. Later, when querying bugs with the new milestone name, no bugs are returned. The 100 bugs are lost because they are associated with the old milestone name but no milestone matches the old name.

Finally, after the manifestation of a request race, a user could see duplicate entries shown on the related web page when refetching. Such cases happen because there are no unique constraints

on the database table schema, and duplicate entries are allowed in the database. However, the application semantics indicate the entry should be unique, which unfortunately the application is also not currently checking or enforcing. Because there are no errors, we categorize them as semantics bugs, which violates the unique intention of application logic.

For race bugs involving file only, a user could see corrupted file contents. In Drupal 1377740, a user reports that the file move operation is not atomic and file contents are accessible before they are fully written. In other words, users can view half-written file contents. The file could also have mixed contents from different concurrent requests, and WordPress 31767 is an example of this, where the .htaccess file is corrupted by concurrent file writes. File races could also corrupt the directory. In Moodle 19718, a user tries to delete a forum, but when the race happens, the operation deletes the entire Moodle data directory.

For race bugs involving cache only, a user could get wrong data, e.g., being able to read data modified by another user or even data that should not be accessible. In WordPress 25883, in a multi-site setup with two networks A and B, when a user requests for a record named "testmetakey" from network A, she may get the record from network B if there is an entry with the same name. This bug is caused by not differentiating cache keys with the same name but from different sites. A user could also get a page with empty data for some components. In Drupal 2879512, a user may get a response with the path aliases field of a node being empty. It happens because the path alias cache key was deleted by a concurrent, racing request after the local request checks the existence of the cache key but before the local request updates the value.

For race bugs involving both database and cache, a user could find that the contents on the returned page do not show the modified data just sent by the request, and this applies to all our studied request races involving both database and cache. In WordPress 20786, when a user attaches an image to a post, the content of the cache key is cleared first, then the database is updated, and finally the cache loads the updated data from the database. If a concurrent remote request accessing the same cache key happens in between the cache clear and database update, the cache key is found to be empty and stale data is loaded to the cache. When the attaching-image request accesses the cache key after a database update, it is a cache hit, and stale data is returned to the user.

For race bugs involving shared-memory data structure only, a user could see the data structure stores partial or wrong data. In Gitlab 63507, a user reports a record with nil Kubernetes token value persisting in the storage. Such request races could also lead to missing records. In MediaWiki 28179, a user uploads multiple files concurrently. When the race happens, only a subset of file records will be shown in the file list.

5.1.4 External effects: performance related. We label one request race from one raw-SQL web application as performance and two as hang in Node.js applications. The difference is that in the performance case, the request waits a long time to get responded to, while in the hang cases, the request never gets responded to.

The request race being labeled as performance is WordPress 2088. When the race happens, repeated pingback operations are issued, and the server is slowed down quickly due to this.

One request race being labeled as hang is fiware-pep-steelskin 269. The hang happens because the first request adds itself to listen for certain events and then waits for the event, while the second request clears the event listeners in between these two operations. As a result, the first request never receives the event and appears to be blocked forever. The other request race being labeled as hang is from deepstream.io. The first request stops the server. While handling the first request, all clients are disconnected, and the handler will fire a stop event. In between these two operations, the second request attempts to connect the server and succeeds, which prevents the stop event from being fired. As a result, the server process hangs and needs to be manually killed.

5.2 Request Races with Latent Effects

We consider a request race to have latent effects where the concurrent requests that cause the erroneous states differ from the request that exposes the external effects of the bug to the client-side users. In other words, the failure is revealed by a subsequent request that does not happen concurrently with the racing request pairs. We consider a request race non-latent if its misbehavior is exposed by concurrent request pair in the request race.

Request races with external effects of crash, performance, and hang are all non-latent. For request races having error messages, they are non-latent if errors are first raised by the database or due to file operation failures. However, some of the errors raised by application validation are latent, meaning a third request will be needed to expose the failure. When this is the case, the application validation is done in the third request but not in the two requests involved in the race. For example, in Moodle 59854, a user can be enrolled twice in the same forum, which means that there can be duplicate entries in the forum_subscription table. This violates the application logic, where one user should be enrolled once in the forum, and the user identity should appear once in the forum_subscription table. There are no error messages shown when enrolling the user for the second time. However, when fetching the user subscription list of the forum, the application checks if every user identity is unique in the list, and an application error is raised if the validation fails. In this bug, the error message is exposed by a subsequent fetching list request, which does not have to be concurrent with the user enrollment requests. As a result, this is a latent bug with an error message effect.

