ABSTRACT

QIU, ZHENGYI. Understanding and Combatting Request Races in Database-backed Web Applications. (Under the direction of Guoliang Jin).

Modern web applications, such as e-commerce platforms and social networking, center around the database for retrieving and storing persistent data. These web applications are inevitably vulnerable to request races because the server hosting the web applications needs to process a large volume of concurrent requests. Recent high-profile incidents illustrate that request races can lead to serious service corruption, severe security vulnerabilities, and huge financial losses. In this proposal, we present study results to understand server-side request races. We then use these results to guide the design of detecting and fixing tools.

Firstly, we conduct a study to understand server-side request races. We investigate bugs collected from open source database-backed web applications, including ones using raw-SQL queries from developers and ones built upon Object-Relational Mapping (ORM) frameworks. We summarize characteristics essential for detection, including the types of racing resources, root cause patterns, and manifestation conditions. We further enlarge the set of request races from web applications developed upon ORM frameworks to make our study more representative, and focus on more aspects, including race effects and fixing strategies, to make it more comprehensive. We revisit characterization questions used in previous studies on newly included request races, distinguish the effects exposed to users and internal effects that cause errors in internal data, as well as if the effects are latent or not. We relate request race fixes with concurrency control mechanisms in languages and frameworks for developing server-side web applications. Our study results are expected to guide the design of tools for detecting request races for ORM web applications, applying the effect-oriented approach to detect a diverse set of request races, and designing automated request-race fixing tools.

Secondly, we build a tool REQRACER for detecting scope-based request races in raw-SQL web applications in PHP. REQRACER design is guided by the study results such as two requests are enough to trigger a request race and atomicity violation account for the majority of request races. 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 potentially concurrent requests. 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.

Lastly, we relate request races with request handler idempotence. Being idempotent means the specific operation does not cause unintended side effects when executed multiple times, and idempotence is a crucial property of request handlers in database-backed web applications where requests can be sent multiple times due to various reasons such as user actions. Developers usually use validations to achieve request handler idempotence, but request races could invalidate the constraints that have been verified in validations under concurrent executions, which makes validation insufficient to enforce request handler idempotence. Request races that invalidate idempotence enforcement account for around 50% of our collected bugs in ORM-based web applications, and they are the focus of my third work. We study how request races invalidate request handler idempotence enforcement and summarize guidance to help developers build idempotent request handlers in a concurrent environment by studying the fixes of related request races. We further extract rules from request races in ORM-based web applications and build a static tool for detecting request races, with a special focus on idempotence enforcement.

© Copyright 2023 by Zhengyi Qiu

All Rights Reserved

Understanding and Combatting Request Races in Database-backed Web Applications

by
Zhengyi Qiu

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2023

APPROVED BY:

_______________________	_______________________
Xiaohui (Helen) Gu	Alexandros Kapravelos
_______________________	_______________________
Xu Liu	Huiyang Zhou

____
Guoliang Jin
Chair of Advisory Committee

BIOGRAPHY

Zhengyi Qiu, a native of Jiangsu, China, spent his formative years in the same province. Prior to his enrollment at Hohai University, he resided with his parents, who provided a supportive environment for his educational pursuits.

In 2014, Zhengyi embarked on a new chapter in his academic journey by joining North Carolina State University (NCSU). Two years later, in 2016, he successfully obtained his Master's degree in Computer Engineering, further solidifying his expertise in the field.

With a thirst for knowledge and a drive to excel, Zhengyi took the next step in his educational path by enrolling in the Ph.D. program in Computer Science at NCSU in 2017. Under the expert guidance of Professor Dr. Guoliang Jin, Zhengyi embarked on an intellectually stimulating and challenging research journey, dedicating himself to the pursuit of academic excellence in his chosen field.

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to my advisor, Dr. Guoliang Jin, for his unwavering patience, encouragement, guidance, and continuous support throughout my extensive Ph.D. journey.. I am also deeply grateful to my esteemed committee members, Dr. Xiaohui (Helen) Gu, Dr. Alexandros Kapravelos, Dr. Xu Liu, and Dr. Huiyang Zhou, for their valuable insights and contributions to my research. Special appreciation goes to Dr. George Rouskas, whose assistance and suggestions proved invaluable in overcoming challenges and completing my program, even when extensions were necessary.

I would like to extend my sincere thanks to Qi Zhao and Shudi Shao for their invaluable assistance in validating bug study results, as well as their invaluable contributions to building and improving the tool artifacts. Their insightful suggestions during the spinning process were immensely helpful.

Furthermore, I would like to express my heartfelt appreciation to my parents for their unwavering encouragement and support throughout this entire journey.

Finally, I would like to extend my gratitude to all the staff members of the Computer Science Department for their contributions and assistance.

TABLE OF CONTENTS

List of Tables vi

List of Figures vii

Chapter 1 INTRODUCTION 1
1.1 Request Races 2
1.2 Request Races Breaking Request Handler Idempotence 3
1.3 Limitations of Existing Work 6
1.4 Contributions 9
1.5 Outline 13

Chapter 2 Terms and Background 14
2.1 Unserializable Interleaving Patterns 14
2.2 External Effects and Internal Effects 15
2.3 ORM Feral Concurrency Control 15
2.4 ACID Transaction 16
2.5 Weak Isolation and Anomalies 17

Chapter 3 Understanding Server-Side Request Races 22
3.1 Methodology of Characteristic Study 22
3.2 Characteristic-Study Results 23
3.3 RQs for Request Race Deep Understanding 28
3.4 Methodology of Deep Study on Effects and Fixes 29
3.5 RQ1: Root Cause Patterns 31
3.6 RQ2: Effects 32
3.6.1 External Effects of Races 32
3.6.2 Request Races with Latent Effects 37
3.6.3 Internal Effects of Latent Request Races 39
3.7 RQ3: Fixes 40
3.8 Threats to Validity 44

Chapter 4 Detecting Server-side Scope-based Request Races 54
4.1 ReqRacer Design 54
4.1.1 Illustrating Example 55
4.1.2 Tracing and Request-Race Inference 58
4.1.3 Replay-Based Validation 61
4.2 Implementation and Evaluation of ReqRacer 62
4.2.1 Effectiveness Results 63
4.2.2 Efficiency Results 65
4.3 Threats to Validity 65

Chapter 5 Understanding and Detecting Request Races Breaking Idempotence . **68**
5.1 RQs for Understanding Request Races Breaking Idempotence 68
5.2 Methodology 70
5.3 Understanding Request Races and Request Handler Idempotence 72
5.3.1 RQ1: Request Races and Request Handler Idempotence 72
5.3.2 RQ2: Reasons Why Request Races Invalidate Idempotence Enforcements 75
5.3.3 RQ3: Fixings for Enforcing Idempotence 79
5.4 Static Detection for Request Races 82
5.4.1 Extract Rules from Request Races 82
5.4.2 Implementation 86
5.4.3 Checking Real-world Web Applications 87
5.4.4 Limitations 89
5.5 Discussion 91
5.5.1 Suggestions of ORM Framework API Development 91
5.5.2 Suggestions to Web Application Developers 92

Chapter 6 Related Work **93**

Chapter 7 Conclusion **97**

References **100**

LIST OF TABLES

Table 2.1 ANSI-SQL Isolation Level Definition with Three Original Phenomena 18
Table 2.2 Consistency Levels and Locking ANSI-SQL Isolation Levels 19

Table 3.1 Web applications and numbers of bugs being studied in deep study on request race effects and fixes 45
Table 3.2 Web applications and numbers of bugs being studied in request race characteristic study 46
Table 3.3 The patterns of unserializable interleavings and their numbers in our studied request races 46
Table 3.4 Overall characteristic-study results 47
Table 3.5 Overall results on racing resource types, root cause patterns, manifestation conditions, and unserializable patterns for AVs 49
Table 3.6 Overall results on external effects. Numbers in parentheses indicate the number of latent request races 50
Table 3.7 Latent bugs: racing-resource types for internal effects, root cause patterns, manifestation conditions, and unserializable patterns for atomicity violation 51
Table 3.8 Overall results on fix strategies 52

Table 4.1 Overall evaluation results. “Reqs” shows the number of requests in the workload. “#Acc.” represents the number of database or cache accesses in the trace. “Racing Resc.” represents the type of racing resources. “Con. 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. 67