Request races with semantics issues could be latent or non-latent.

For races on database, if the request has a read operation after a write or the request fetches data only, an experienced user can determine if the returned page matches the expected results, and such bugs are non-latent. This applies to the first three subcategories of request races leading to semantics violations we list in Table 3.

The remaining two subcategories are latent. Request races could break data silently if unwanted data overwritten is not noticed. If such silent data overwriting is not handled properly, it could lead to failures that are difficult to understand and diagnose and, sometimes, great data loss. The previously discussed Bugzilla 926952 is such a request race.

For bugs having duplicate entries with no database errors, additional application semantics are needed to determine if such duplicate entries are allowed.

Table 4: Latent bugs: racing-resource types for internal effects, root cause patterns, manifestation conditions, and unserializable patterns for atomicity violation

	raw-SQL	Django	Hibernate	Laravel	Ruby on Rails	Node.js	Total
Number of bugs studied	43	15	1	2	27	5	93
Racing-Resource Types							
Database single-table	32	12	0	2	23	3	72
Database multi-table	1	2	0	0	1	0	4
File	5	1	1	0	1	0	8
Cache only	4	0	0	0	2	0	6
Shared-memory data structure	1	0	0	0	0	2	3
Root Cause Patterns and Manifestation Conditions							
Atomicity violation with two instances of the same request handler	31	11	1	2	25	4	74
Atomicity violation with two different request handlers	11	4	0	0	2	0	17
Order violation within one request handler	1	0	0	0	0	1	2
Unserializable Patterns for Atomicity Violation							
$(\epsilon\|R)R'(A\|W\|D)(A'\|W'\|D'\|R')$	42	15	1	2	27	4	91

For races on file, if a request only modifies a file or directory without reading the content later, request races involving such requests are latent, as another request that read the file or directory content is needed to detect the problem.

For races on cache only, if the involved request handler modifies the value of a cache key, the response usually indicates if the modification succeeds without returning contents of the cached data. This type is latent, and users need to check the cached data to verify if misbehaviors happen with an extra request.

For races on cache and database, these are all non-latent because one request involved in such request races first clears the cache, then loads data from database to cache, and finally returns the values in cache to users. When the request races are triggered, users can immediately notice if misbehavior happens by investigating the contents on the returned page, as the returned data will be stale.

For races on shared-memory data structure, when the involved requests do modification without reading the data structure, request races are latent as users need to send another request to determine if something is wrong.

Overall, we find that latent request races are more challenging to handle. They may either need extra requests to expose the external effects or need application-specific semantics knowledge, and thus are more critical.

5.3 Internal Effects of Latent Request Races

To better handle latent request races and catch them early, we analyze the latent bugs in more detail. Specifically, we pay close attention to how the data in the persistent storage are erroneous to achieve a better understanding of what tools can be developed to detect such bugs before they are exposed to users.

Table 4 shows the overall results. Latent request races could involve only a single type of racing resource. We find latent request races involving all types of racing resources, i.e., database, file, cache, and shared-memory data structures.

For latent request races on database, the internal effects include wrong data in a single table and inconsistent data between tables having application logic correlation.

The majority of latent races involve a single table. Internally, as they exhibit the atomicity violation pattern, where a remote write, append, or delete operation happens between two local operations, the second operation will be affected by the remote operation, but the internal effect requires a subsequent request fetching the just changed data to make the misbehavior externally visible.

Web applications could use multiple tables to store data that are correlated. In WordPress, post data are stored in a post table, and data correlated to the post are stored in a post_meta table, which is also used for recovering a trashed post. However, when the race happens, the data between these two tables can be inconsistent. For example, a comment could not be backed up in the post_meta table and disappears when recovering the post, to which the comment belongs. Note that such correlation is inferred from web application logic, but no validations or constraints enforce the correlation in the application code or the database schema.

For latent request races on files, the internal effects are corrupted file or directory, where the file is corrupted with incomplete or wrong data or the directory structure is broken. For latent request races on cache, the internal effects could be overwritten cache data or cache key not being updated due to key deletion. For latent request races on shared-memory data structures, the internal effects could be shared data structures containing wrong data.

In Table 4, we also show the root cause patterns, manifestation conditions, and unserializable patterns for atomicity violations, from which we can see that some request races violating order assumptions also require a subsequent fetching request to expose the misbehaviors, and are thus latent.

6 RQ3: FIXES

With the context of feral concurrency control presented in Section 2, fix strategies can be categorized as using database transactions, various locking strategies, validations, and semantics changes. We do not find request races fixed by using the application-level association primitive. Table 5 shows the overall results. Below, we detail each fix strategy.

Transactions. Developers fix the races by wrapping queries into a transaction to prevent database from being in a state where mixed contents from concurrent requests exist. In MediaWiki 129462, developers wrap a select and insert query into a transaction to avoid duplicate entries.

More interestingly, request races could also sometimes be fixed by removing certain transactions, and we find four such cases in

Table 5: Overall results on fix strategies

	raw-SQL	Django	Hibernate	Laravel	Ruby on Rails	Node.js	Total
Number of bugs studied	109	35	5	3	86	11	249
Database							
Transaction: Add transaction	4	1	0	1	1	0	7
Transaction: Remove transaction	0	0	0	0	4	0	4
Locking: Database lock	11	9	0	0	15	2	37
Locking: Distributed lock	0	0	0	0	2	0	2
Validation: Presence	1	0	0	0	1	0	2
Validation: Uniqueness	2	0	0	0	1	0	3
Validation: Custom constraint	13	4	0	1	2	0	20
Semantics: Catch and handle errors on app side	5	9	1	0	13	0	28
Semantics: Add database-side uniqueness constraint	7	0	0	0	5	1	13
Semantics: Handle conflict in query	6	0	0	0	3	0	9
Semantics: Rescue database data on race	2	2	0	0	1	0	5
Semantics: Frontend	4	0	0	0	1	0	5
Other semantics	14	3	0	0	10	0	27
Order enforcement	0	3	0	0	12	1	16
File							
Locking: File lock	5	2	0	0	0	0	7
Locking: Custom lock based on database entries	1	0	0	0	0	0	1
Validation: Presence	5	0	0	1	0	0	6
Semantics: Catch and handle errors on app side	6	0	0	0	2	0	8
Semantics: Write to unique file or dir	6	0	1	0	3	0	10
Semantics: Read in fixed chunk size	1	0	0	0	0	0	1
Order enforcement	0	0	0	0	1	0	1
Cache only							
Validation: Custom constraint	4	0	0	0	0	0	4
Semantics: Check return value of cache fetch	1	0	0	0	0	0	1
Semantics: Write to unique cache key	3	0	0	0	0	0	3
Semantics: Use correct API	0	0	0	0	2	0	2
Semantics: Use new storage	2	0	0	0	0	0	2
Cache with database							
Validation: Custom constraint	2	0	0	0	0	0	2
Semantics: Postpone cache operation after database update	3	1	0	0	0	0	4
Shared-memory data structure							
Locking: Thread lock	0	0	2	0	1	1	4
Validation: Custom constraint	0	0	1	0	1	5	7
Semantics: Catch and handle errors on app side	0	0	0	0	1	0	1
Semantics: Change storage to database	1	0	0	0	0	0	1
Semantics: Include unique value in condition	0	0	0	0	1	0	1
Order enforcement	0	1	0	0	3	1	5

our studied request races. In Discourse 3854, users report that they are not being notified after a staff member responds to their posts. Users can only find out the response by checking the messages page in their profiles. When a staff member responds to a post, a message entry is created in the database. Right after the message creation, an asynchronous job is dispatched to alert users of the new response message. However, the message creation operation is wrapped in a transaction. When the asynchronous job fetches the just created message, the transaction is not committed yet, and the job fetches nothing. As a result, users are not notified of the response from staff. By removing the transaction, the asynchronous job can read the necessary data to notify the right users about the staff's response.

Locking. For request races on database, we see developers using two types of locks to fix races, which are database locks and distributed locks.