Table 5.1 Determine if Request Races Happen in Request Handlers that Need to Be Idempotence 72
Table 5.2 Business Logic of Request Handlers that Need to be Idempotent 73
Table 5.3 Idempotence Enforcement Methods Used Before Fixing 75
Table 5.4 Proposed Patches for Idempotence Enforcement 79
Table 5.5 Exception Handling Logic 79
Table 5.6 API *save* and *create* in Request Races 83
Table 5.7 API *save* and *create* with Request Handler Logic 83
Table 5.8 Tool Detection Results 88

LIST OF FIGURES

Figure 1.1 A high-level illustration of the Instacart incident 2
Figure 1.2 Posthog 386, a scope-based request race 4
Figure 1.3 Openproject 9265, a level-based request race 5

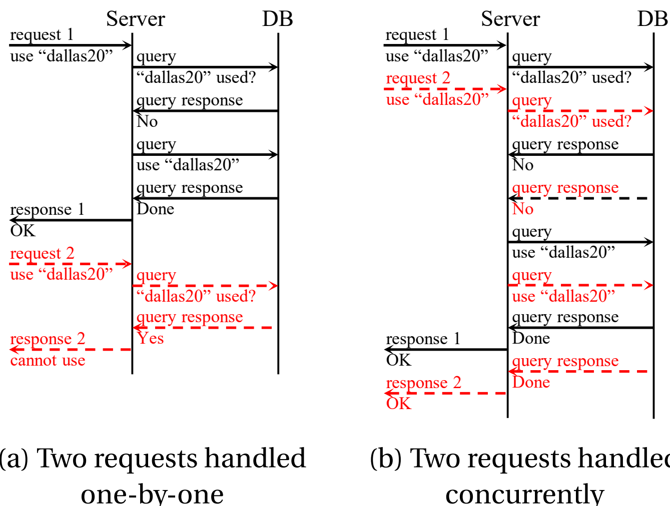
Figure 4.1 The architecture of REQRACER 55
Figure 4.2 An example of dependency-graph construction 56

CHAPTER

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. 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*.

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.



(a) Two requests handled one-by-one

(b) Two requests handled concurrently

Figure 1.1: A high-level illustration of the Instacart incident

1.1 Request Races

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 [51], Starbucks gift-card duplicate balance transfer [66], and Flexcoin bankruptcy caused by wallet overdraw [55].

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 [51]. Figure 1.1 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 1.1a shows the case when these two coupon-redemption requests are sent synchronously, i.e., sending one request after receiving the response from 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 1.1b, 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 1.1b, 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.

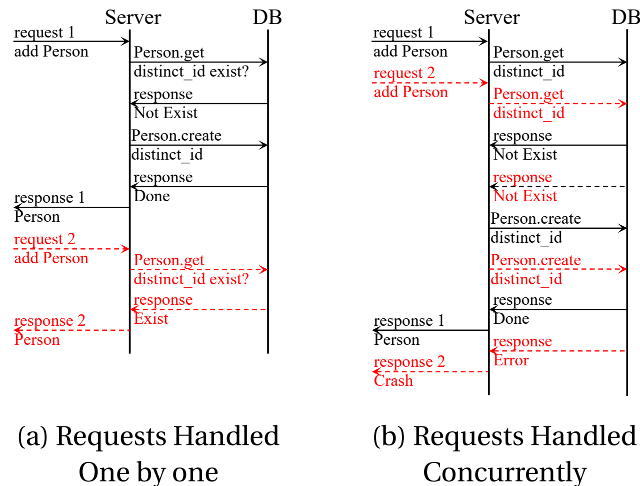
As exemplified by these incidents, request races can lead to serious service corruption, severe security vulnerabilities, and huge financial losses. Two recent trends make request races an emerging threat to the reliability and security of web applications. First, the development of cloud platforms greatly eases the deployment of web applications, and the number of web applications increases. Secondly, getting access to websites 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.

1.2 Request Races Breaking Request Handler Idempotence

Idempotence is a property of programming that ensures the result of an operation is the same, no matter how many times the operation is performed. In addition, idempotence makes it easier to manage errors and recover from failures. If an operation is idempotent and it fails, it can be safely re-executed without affecting other parts of the system. It is widely acknowledged that idempotence is crucial to distributed systems [95].

Modern web applications regularly handle a large number of requests from users. Upon receiving HTTP requests, the servers hosting these web applications invoke corresponding request handlers to process these user requests. Idempotence is crucial for web application request handlers. Requests may be sent multiple times due to network issues, user operations, or other factors. For example, consider a user who creates a new account. If the user clicks the create button multiple times, an idempotent request handler can ensure that the account with the same username is created once, even if multiple identical requests are received by the server.

Furthermore, idempotence helps ensure the reliability and consistency of web applications. In a distributed or multi-node environment, requests may be processed by different



(a) Requests Handled One by one

(b) Requests Handled Concurrently

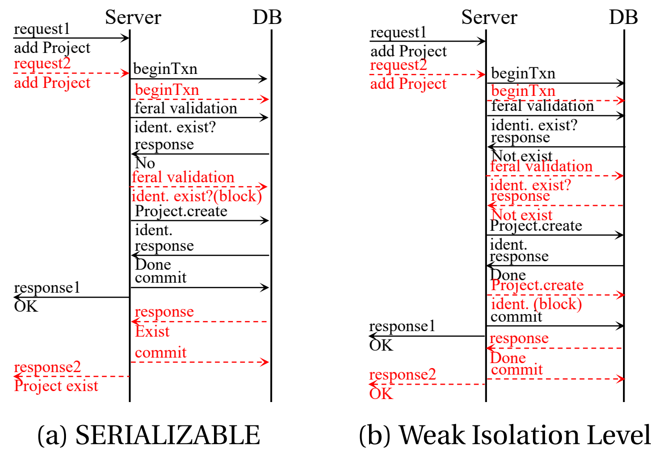
Figure 1.2: Posthog 386, a scope-based request race

servers or application instances. Idempotent request handlers ensure that the system remains consistent because if one request fails, it can be retried without affecting the system and will receive the same results.

To enforce request handler idempotence various methods are adopted by developers. One common method used is validation, including app-side validation and feral uniqueness validation. The business logic, such as adding a new account, can only be performed after particular validations pass. App-side validations can be implemented by checking certain conditions on the returned results from a SELECT query. Feral uniqueness validations are easier to use in ORM web applications because they are automatically triggered by the frameworks after declared. Validations help developers to achieve request handler idempotence under sequential executions. For example, when a user sends a second request to add the same account after sending the first one, the request handler runs a validation check on the results from a SELECT query and sees the username already exists, which was added by the first request, it then skips the following business logic to add an account.

Unfortunately, under concurrent executions request races invalidate the constraints that have been verified in validations under concurrent executions, which makes validation insufficient to enforce request handler idempotence.

Validations and the queries sent by ORM APIs following it can be freely interleaved when not wrapped into a transaction and a scope-based request race happens. Figure 1.2 shows an example of a scope-based request race from Posthog. In the validation, the *Person.get*



(a) SERIALIZABLE (b) Weak Isolation Level

Figure 1.3: Openproject 9265, a level-based request race

API is called and it sends a query to check if a person record with the distinct_id exists in the database. If validation passes, the *Person.create* API is called to insert a person record with the distinct_id. Figure 1.2a shows the interleaving when requests are sent sequentially. Only one request passes validation and only one person record with the distinct_id exists in the database. However, when requests are sent concurrently, as the interleaving in Figure 1.2b shown, both requests pass the validation and execute the following *Person.create* API. The database schema has unique constraints on the distinct_id column of the person table, and an error is returned, which causes an application crash. Because the application status changes from running to crashing when executing multiple concurrent requests, the validation fails to enforce request handler idempotence.

When validation and ORM APIs that send business logic queries are wrapped into a transaction, a level-based request race can happen based on the configured isolation levels and invalidate idempotence enforcement. Figure 1.3 shows an example of a level-based request race from Openproject. The feral validation sends a query to check if a project with the identifier exists in the database, and a project is added if the identifier does not exist. These operations are wrapped into a transaction automatically by the ORM framework. In Figure 1.3a, when the database isolation level is configured to SERIALIZABLE, the database coordinates read operations using locks. Only one request passes feral validation and the other one finds project record with the identifier exists in the database. Figure 1.3b shows the scenario when the database isolation level is weaker than SERIALIZABLE. Read operations are no longer coordinated by the database using locks, and both requests pass the

feral validation and execute the following *Project.create*. Even though the database coordinates write queries, it does not prevent duplicate records from being inserted. Because the database has duplicate records when concurrent requests execute, which is different from a single request, the feral validation fails to enforce request handler idempotence.

1.3 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 [61, 90] and proposed various techniques for different purposes, e.g., bug detection [50, 60, 71, 107, 127, 128], program testing [84, 99, 108, 113, 121], failure diagnosis [41, 42, 74, 92], and fixing [73, 75, 87, 88]. 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 1.1 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 1.1. 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 [81] and *distributed concurrency bugs* in distributed and cloud systems [82, 85, 86, 89] 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 [56, 114]. 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 [37, 48, 72, 94, 100, 104, 125] 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 [79, 98], static [130], and model-checking [63] approaches. The dynamic and static approaches [79, 98, 130] only detect request races between two instances of the *same* request handler, and one of them [79] 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 1.1, 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 [63] alleviates the need to model happens-before relationships between distinct request handlers by constructing concurrent test cases manually, limiting the applicability of the approach.

No existing studies define idempotence for request handlers. The general definition of idempotence requires "the results to be the same". Developers usually render various error messages if any user inputs are invalid, which makes "the results to be different". For request handlers, we care more about if the system states stay the same, and such rendered error messages do not change system states and we need to adapt the idempotence definition for request handlers.

No existing studies shed light on the relationship between request races and request handler idempotence enforcement. Our previous work [103], a comprehensive request race study, shows two-thirds of request races happen between instances of the same request handler. It seems to indicate that request races that happen between instances of the same request handler and request handler idempotence enforcement are related because both have a flavor of "applying the same operations multiple times", but it remains unclear how common request races that invalidate request handler idempotence are in these request races.

Existing studies [103, 116] do not provide guidance on how to build idempotent request handlers. They summarize strategies to fix request races, such as using locks and exception handling, but it remains unclear which strategies are suitable for fixing request races that invalidate idempotence enforcement.

Existing tools [79, 102, 116] cannot efficiently detect request races that invalidate idempotence enforcement in ORM web applications. They analyze query syntax to find conflict database accesses. However, the syntax can be very complex and these tools cannot cover all syntax. ORM APIs are suitable for static detection because they abstract away complex query syntax and are easier to analyze. In this work, we will build a static tool to help developers unearth request races that invalidate idempotence enforcement in ORM web applications.

1.4 Contributions

In this section, we present the primary contributions of our first two works and introduce our third work.

Understanding Server-side Request Races. 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 reveals many general findings on racing-resource types, manifestation conditions, and root-cause patterns, which are useful and necessary to guide the development of detection tools for 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 further improve our study by addressing the following aspects to make it more representative and comprehensive.

First, our first study 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 web applications share the same characteristics as in raw-SQL web applications. In this study, we enlarge the bug set by including bugs from ORM frameworks written in different programming languages.

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 [61]. In this study, we use it as a roadmap to follow and make such a distinction for request races.

Lastly, the characterization on the fix strategies in our first study 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 [45]. In this study, we focus on how common these primitives are used by programmers to fix request races in ORM web applications, and whether some fixes in raw-SQL web applications can be viewed as equivalent to some feral concurrency control primitives from ORM frameworks.

To address the aspects 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 [102] and 92 request races from ORM-based applications that have not been studied before.

Our study reveals that: (1) request races from ORM 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; (3) various fix strategies other than using synchronization mechanisms are used to fix request races.

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 web applications, applying the effect-oriented approach to detect a diverse set of request races, and designing automated request-race fixing tools.

Detecting Server-side Scope-based Request Races. 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 [81], 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.

Understanding and Detecting Request Races Breaking Idempotence. We check 85 bugs racing on the database records in web applications built upon ORM frameworks from our previous study [103] and filter out bugs that race on other resources and happen in raw SQL or Node.js web applications.

We define request handler idempotence as the following. A request handler can be considered idempotent if subsequent executions with the same input parameters no longer have side effects on resources or system states after the first execution of the request handler. Side effects on resources include duplicate records generated in the database and specific logic violations such as voucher multiple redemptions, and side effects on system states include application crashes. Responses containing developer rendered error messages are not considered side effects because they do not change resources or system states.

Our preliminary results show that around 50% of the collected request races invalidate idempotence enforcement. *Firstly*, we further do a deep study on these races. We look into the manifestation condition of included request races and the request handler business logic with fixing patches, which has the expected functionality from developers, to determine if a request handler needs to be idempotent.

Secondly, we document what idempotence enforcement methods are used before fixing. We find that app-side validations, feral uniqueness validations, and database unique constraints are the used methods. We further look into how these methods are implemented and whether they work to achieve request handler idempotence under sequential executions. We reason about how request races invalidate these methods under concurrent executions.

Thirdly, we investigate how different proposed patches address request races that invalidate request handler idempotence enforcement. We find out that exception handling is the most commonly used to achieve idempotence under concurrent executions. We further investigate how these exception handling blocks are implemented to guide developers on building idempotent request handlers. We discuss whether other proposed patches can help enforce request handler idempotence under concurrent executions.

We extract rules from request races in ORM-based web applications and build a static tool for detecting request races, with a special focus on idempotence enforcement. We first identify ORM APIs that are frequently involved in request races that invalidate request

handler idempotence enforcements. We focus on ORM APIs because they simplify database access by abstracting away the complexity of writing raw SQL queries and they are suitable for static detection. We examine sites where these ORM APIs are used and extract rules about what usage of APIs may lead to request races. We extract two rules from 23 request races that lead to duplicate entries and application crashes and one rule from 5 request races that lead to incorrect column value update.

To measure the applicability of the extracted rules we implemented a static request race detection tool. The tool finds a total of 447 new bugs in the latest versions of 19 ORM web applications, including 10 built on Ruby on Rails, eight on Django, and one on Laravel. In these new bugs, 304 will lead to application crashes, 83 will lead to wrong values during concurrent updates, and 60 will lead to duplicate entries in the database.

1.5 Outline

The remainder of the dissertation proceeds as follows. Chapter 3 presents our study results of request races in database-backed web applications. Chapter 4 presents the design and evaluation of ReqRacer. Chapter 5 presents our research for understanding request races that break request handler idempotence enforcement and a static tool for detecting request races with a special focus on idempotence enforcement.

CHAPTER

2

TERMS AND BACKGROUND

In this chapter, we present some terms used in the following chapters and background in web applications, ORM frameworks, and databases.

2.1 Unserializable Interleaving Patterns

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 study [102], 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 [102, 130].

2.2 External Effects and Internal Effects

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 [61], but we will study request races in web applications, our results are different due to the difference in study subjects.

In the first piece of work, 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.

2.3 ORM Feral Concurrency Control

Modern ORM frameworks provide two types of feral concurrency control primitives on top of transaction and locking [45], and they are:

(1) Application-level validation. Before saving a record to the 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 addition, the previous work [45] also applied the theory of invariant confluence [44] 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.

2.4 ACID Transaction

In database-backed applications, a transaction is a sequence of read and write operations interacting with shared data storage. One transaction executes atomically as if it is a single operation, even though it may include multiple reads and writes to the database. Developers can simplify the design of applications involving transactions, which are treated as a single code snippet for data access.

By definition, a database transaction must have four properties: *Atomicity*, *Consistency*, *Isolation*, and *Durability*, and they are often referred to as *ACID properties* for transactions. We summarize the definition of the ACID properties as follows:

- *Atomicity*. The *acid* property is also known as the 'All or nothing rule'. The transaction is successfully executed if all of its operations are completed without any errors; otherwise, the transaction is discarded, and the database is restored to the state before executing the transaction. There is no midway where a transaction executes partially. If all operations are executed, the transaction is *committed*; if the transaction is discarded, the transaction is aborted.