Regarding the database lock, it could be the generic table locks in the database. In Bugzilla 292544, developers lock the whole table to prevent remote concurrent reads get in-progress local write contents. It could be the generic row locks provided by database. In MediaWiki 51581, developers add "FOR UPDATE" in the query string, which locks the selected entry and avoids concurrent modification. It could also be a lock implemented using entries in the database. In Drupal 1182754, developers create an entry named as advagg_insert_bundle_db in the semaphore table to store the lock versions and avoid duplicate entries in database.

Distributed lock is a cross-process lock. It synchronizes executions between requests and can be implemented using Redis. In Discourse 8819, distributed locks are used to avoid inconsistent column contents while processing a post.

For races on file, we see file locks and custom locks implemented using database entries. In WordPress 31767, file locks are used to prevent concurrent file writing, which avoids corrupting the .htaccess file. In WordPress 34878, developers implement a lock by entries in the wp_option table to prevent critical failures that happen because of concurrent file deletion during core updates.

For races on shared-memory data structure, thread locks are used. In spree 6719, developers use a random number generator using thread locks to avoid writing entries with the same random number to the database.

We do not see any request races on cache-related request races being fixed with locks.

Constraint validations. For races on database, we find that developers use presence validation, uniqueness validation, and custom constraint validation to fix races. Note that, the concept of constraint validation is borrowed from the context of feral concurrency control primitives provided by ORM frameworks. We extend it to the scope of raw-SQL web applications, and we also extend it to operations not related to database accesses.

Presence validation checks if a given entry is null or empty. In Gitlab 6881, the fix checks if a given project is null and skips removing records of the associated artifacts if the check finds that the project is not null.

Uniqueness validation checks if a given entry is unique by comparing it with other records in the database. In other words, a SELECT query is sent checking if the unique field value of the entry appears in the database. Then, the application would issue an INSERT or an UPDATE query based on the SELECT query result. In Moodle 46651, developers use the uniqueness validation to avoid duplicate entries, where a SELECT query is sent to check if the record to be inserted exists. If it does not, an INSERT query is issued; if it does, an UPDATE query updating the existing record with current parameters is issued.

Custom constraint validation could be some checks related to specific application logic, and without the check, races can happen and cause misbehaviors. The check could be on a local variable holding content from a file, a query result from the database, or values generated during the application execution. In Bugzilla 391073, the race is fixed by checking if _throw_error function is called within eval(), and the fix skips unlocking the table if the condition is true. In WordPress 11073, the race is fixed by reading and checking the post status, and the fix skips adding the comment if the post has a trash status.

For races on file, we find that developers use presence validation to fix races. In MediaWiki 51391, the failure of a mkdir operation could be caused by a concurrent request doing the same operation. The fix adds a presence checking, which checks if the directory already exists. If the mkdir fails because the directory has been created by a concurrent request, no warnings or errors should be passed to the users causing panic. This fix pattern is also used in Drupal, Moodle, and October. We do not find custom constraint validation for races on file.

For races involving cache, we only find developers use custom constraint validation but no uniqueness or presence validation, and the check is also specific to application logic. In MediaWiki 94491, the fix adds a check to see if a user rename status is finished in the database but not in the cache. The cache will be purged if the condition is true and loads the latest data from the database. In Drupal 2879512, the fix checks if the cache entry was created but now is deleted when executing the code line and skips doing certain operations if the condition is true, which avoids corrupting cache with incomplete data.

Our results show that even in ORM-based web applications, the number of request races fixed by adding constraint validation is small. For the two types of constraint validations that can be found in our studied request races, the I-confluence of the presence constraint depends and the uniqueness constraint is not I-confluent. Therefore, while adding these constraint validations can reduce the racing window, the request races may still not be completed fixed. For custom constraint validations, their I-confluence needs to be investigated case-by-case, and we leave it for future work.

Semantics changes. Partly due to the small number of request races fixed by using constraint validations, 49% of request races are fixed involving semantics changes. There are various fix strategies that change the application logic, and we are going to discuss some major approaches.

Catch and handle errors on the application side is a commonly seen fix strategy. By properly handling the errors, users will not panic by a crash that only provides difficult-to-understand information and instead with informative, case-specific error messages.