- *Consistency*. The *consistency* property guarantees that integrity constraints are maintained before and after executing the transaction. The database transit from one legitimate state to another, where no constraint violations happen. If the data violate any of the constraints, the whole transaction will fail and be aborted.

- *Isolation*. The *isolation* property guarantees that transactions running concurrently cannot see each other's intermediate results. The isolation guarantee is also referred

to as SERIALIZABLE. However, databases provide more isolation levels other than SERIALIZABLE to essentially for the best suit for developer's design. These isolation levels are weaker than SERIALIZABLE and have better performance. However, they allow concurrent transactions to observe intermediate results to some degree.

- *Durability.* The *durability* property guarantees that the effects of one transaction are stored in the database once it completes execution. These effects persist even if a system failure occurs, such as a power outage or node crash. The records should be recovered once the system comes back to normal.

2.5 Weak Isolation and Anomalies

Among all isolation levels, SERIALIZABLE, under which concurrent execution of multiple transactions is equivalent to serial execution, frees developers from the task of reasoning about the concurrent interleaving of transactions. However, SERIALIZABLE comes with drawbacks in performance. When developers seek better performance, they use weak isolations but the situation is not ideal.

Terminology. A *transaction* can be treated as a list of read and write operations, and each operation is associated with the data it accesses. It transforms the database from one consistent state to another. A *history* models the interleaved execution of a set of transactions running concurrently as a linear ordering of their operations. A *data conflict* happens if two operations are from distinct transactions on the same data and one of them is a write.

ANSI Isolation Definition. In ANSI-SQL standard [117], four isolation levels are introduced. From strongest to weakest, they are SERIALIZABLE, REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED. SERIALIZABLE follows the definition that transactions running concurrently could be treated as running one by one in a certain order. The other three isolation levels come with undesired consequences, *Phenomena*, which are *Dirty Read*, *Non-repeatable Read*, and *Phantom*.

Berenson et al. described these Phenomena [46], and we summarize as follows:

- *P1 (Dirty Read):* Transaction T 1 modifies a data item. Another transaction T2 then reads that data item before T1 performs a COMMIT or ROLLBACK. If T1 then performs a ROLLBACK, T2 has read a data item that was never committed and so never really existed, which is obviously incorrect.

Table 2.1: ANSI-SQL Isolation Level Definition with Three Original Phenomena

Isolation Level	Dirty Read(P1)	Fuzzy Read (P2)	Phantom (P3)
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
Locking REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible

- *P2 (Non-Repeatable or Fuzzy Read):* Transaction T1 reads a data item x. Another transaction T2 then modifies or deletes x and commits. If T1 then attempts to repeat reading x, the returned value of x is different from the previous read, or even worse, x does not exist in the database.

- *P3 (Phantom):* Transaction T1 reads a set of data items satisfying some predicate C. Transaction T2 then creates data items that satisfy the same predicate C and commits. If T1 then rereads with C, it gets a set of data items different from the first read.

All Phenomena listed above will not happen in SERIALIZABLE but could happen in a weak isolation level. The four isolation levels defined in ANSI-SQL Standard can be expressed to prohibit one or more Phenomena and summarized in table 2.1. Specifically, READ UNCOMMITTED allows all Phenomena to happen; READ COMMITTED prohibits Dirty Read (P_1); REPEATABLE READ prohibits both Dirty Read (P_1) and Fuzzy Read (P_2); and SERIALIZABLE prohibits all three Phenomena ($P_1 - P_3$).

Critiques on ANSI Isolation Definition. Berenson et al. criticized that the definition in ANSI-SQL Standard was ambiguous because it may lead to a common misconception that prohibiting these three Phenomena implies SERIALIZABLE. These Phenomena can be interpreted in either a loose or strict way, where a loose interpretation excludes a larger set of execution histories than a strict interpretation. However, even with loose interpretation, forbidding these Phenomena does not guarantee true serializability. To solve the ambiguity of isolation level definition in ANSI-SQL Standard, Berenson et al. proposed various isolation levels, which are defined by prohibiting the following Phenomena:

- *P0:* $w_1[x]...w_2[x]...((c_1 \text{ or } a_1))$

- *P1:* $w_1[x]...r_2[x]...((c_1 \text{ or } a_1))$

- *P2:* $r_1[x]...w_2[x]...((c_1 \text{ or } a_1))$

Table 2.2: Consistency Levels and Locking ANSI-SQL Isolation Levels

Locking Isolation Level	Proscribed Pheno.	Read Locks	Write Locks
Degree 0	None	None	Short write locks
Degree 1 = Locking READ UNCOMMITTED	P0	None	Long write locks
Degree 2 = Locking READ COMMITTED	P0, P1	Short read locks	Long write locks
Locking REPEATABLE READ	P0, P1, P2	Long data-item read locks, short phantom read locks	Long write locks
Degree 3 = Locking SERIALIZABLE	P0, P1, P2, P3	Long read locks	long write locks

- *P3:* $r_1[P]...w_2[y \text{ in } P]...((c_1 \text{ or } a_1))$

Proscribing P_0 is to disallow transaction T_2 to update the data item x if an uncommitted transaction T_1 is updating x, which is equivalent to saying that long-term Write locks are held on data items. Proscribing P_1 is to disallow transaction T_2 to read the same uncommitted data from transaction T_1, which is equivalent to transaction T_1 acquiring a long write lock and transaction T_2 acquiring at least a short-term read-lock. Proscribing P_2 is to prevent transaction T_2 from reading modification from an uncommitted transaction T_1, and requires usages of long-term read and write locks. P_3 deals with queries based on predicates. Proscribing P_3 requires that a transaction T_2 cannot modify a predicate P by inserting, updating, or deleting a row such that its modification changes the result of a query executed by an uncommitted transaction T_1 based on predicate P. To avoid this situation, T_1 acquires a long phantom read-lock on predicate P.

As shown in table 2.2, the READ UNCOMMITTED level proscribes P0; the READ COMMITTED level proscribes P0 and P1; the REPEATABLE READ level proscribes P0-P2; and SERIALIZABLE proscribes P0-P3.

Graph Approach to ANSI Isolation Definition. The four isolation levels are precisely described by avoiding four Phenomena mentioned in table 2.2. However, Adya et al. reviewed these isolation definitions and criticized the preventative approach, i.e. using the locking approach, is overly restrictive. The preventative approach rules out optimistic and multi-version implementations, resulting in disallowing legal histories permitted by optimistic and multi-version implementations. An example is that Phenomena P_0 is allowed to happen in optimistic concurrency control because uncommitted transactions can access and modify the local copy of the same object concurrently. SERIALIZABLE can be provided

by allowing some transactions to commit and abort the other transactions that modify the same objects. Therefore, proscribing P_0 is overly restrictive to database implementations adopting optimistic concurrency control.

The reason for the over-restrictiveness is that the isolation levels defined by Berenson et al. were based on lock schemes. Instead of using locking schemes, Adya et al. proposed an implementation-independent approach to describe isolation levels. The approach represents concurrent transaction histories based on graphs [38]. In this approach, each tuple or row in the database is modeled as an object. A transaction reads and writes objects and indicates a total order in which these operations occur. Unlike the previous definition, each object is associated with a version number. For example, when a transaction T_i writes an object x, it creates a new version m of x, which is $x_{i.m}$. When a transaction reads x, it reads a version of x created by other transactions previously. A history contains two parts: a partial order of events that reflects the operations of transactions, including read, write, commit and abort, and a total order on committed versions of each object. When transaction T_i commits, each version of x_i created by T_i becomes a part of the committed state, and it is termed as T_i installs x_i.

The graph approach represents committed transactions and their relations in a directed graph called Direct Serialization Graph (DSG). It reflects a specific history H running a set of transactions, e.g. DSG(H), , in which each transaction is a node, and edges are the conflicts between two different transactions from the transaction set.

The conflicts are classified into three categories based on operations and orders of source and destination.

- Direct Read Dependency (wr edge). A transaction T_j directly read-depends on another transaction T_i if either of the following two conditions is satisfied:
  - Item-read-dependency: T_i installs some version of object x_i and T_j reads x_i.
  - Predicate-read-dependency: T_j performs a read based on some predicate, which includes the object x, and the installation of a new version by T_i changes the predicate results.