On request races involving cache only, the fix strategy "check return value of cache fetch" is similar. In WordPress 15545, the request handler updates cache and then reads and returns the updated data to users. However, a concurrent cache key deletion could make the cache read in the first request handler return an empty array. Developers think valid information, even stale, is better than an empty array. The fix first stores stale cache data into a variable. If the cache read gets an empty array, stale data is returned.

For races on database, developers would add uniqueness constraints to the table schema, which prevents duplicate entries on the database side. They would further add "ON CONFLICT" in the query string, so the database updates fields of duplicate entries instead of raising an error. In order to ensure data integrity, developers refetch the just updated data to see if any semantic assumptions are violated. In WordPress 22023, developers refetch the just inserted entry and delete duplicates if more than one is returned. Developers also fix races by disabling a button in the frontend to prevent concurrent requests from happening. With these subcategories of semantics changes, we label 27 request races involving database as other semantics changes, as the semantics changes in these fixes are very diverse and application specific.

For races on file, developers would write contents to unique temp files and then use atomic rename to avoid corrupted files. One request race is fixed by reading a file in a fixed chunk size. In Moodle 41291, the file size was stored in a local variable, and the file size was used to read file content. However, a concurrent request could modify the file and changes the file size. As a result, truncated or corrupted file content could be read. The fix reads the file in a fixed chunk size until reaching the end of the file.

For races on cache, other than checking the return value of cache fetching, developers could also fix request races by directing writes to unique cache keys, using correct cache APIs, or using new storage. The first two cases are intuitive, and we give an example of the third case. In MediaWiki 105105, the race is fixed by using WANObjectCache, which broadcasts cache updates to all sites so that no site will have stale data to be returned to users.

For races on cache and database, developers could enforce the cache update to happen after database updates, making cache load the latest data after the database is updated.

For races on shared-memory data structure, change storage to database and include unique value in condition are used to fix two request races.

Order enforcement. For order violation races, developers fix them by enforcing the order, which can be achieved by moving the code snippet, registering the function with the event happening at the proper timing, adding delay, and using synchronized operations.

7 DISCUSSION

Next, we discuss future research opportunities enabled by our results in combating request races.

Pattern-based detection. In our studied bugs, ORM-based web applications have many races manifesting as order violations, while raw-SQL web applications have only one. While techniques proposed for detecting races manifesting as atomicity violations in raw-SQL web applications [72] can be adapted for ORM-based web applications, we need new techniques for detecting races manifesting as order violations in ORM-based web applications.

API-usage-guided detection. As discussed in Section 5.1.1, ORM frameworks provide APIs that are not always concurrency-safe, and programmers are expected to handle errors returned by these APIs. However, developers of ORM-based web applications could have a misunderstanding of these APIs or fail to handle errors returned by these APIs. Future research could detect API misusages and missing error handling in ORM-based web applications.

Effect-oriented detection. Our results show that more than half of the collected bugs do not have an explicit error message alerting users that misbehavior happens. Future work can use an invariant-based method to detect such bugs.

Bug fixing. Our results show that developers often change application logic to fix races, and this is true even for web applications built on top of ORM frameworks that provide feral concurrency-control mechanisms. Future work on race fixing should be aware of this finding. If generic synchronization mechanisms are used for automated fixing, the evaluation may need to compare automated patches with manual ones.

8 RELATED WORK

A lot of research efforts have been spent on races and concurrency bugs in multi-threaded programs. Researchers have conducted thorough empirical characteristic studies [45, 64] and the study results guide the development of tools for various purposes, e.g., bug detection [39, 40, 44, 48, 65, 74, 83, 84], program testing [47, 58, 69, 75, 77, 81], failure diagnosis [34, 35, 51, 53, 66], and fixing [50, 52, 61, 62]. Researchers have also proposed techniques targeting process races on the operating-system level [55] and distributed concurrency bugs in distributed and cloud systems [56, 57, 59, 60, 63]. As argued in our previous study [72], these techniques target races inside the system layer, and they are not effective for request races, which are races in the web applications hosted on top of the system layer.