  If transaction T_i and T_j have a direct read-dependency in history H, an edge $T_i \xrightarrow{\text{wr}} T_j$ will be added to the DSG(H).

- Direct Anti-Dependency (rw edge). A transaction T_j directly anti-depends on another transaction T_i if either of the following two conditions is satisfied:

- Item-anti-dependency: T_i reads som object version x_k and T_j installs the next version of object x.
- Predicate-anti-dependency: T_i performs a read based on some predicate, and T_j installs a newer version of object x, which changes the matching results of predicate.

If transaction T_i and T_j have a direct anti-dependency in history H, an edge $T_i \xrightarrow{\text{rw}} T_j$ will be added to the DSG(H).

- Direct Write Dependency (ww edge). A transaction T_j directly write-depends on another transaction T_i if T_i installs a version x_i and T_j installs the next version of the same object x.

  If transaction T_i and T_j have a direct write-dependency in history H, an edge $T_i \xrightarrow{\text{ww}} T_j$ will be added to the DSG(H).

T_j conflicts with T_i, or T_j depends on T_i if there are one or more direct dependency edges from T_i to T_j which shows $T_j \longleftarrow T_j$ in the DSG. Based on the DSG, Adya et al. proposed new Phenomena to define isolation levels that are implementation-independent. The new definition of isolation levels was based on proscribing certain cycles in the DSG:

- READ UNCOMMITTED proscribes any transaction histories that the corresponding DSG consists of cycles composed of only write dependency edges.
- READ COMMITTED proscribes any transaction histories that the corresponding DSG consists of cycles formed only with read dependency edges and write dependency edges.
- REPEATABLE READ proscribes any transaction histories that the corresponding DSG consists of cycles that only contain read dependency edges, write dependency edges and item anti-dependency edges.
- SERIALIZABLE proscribes any transaction histories that the corresponding DSG consists of any cycles.

CHAPTER

3 UNDERSTANDING SERVER-SIDE REQUEST RACES

In this chapter, we show the characteristics of bugs collected from raw-SQL and ORM web applications. Our dataset can be found at `https://github.com/caseqiu213/MSR2022_dataset`.

3.1 Methodology of Characteristic Study

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

In our characteristic study, we follow the same methodology as taken by existing studies on concurrency bugs in multi-threaded applications [90], races in Node.js applications [56, 114], and performance bugs in web applications [109, 119].

Table 3.2 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 [109, 119], 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.

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 understudied 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.

3.2 Characteristic-Study Results

Table 3.4 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 [81] 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 [79, 98] 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 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 [130] 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 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 3.3 summarizes the results. The first four patterns are mostly the same as the ones by Zheng and Zhang [130] 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 $(A|W)(A'|R'|W'|D')(A|R)$,

[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.

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 [130] 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 [130] 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 [68, 127, 128] 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 [16], 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 [17], 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 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 [56, 114], 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.

3.3 RQs for Request Race Deep Understanding

To have a deep understanding of request race effects and fixes, we focus on the following three research questions to have a deep understanding of request races:

- **RQ1:** What are the characteristics of request races in ORM 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 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 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 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 fixing tools with our results on **RQ3**.

3.4 Methodology of Deep Study on Effects and Fixes

Application and Bug Selection. We start from the bugs in our previous study [102] 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 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 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 [61, 90], performance bugs in web applications [110, 118, 120], and non-deadlock concurrency bugs [57, 102, 115] and deadlock bugs in web applications [101] 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 web applications, and 11 in Node.js-based web applications. Table 3.1 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 [102]. 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 [61] and the other on feral concurrency control [45].

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 [61]. 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.

3.5 RQ1: Root Cause Patterns

Following the same characterization methodology in our previous study [102], we characterize request races newly collected from ORM 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 web applications share similar characteristics as those studied in the previous study on these aspects. Table 3.5 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 [102] included few request races in ORM 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 [102]. 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.

Root-cause patterns and manifestation conditions. Similar to our previous study [102], 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 [79, 98] 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 [102]. 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 web applications. This shows that it is still important to detect request races between different request handlers in ORM web applications, and the happens-before relationship modeling techniques in ReqRacer [102] 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 3.5 show the numbers. Our study shows that ORM 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.

3.6 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 3.6 shows the overall results, and we discuss the results in detail below.

3.6.1 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.

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 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.

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.

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.

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.

3.6.2 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 3.6.

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.

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.

3.6.3 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 3.7 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 3.7, 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.

3.7 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 3.8 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 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 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.

3.8 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. 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 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 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 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.

Table 3.1: Web applications and numbers of bugs being studied in deep study on request race effects and fixes

Application Type	Application	Framework / Language	Number on Server Side
Raw-SQL	DNN [9]	-/C#	4
	Bugzilla [3]	-/Perl	11
	Drupal [10]	-/PHP	31
	MediaWiki [14]	-/PHP	28
	Moodle [15]	-/PHP	15
	WordPress [33]	-/PHP	18
	Odoo [19]	-/Python	2
ORM-based	Oscar [22]	Django/Python	2
	PostHog [24]	Django/Python	3
	Redash [25]	Django/Python	4
	Saleor [28]	Django/Python	4
	Sentry [29]	Django/Python	8
	Weblate [32]	Django/Python	6
	Zulip [34]	Django/Python	8
	BroadLeaf [2]	Hibernate/Java	2
	OpenMRS [20]	Hibernate/Java	3
	October [18]	Laravel/PHP	1
	Pixelfed [23]	Laravel/PHP	2
	AlchemyCMS [1]	Ruby-on-Rails/Ruby	1
	Canvas LMS [4]	Ruby-on-Rails/Ruby	29
	Danbooru [6]	Ruby-on-Rails/Ruby	3
	diaspora* [7]	Ruby-on-Rails/Ruby	2
	Discourse [8]	Ruby-on-Rails/Ruby	9
	Gitlab [11]	Ruby-on-Rails/Ruby	30
	LinuxFr.org [13]	Ruby-on-Rails/Ruby	1
	OpenProject [21]	Ruby-on-Rails/Ruby	2
	Redmine [26]	Ruby-on-Rails/Ruby	2
	ROR [27]	Ruby-on-Rails/Ruby	1
	Sharetribe [30]	Ruby-on-Rails/Ruby	2
	Spree [31]	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.

Table 3.2: Web applications and numbers of bugs being studied in request race characteristic study

Application (Abbreviation)	**Server-Side Language**	**Number on Server Side**	**Number on Client Side**
WordPress (WP) [33]	PHP	18	22
MediaWiki (MW) [14]	PHP	28	10
Drupal (DPL) [10]	PHP	31	7
Moodle (MDL) [15]	PHP	15	8
BugZilla (BZ) [3]	Perl	11	0
Odoo (OD) [19]	Python	2	8
DNN (DNN) [9]	C#	4	1
OpenMRS (MRS) [20]	Java	2	0
Gitlab (GL) [11]	Ruby	26	5
Discourse (DC) [8]	Ruby	7	2
Redmine (RM) [26]	Ruby	2	0
Spree (SPR) [31]	Ruby	0	2
Node.js-based (Node.js)	Javascript	11	1
Total		157	66

Table 3.3: 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

Table 3.4: 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													
AV with two instances of the same request handler	10	21	22	9	6	2	3	2	6	2	2	6	91
AV with two different request handlers	8	7	9	6	4	0	1	0	6	3	0	2	46
OV 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													
AV: Change application logic	4	5	3	2	1	0	0	0	7	1	0	0	23
AV: Handle the race properly when it happens	2	6	4	5	1	0	0	0	3	0	0	0	21
AV: Guarantee column value uniqueness	2	0	0	3	1	0	1	0	1	0	1	1	10
AV: Use duplication-tolerant SQL query	1	5	0	0	0	0	0	0	0	0	0	0	6
AV: Refactor query statement	0	1	7	1	0	1	0	0	0	1	0	0	11
AV: Add transaction or leverage existing transaction	0	0	0	0	3	0	0	0	0	1	0	0	4
AV: Lock the table	0	2	3	0	4	0	0	0	0	0	1	0	10
AV: Use atomic API	0	0	0	0	0	0	0	0	0	0	0	3	3
AV: Fix in frontend	2	1	1	0	0	0	0	0	0	0	0	0	4
OV: Enforce order by callback function	0	0	0	0	0	0	0	0	5	1	0	1	7
OV: Use synchronous execution	0	0	0	0	0	0	0	0	2	0	0	0	2
OV: Remove transaction	0	0	0	0	1	0	0	0	1	1	0	0	3