Specific to the external and internal effects of concurrency bugs, the previous study [45] focuses on concurrency bugs in MySQL, which is a multi-threaded program. While MySQL is commonly used as the backend database in web applications, concurrency bugs in MySQL are different from request races in web applications, which follows from the same arguments made in our previous study [72]. Nevertheless, we follow their methodology and get inspiration from their study to conduct ours.

Also note that our study focuses on request races on the server-side, and thus we have a different focus compared with recent work that focuses on client-side race detection [33, 38, 49, 67, 70, 73, 82] and fixing [32].

On the studies of concurrency bugs in web applications, while the one by us [72] is the most comprehensive and the state-of-the-art, there are also two related studies on concurrency bugs in applications developed on top of the Node.js framework [43, 78]. However, the number of request races in web applications covered by these two studies is only 11 [72], and the remaining concurrency bugs are not on the server-side or from applications that are not web applications. While we focus on request races in this work, a study focusing on deadlocks has also been conducted [71].

9 THREATS TO VALIDITY

In this section, we describe several potential validity threats our study may be subject to and our ways to address them.

(1) The applications in our study cannot represent all real-world ORM-based web applications. To minimize this threat, we choose popular web applications with more than 2K stars and exclude those inactive ones since 2017. Our application selection covers popular ORM frameworks written in various programming languages.

(2) We may miss relevant bug reports while searching for races in the bug tracking system. We mitigate this threat by using keyword search in both bug descriptions and discussion as well as the commit history. We also include all request races from the previous study. To this end, our numbers of request races from raw-SQL and ORM-based web applications are close.

(3) We inspect bug reports manually, which may be subject to human errors while characterizing request races. To alleviate this threat, we first double-check the characterization results in our previous study. Then, two authors first independently investigate the bugs, including those that have been previously studied and those newly collected from ORM-based web applications, with all available resources, including bug description and discussion, patches, and source code. Once they finish, they cross-check their results and reach a consensus.

10 CONCLUSION

Request races impose a great challenge to the reliability and security of web applications. To better understand request races, we complement the state-of-the-art study by augmenting more request races from ORM-based web applications in different programming languages. We investigate the external effects of request races and characterize those lead to semantics violations. We further divide request races into latent and non-latent and study the internal effects of latent ones. We summarize various fix strategies for raw-SQL and ORM-based web applications in the context of feral concurrency control primitives available in ORM frameworks. We expect our results to be useful to guide the design and development of future tools for combating request races.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers, from both the program committee and the shadow program committee, for their valuable feedback and helpful suggestions. This project was partly supported by the National Science Foundation under the grant CCF-2008056.

REFERENCES

[1] AlchemyCMS - AlchemyCMS is a Rails CMS engine. https://github.com/AlchemyCMS/alchemy_cms.

[2] Broadleaf Commerce - Enterprise eCommerce framework based on Spring. https://github.com/BroadleafCommerce/BroadleafCommerce.

[3] Bugzilla. https://bugzilla.mozilla.org/.

[4] Canvas LMS - The Open LMS by Instructure, Inc. https://github.com/instructure/canvas-lms.

[5] Danbooru - A taggable image board written in Rails. https://github.com/danbooru/danbooru.

[6] diaspora* - A privacy-aware, distributed, open source social network. https://github.com/diaspora/diaspora.

[7] Discourse - A platform for community discussion. Free, open, simple. https://github.com/discourse/discourse.

[8] DNN Platform Issue Tracker. https://dnntracker.atlassian.net.

[9] Drupal. https://git.drupalcode.org/project/drupal.

[10] Gitlab. https://about.gitlab.com.

[11] LinuxFr.org - A French-speaking website about Free software / hardware / culture / stuff. https://github.com/linuxfrorg/linuxfr.org.

[12] MediaWiki - The collaborative editing software that runs Wikipedia. https://github.com/wikimedia/mediawiki.

[13] Moodle Tracker. https://tracker.moodle.org/.

[14] October - Self-hosted CMS platform based on the Laravel PHP Framework. https://github.com/octobercms/october.

[15] Odoo - A suite of web based open source business apps. https://github.com/odoo/odoo.

[16] OpenMRS - Medical Record System. http://openmrs.org.

[17] OpenProject - OpenProject is the leading open source project management software. https://github.com/opf/openproject.