	WP	MW	DPL	MDL	BZ	OD	DNN	MRS	GL	DC	RM	Node.js	Total
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													
AV: Change application logic	0	0	1	1	0	0	0	0	0	0	0	0	2
AV: Handle the race properly when it happens	0	1	7	2	0	1	0	1	2	1	0	0	15
AV: Guarantee file name uniqueness	1	2	1	1	0	0	0	0	0	1	0	0	6
AV: 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													
AV: Change application logic	2	4	3	0	0	0	0	0	0	0	0	0	9
AV: Handle the race properly when it happens	0	1	1	1	0	0	0	0	0	0	0	0	3
AV: Guarantee cache key uniqueness	3	0	0	0	0	0	0	0	0	0	0	0	3
OV: 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													
AV: Use database instead	0	1	0	0	0	0	0	0	0	0	0	0	1
AV: Add language-provided lock	0	0	0	0	0	0	0	1	0	0	0	0	1
OV: Change application logic	0	0	0	0	0	0	0	0	2	0	0	4	6
OV: Add condition to enforce order	0	0	0	0	0	0	0	0	0	0	0	2	2
OV: Read again after a delay	0	0	0	0	0	0	0	0	2	0	0	0	2

Table 3.5: Overall results on racing resource types, root cause patterns, manifestation conditions, and unserializable patterns for AVs

	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							
AV with two instances of the same request handler	73	19	4	3	53	6	158
AV with two different request handlers	35	12	1	0	10	2	60
OV within one request handler	1	4	0	0	23	3	31
Unserializable Patterns for AV							
$(\epsilon \mid R)R'(A \mid W \mid D)(A' \mid W' \mid D' \mid R')$	91	28	5	3	60	4	191
$A(A' \mid R' \mid W')A$	2	0	0	0	0	0	2
$W(A' \mid R')(W \mid A)$	2	1	0	0	1	0	4
$DD'AA'$	4	0	0	0	0	0	4
$(A \mid W)(A' \mid D' \mid W')R$	9	2	0	0	2	4	17

Table 3.6: 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)

Table 3.7: 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							
AV with two instances of the same request handler	31	11	1	2	25	4	74
AV with two different request handlers	11	4	0	0	2	0	17
OV 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

Table 3.8: 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

	raw-SQL	Django	Hibernate	Laravel	Ruby on Rails	Node.js	Total
Number of bugs studied	109	35	5	3	86	11	249
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

CHAPTER

4

DETECTING SERVER-SIDE SCOPE-BASED REQUEST RACES

In this chapter, we discuss the design and implementation details of ReqRacer, and show the evaluation results.

4.1 ReqRacer Design

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 4.1 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

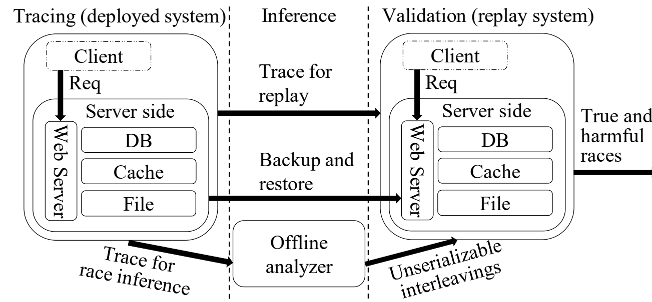


Figure 4.1: The architecture of REQRACER

by the second stage, and observes their effects. Racing requests with triggered harmful unserializable interleavings will be reported.

4.1.1 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 4.2. 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 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

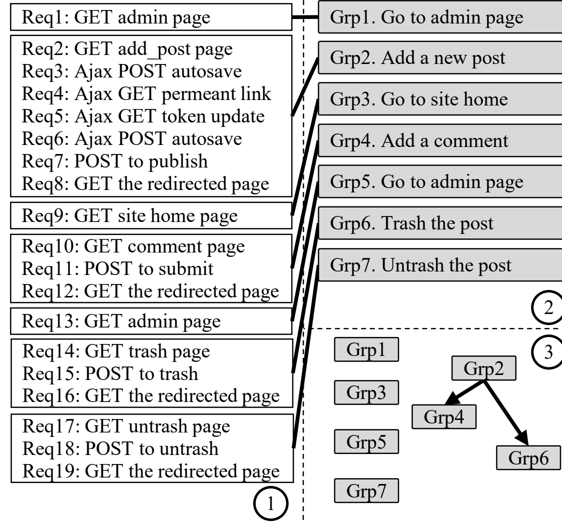


Figure 4.2: An example of dependency-graph construction

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 4.2, 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 4.2. 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 4.2.②. REQRACER recognizes these RRR dependencies through program modifications to embed tokens into requests and responses. Note that the text description in Figure 4.2.② 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 4.2.③ 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 4.2.③ 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 4.2, 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.

4.1.2 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 4.2, 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 [80, 81] 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.

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.

Dependency-Graph Construction

As illustrated in Section 4.1.1, 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 [5]. 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.

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 3.3. 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 [68, 127, 128] to handle the majority of races between two instances of the same request handler as discussed in Section 3.2, 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.

4.1.3 Replay-Based Validation

To further prune false positives, we employ a replay-based validation phase, which tries to validate each inferred request race. The high-level idea is to explore different interleavings of conflicting accesses in a pair of racing request handlers and prune those pairs that do not lead to program misbehaviors as false positives. Using such a validation phase to prune false positives in predictive race and concurrency-bug detection is pioneered by RaceFuzzer [108] and CTrigger [99] for multi-threaded programs, and this approach is also adopted by race and concurrency-bug detection in other domains [62, 70, 72, 81, 85, 86, 94, 125].

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 4.1.1, 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.

4.2 Implementation and Evaluation of ReqRacer

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 [12], 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.

4.2.1 Effectiveness Results

Table 4.1 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 3.2, 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 4.1 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.

To summarize, the happens-before relationships modeling and the unserializable interleaving checking combined are very effective and prune the majority of false positives. The

replay-based validation step further prunes false positives that do not lead to application misbehaviors or database errors during real runtime. Without the validation step, these false positives would remain in our detection results. The numbers of false positives before applying the validation step are thus the numbers of false positives pruned by this replay-based validation step. The validation step is essential for our framework to achieve low to no false positives.

4.2.2 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.

4.3 Threats to Validity

The evaluation of REQRACER is 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.

Table 4.1: Overall evaluation results. “Reqs” shows the number of requests in the workload. “#Acc.” represents the number of database or cache accesses in the trace. “Racing Resc.” represents the type of racing resources. “Con. 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 Resc.**	**Reqs**	**#Acc.**	**Con. Reqs**	**RRR**	**SPK**	**S**	**R**	**TP**	**Likely TP**	**FP**	**Con. Reqs**	**S**	**R**	**TP**	**Likely TP**	**FP**
WP 11073	DB	24	533	118	-22	-15	-74	-6	1	0	0	3	0	-2	0	1*	0
WP 11437	DB	9	181	26	-10	-2	-11	-2	0	1*	0	1	0	0	1	0	0
WP 24933	DB	15	277	18	-3	-8	-6	0	1	0	0	2	0	-2	0	0	0
MW 40594	DB	18	1008	14	0	0	-14	0	0	0	0	1	0	0	1	0	0
MW 69815	DB	23	2069	11	0	0	-11	0	0	0	0	1	0	0	1	0	0
MDL 28949	DB	47	2878	407	-37	-51	-297	-17	1	4*	0	11	0	-4	2/4*	1*	0
MDL 43421	DB	31	1141	122	-11	-48	-44	-18	0	1*	0	5	0	-1	1/1*	1*	1
MDL 51707	DB	14	250	21	-4	0	-16	0	1	0	0	3	-3	0	0	0	0
MDL 59854	DB	101	1969	492	-25	-81	-375	-10	0	1*	0	20	-12	-2	2/3*	1*	0
DPL 1484216	DB	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	14	263	10	-2	0	-7	0	1	0	0	9	-7	-2	0	0	0

CHAPTER

5

UNDERSTANDING AND DETECTING REQUEST RACES BREAKING IDEMPOTENCE

In this chapter, we present our research results on understanding and detecting request races breaking idempotence.

Our preliminary study for this research shows out that around 50% of the included request races invalidate request handler idempotence enforcement. Therefore, it is beneficial to deeply study how request races invalidate existing idempotence enforcement and why existing enforcement is not concurrency safe.

5.1 RQs for Understanding Request Races Breaking Idempotence

Specifically, in this work, we focus on answering the following three research questions:

- **RQ1:** How common are request races that invalidate request handler idempotence enforcements? How to determine if a request handler needs to be idempotent?
- **RQ2:** How request races invalidate request handler idempotence enforcements under concurrent executions?
- **RQ3:** What strategies are suitable for enforcing request handler idempotence and what guidance can we provide to developers on building idempotent request handlers?

We check 85 bugs racing on the database records in web applications built upon ORM frameworks from our previous study [103] and filter out bugs that race on other resources and happen in raw SQL or Node.js web applications.

We define request handler idempotence as the following. A request handler can be considered idempotent if the system states remain the same without application misbehaviors no matter how many times the same request handler is executed with the same input parameters. Application misbehaviors include duplicate records generated in the database and application crashes, but do not include developer rendered error messages.

For **RQ1**, we have shown that around 50% of the collected request races invalidate idempotence enforcement and we further do a deep study on these races. We look into the manifestation condition of included request races and the request handler business logic with fixing patches, which has the expected functionality from developers, to determine if a request handler needs to be idempotent.

For **RQ2**, we document what idempotence enforcement methods are used before fixing. We find that app-side validations, feral uniqueness validations, and database unique constraints are the used methods. We further look into how these methods are implemented and whether they work to achieve request handler idempotence under sequential executions. We reason about how request races invalidate these methods under concurrent executions.

For **RQ3**, we investigate how different proposed patches address request races that invalidate request handler idempotence enforcement. We find out that exception handling is the most commonly used to achieve idempotence under concurrent executions. We further investigate how these exception handling blocks are implemented to guide developers on building idempotent request handlers. We discuss whether other proposed patches can help enforce request handler idempotence under concurrent executions.

We extract rules from request races in ORM-based web applications and build a static

tool for detecting request races, with a special focus on idempotence enforcement. We first identify ORM APIs that are frequently involved in request races that invalidate request handler idempotence enforcements. We focus on ORM APIs because they simplify database access by abstracting away the complexity of writing raw SQL queries and they are suitable for static detection. We examine sites where these ORM APIs are used and extract rules about what usage of APIs may lead to request races. We extract two rules from 23 request races that lead to duplicate entries and application crashes and one rule from 5 request races that lead to incorrect column value update.

To measure the applicability of the extracted rules, we implemented a static request race detection tool. The tool finds a total of 447 new bugs in the latest versions of 19 ORM web applications, including 10 built on Ruby on Rails, eight on Django, and one on Laravel. In these new bugs, 304 will lead to application crashes, 83 will lead to wrong values during concurrent updates, and 60 will lead to duplicate entries in the database.

5.2 Methodology

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

Bug Selection. We start from bugs in our previous study [103] collected from the bug-tracking system of real-world open-source web applications. We first double-check the bug reports to make sure that our understanding is consistent with the previous results and we make the following three changes to the results from our previous study. We change the racing resource category of two bugs from database records to shared memory objects. We remove one bug because we conclude it is not a race bug but a performance improvement. We change the root cause pattern of one bug from order violation to atomicity violation.

Next, we take several steps to filter bugs. In this work, we are going to focus on atomicity violation races involving Object-Relational Mapping (ORM) APIs. ORM APIs allow developers to interact with the database. Therefore, request races that race on files, cache, and shared-memory objects are excluded. Request races that happen in raw-SQL and Node.js web applications are also excluded because they are not built on ORM frameworks. To this end, we have 85 request races that happen in ORM web applications and they race on database records.

Request Race Study. In our study, we aim to understand the relationship between request races and request handler idempotence. To answer **RQ1**, we first categorize these

85 bugs based on the following criteria to find out request races invalidating idempotence enforcement. 19 request races happen between different request handlers and 17 request races happen between instances of the same request handler but with different input parameters. Request races that invalidate idempotence enforcement happen between instances of the same request handler with the same input parameters, and these 36 are excluded. We further investigate the business logic of the remaining races to determine if the request handlers need to be idempotent. We find 41 request races invalidate request handler idempotence enforcement and we summarize three types of business logic that require idempotent request handlers. We adapt the general definition of idempotence for request handlers.

To answer **RQ2**, we document what methods are adopted by developers to achieve request handler idempotence. We find that app-side validations and feral uniqueness validations are the most commonly used methods. We look into how these methods are implemented and check if using them can achieve request handler idempotence under sequential and concurrent executions. We find that 5 request handlers are not idempotent under sequential executions and 36 request handler idempotence enforcements are invalidated by request races under concurrent executions. We reason about how request races invalidate these approaches under concurrent executions.

To answer **RQ3**, we look into how different proposed patches address request races that invalidate request handler idempotence enforcement. We find that exception handling is the most commonly used, and it is used to fix 25 request races that invalidate idempotence enforcement. We discuss whether other proposed patches help enforce request handler idempotence under sequential and concurrent executions. We further look into how exception handling code blocks are implemented and summarize the following four logic, which are retrying the failed transaction, getting and returning the latest data, overwriting the database with current request data, and catching the exception and rendering an error message. We expect our results can be useful to guide developers to build idempotent request handlers.

Finally, we extract rules from the studied request races and build a static tool to evaluate the rules. We start by investigating what ORM APIs are frequently involved in request races and find out *save* and *create* account for the majority. By examining 27 request races that happen in request handler storing objects with attributes that need to be unique, we summarize 2 rules for detecting request races leading to duplicate entries and application crashes. By examining 5 request races that happen in request handler doing atomic in-

crements, we summarize 1 rule for detecting incorrect column value updates. We run the static tool on the latest version of 19 web applications to detect new bugs.

5.3 Understanding Request Races and Request Handler Idempotence

In this section, we present our study results and answer the three research questions.

5.3.1 RQ1: Request Races and Request Handler Idempotence

We present results about how common are request races that invalidate request handler idempotence enforcement and what we have done to determine if a request handler needs to be idempotent. After that, we adapt the general definition of idempotence to a request handler.

We determine if a request race is related to request handler idempotence by checking request race manifestation conditions and request handler business logic. Table 5.1 shows how we determine if request races are related to idempotence.

Table 5.1: Determine if Request Races Happen in Request Handlers that Need to Be Idempotence

Label	Number
Not related: Happen between different request handlers	19
Not related: Happen between same handler but different input parameters	18
Need not to be idempotent: Atomic increments are not idempotent	7
Need to be idempotent	41
Total	85

When checking the manifestation conditions, 37 request races are pruned because they do not satisfy the general definition of idempotence which requires applying the same operations multiple times 19 request races happen between different request handlers. The operations from different request handlers are different. 18 request races happen between instances of the same request handler but application misbehaviors only happen when the

Table 5.2: Business Logic of Request Handlers that Need to be Idempotent

Logic	Number
Storing objects with attributes that need to be unique	36
Sending messages filled by table data	3
Applying a voucher to the order	2
Total	41

request handler input parameters are different. With different input parameters, the request handlers add or update different values in the database, and the operations from these request handlers are different. We do not consider these request races when discussing request handler idempotence.

We further investigate the business logic of the remaining 48 request races to determine if the request handlers need to be idempotent. We refer to the business logic with the fixing patch because the code has the correct functionality that developers expect. The request handlers of 7 request races should do atomic increments on column values. They are not idempotent because executing atomic increments multiple times has different results in the database from executing it once, and therefore these request handlers do not need to enforce idempotence.