[18] Oscar - Domain-driven e-commerce for Django. https://github.com/django-oscar/django-oscar.

[19] Pixelfed - Photo Sharing. For Everyone. https://github.com/pixelfed/pixelfed.

[20] PostHog - PostHog provides open-source product analytics that you can self-host. https://github.com/PostHog/posthog.

[21] Redash - Make Your Company Data Driven. Connect to any data source, easily visualize, dashboard and share your data. https://github.com/getredash/redash.

[22] Redmine. https://www.redmine.org/.

[23] ROR Ecommerce - Ruby on Rails Ecommerce platform, perfect for your small business solution. https://github.com/drhenner/ror_ecommerce.

[24] Saleor Commerce - A modular, high performance, headless e-commerce platform built with Python, GraphQL, Django, and React. https://github.com/saleor/saleor.

[25] Sentry - Sentry is cross-platform application monitoring, with a focus on error reporting. https://github.com/getsentry/sentry.

[26] Sharetribe - Sharetribe Go is a source available marketplace software, also available as a hosted, no-code SaaS product. https://github.com/sharetribe/sharetribe.

[27] Spree - Open Source headless multi-language/multi-currency/multi-store eCommerce platform. https://github.com/spree/spree.

[28] Weblate - Web based localization tool with tight version control integration. https://github.com/WeblateOrg/weblate.

[29] WordPress Trac. https://core.trac.wordpress.org/.

[30] Zulip - Zulip server and web app — powerful open source team chat. https://github.com/zulip/zulip.

[31] Aaron Hnatiw, Security Compass. 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.

[32] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing Event Race Errors by Controlling Nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 289–299. https://doi.org/10.1109/ICSE.2017.34

[33] 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

[34] 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

[35] 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

[36] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (nov 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[37] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1327–1342. https://doi.org/10.1145/2723372.2737784

[38] 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

[39] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. 2014. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 28–39. https://doi.org/10.1145/2594291.2594323

[40] 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

[41] Jack Cable. 2016. Race Condition in Redeeming Coupons. https://hackerone.com/reports/157996.

[42] Lucian Constantin. 2014. 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.

[43] 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

[44] 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

[45] 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

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

[47] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. 2012. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. ACM, New York, NY, USA, 210–220. https://doi.org/10.1145/2338965.2336779

[48] 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

[49] 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

[50] 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

[51] 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

[52] 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

[53] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 344–360. https://doi.org/10.1145/2815400.2815412

[54] Simon Koch, Tim Sauer, Martin Johns, and Giancarlo Pellegrino. 2020. Raccoon: Automated Verification of Guarded Race Conditions in Web Applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. ACM, New York, NY, USA, 1678–1687. https://doi.org/10.1145/3341105.3373855

[55] 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

[56] 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

[57] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S. Gunawi, and Shan Lu. 2019. DFix: Automatically Fixing Timing Bugs in Distributed Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 994–1009. https://doi.org/10.1145/3314221.3314620

[58] 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

[59] 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

[60] 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

[61] 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

[62] 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

[63] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. CloudRaid: Hunting Concurrency Bugs in the Cloud via Log-Mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/3236024.3236071

[64] 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)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[65] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. 2008. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 277–288. https://doi.org/10.1109/ISCA.2008.4

[66] 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

[67] 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

[68] 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

[69] 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

[70] 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

[71] Zhengyi Qiu, Shudi Shao, Qi Zhao, and Guoliang Jin. 2021. A Characteristic Study of Deadlocks in Database-Backed Web Applications. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Wuhan, China, 510–521. https://doi.org/10.1109/ISSRE52982.2021.00059

[72] 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 Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 842–854. https://doi.org/10.1145/3468264.3468594

[73] 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

[74] 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

[75] 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

[76] 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

[77] 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

[78] 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, 520–531. https://doi.org/10.1109/ASE.2017.8115663

[79] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (Singapore, Singapore) *(CIKM '17)*. ACM, New York, NY, USA, 1299–1308. https://doi.org/10.1145/3132847.3132954

[80] 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)*. Association for Computing Machinery, New York, NY, USA, 800–810. https://doi.org/10.1145/3180155.3180194

[81] 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

[82] 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

[83] 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

[84] 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

[85] 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