Table 5.2 lists the business logic in request handlers that need to be idempotent for the remaining 41 request races.

```
ALTER TABLE ONLY
  public.discussion_topic_materialized_views
  ADD CONSTRAINT
  discussion_topic_materialized_views_pkey
  PRIMARY KEY (discussion_topic_id);

class DiscussionTopic::MaterializedView < ActiveRecord::Base
    def self.for(discussion_topic)
      self.find_by_discussion_topic_id (discussion_topic.id) ||
      self.create!(:discussion_topic => discussion_topic)
```

36 request handlers store a new object with attributes that need to be unique in the

database. In the above code snippet, table discussion_topic_id_materialized_views has a primary key of discussion_topic_id, which needs to be unique in the database. In the request handler, a DiscussionTopicMaterializedView object is created by specifying the value of the primary key. Such request handlers need to be idempotent, otherwise, duplicate entries or database duplicate key errors could happen.

3 request handlers send messages or emails to users that use data from a table. In the following code snippet, a turtle_message object is fetched from Message table and it is passed as input to do_add_reaction to send a message to users. Such request handlers need to be idempotent, otherwise, users get duplicated messages or emails.

```
1 def send_initial_realm_messages(realm: Realm) -> None:
2   turtle_message = Message.objects.get(id__in=message_ids,
        content__icontains="cute/turtle.png")
3   ...
4   do_add_reaction(turtle_message...)
```

2 request handlers apply a voucher when placing an order. In the following code snippet, when placing an order with a voucher, the request handler reads data from the voucher table. It verifies that there are vouchers available and update the object attributes to reflect discounts. If no vouchers are available, order discount attributes are not updated. Such request handlers need to be idempotent, otherwise, operations violating application logic invariants could happen, such as a one-usage voucher being used multiple times.

```
1 def create_order(self):
2   ...
3   voucher = self._get_voucher()
4   if voucher is not None:
5       discount = self.discount
6       order_data['voucher'] = voucher
7       order_data['discount_amount'] = discount.amount
8       order_data['discount_name'] = discount.name
9   order = Order.objects.create(**order_data)
```

After determining if a request handler requires to be idempotent, we can adapt the definition of idempotence for a request handler. We define request handler idempotence

as the following. A request handler can be considered idempotent if subsequent executions with the same input parameters no longer have side effects on resources or system states after the first execution of the request handler. Side effects on resources include duplicate records generated in the database and specific logic violations such as voucher multiple redemptions, and side effects on system states include application crashes. Note that we consider situations such as when a user receives an error message saying "the record already exists" idempotent because no duplicate entries are in the database, the application does not crash, and application logic is not violated.

Our results show that around 50% of our collected request race bugs in ORM-based web applications invalidate request handler idempotence. Such a distribution indicates that it is beneficial to study thoroughly on this type of request race.

5.3.2 RQ2: Reasons Why Request Races Invalidate Idempotence Enforcements

Next, we discuss how request races invalidate idempotence enforcement in request handlers under concurrent executions.

Table 5.3 lists what idempotence enforcement methods are used before fixing.

Table 5.3: Idempotence Enforcement Methods Used Before Fixing

ORM framework	Idempotence Enforcement Methods	Number
Ruby on Rails	App-side validation	17
	Feral uniqueness validation	7
	Database unique constraints only	1
	None	2
Django	App-side validation	12
	Database unique constraints only	1
Laravel	None	1

36 request races invalidate the idempotence enforcement in request handlers that works under sequential executions. 29 request races happen when app-side validation is used and 7 happen when feral uniqueness validation is used. We can interpret from the name that they are implemented differently. App-side validation is manually defined by developers

in the application source code and feral uniqueness validation is supported by the feral concurrency control mechanisms provided by ORM frameworks.

Next, we discuss how they work under sequential executions and break under concurrent executions. We then discuss more on their difference.

We use the following code snippet from a Canvas-lms bug to illustrate how scope-based request races break app-side validations.

```
1  def set_data
2    ...
3    cd = CustomData.where(user_id: @context.id, namespace:
       @namespace).first
4    cd = CustomData.new(user_id: @context.id, namespace:
       @namespace) unless cd
5    ...
6    cd.save
```

In this code snippet, in line 3, the function calls API *first* to get one from the objects that satisfy the where condition using the provided user_id and namespace. The result from *first* is validated using the unless clause in line 4. If the result is null, which indicates no objects with the provided user_id and namespace exist, a new object is created by calling *CustomData.new*; otherwise, objects with the provided user_id and namespace exist and execution of *CustomData.new* is skipped. The *save* API is called in line 6 and it can do both INSERT and UPDATE. A new record is inserted if the CustomData object is newly created, and an existing record is updated if the CustomData object is not newly created. App-side validations can help enforce idempotence under sequential execution because only the first request finds the object does not exist and executes ORM API to add it. Other requests will update existing records. Executing the request handler multiple times with the same input parameters has the same results in the database as executing it only once, and the application does not crash. The request handler is idempotent under sequential execution. However, under concurrent execution, when validating results from *first* and calling *save* are not correctly scoped into a transaction, they can be freely interleaved and scope-based request races happen. Both concurrent executing request handlers get the same results from the app-side validation indicating that no objects with the provided user_id and namespace exist and execute *save* to add a new record, which violates the

database unique constraints and crashes the application. Idempotence cannot be enforced under concurrent executions.

We use the following code snippet from a Canvas-lms bug to illustrate how level-based request races break feral uniqueness validations.

```
1  class PlannerOverride < ActiveRecord::Base
2    validates :plannable_id, uniqueness: {scope: [:user_id,
       :plannable_type]}
3
4  def create
5    plannable_type = PlannerHelper::PLANNABLE_TYPES [params
       [:plannable_type]]
6    plannable = plannable_type.constantize .find(params
       [:plannable_id])
7    planner_override = PlannerOverride
       .new(plannable:plannable, user:@current_user, ...)
8    planner_override.save
```

In line 2 of the code snippet, the feral uniqueness validation is declared to verify if plannable_-id is unique concerning user_id and plannable_type in the PlannerOverride model file. In line 8, when the *save* API is called by the planner_override object, the Ruby on Rails framework wraps a transaction for *save* and validates if the plannable_id of the planner_-override object is unique about values of user_id and plannable_type in the database. When requests are sent sequentially, only the first passes feral uniqueness validation, and later request gets the error message as the response. The request handler is idempotent under sequential execution because the application does not crash and there are no duplicate entries in the database. However, level-based request races can invalidate such idempotence enforcement. Under isolation levels weaker than SERIALIZABLE, transactions running concurrently can lead to non-serializable behaviors. It is still possible that concurrent requests pass the feral uniqueness validation and execute *save* to add a new record, which breaks request handler idempotence.

Both these two types of validations work to enforce request handler idempotence under sequential executions but do not work under concurrent executions. They have the following difference. For app-side validation, because it is manually defined by develop-

ers, developers can validate various properties to control the execution of the following operations, but developers need to define them every time a certain property needs to be verified. For feral uniqueness validation, it only verifies the uniqueness property. Developers only need to declare the feral uniqueness validation once in the model file regarding which attributes of the model objects need to be unique, and it executes every time API *save* is called by the model objects. It executes before *save* and is wrapped into the same transaction with *save*. If the validation fails, the whole transaction is automatically canceled and an ActiveRecord::RecordInvalid exception is thrown. The exception is handled by the framework and a message saying "the value has already been taken" is returned to the user.

3 request races happen when no idempotence enforcement methods are used in the request handlers, and these request handlers are not idempotent under sequential execution.

2 request races, 1 in Rails apps and 1 in Django apps, happen when developers use only database unique constraints. The following code snippet shows how database unique constraints are defined.

```
1 CREATE TABLE `auth_tokens` (
2     ...
3     `token` varchar(255) DEFAULT NULL,
4     UNIQUE KEY `index_auth_tokens_on_token` (`token`) USING
        BTREE
5     ...
6 )
```

When creating the table auth_tokens, a unique key named index_auth_tokens_on_token is defined in line 4 of the code snippet, and the unique constraints are applied to column token. Database unique constraints prevent duplicate entries from being stored, and a duplicate key error is thrown by the database when a query execution violates the defined unique constraints. In ORM web applications, the error causes an exception that bubbles up the framework stack and if not properly handled, crashes the application. Under sequential execution, only the first request succeeds and the following requests could crash the application due to unhandled exceptions. The request handlers in these 2 request races are also not idempotent under sequential execution.

Request races can break the implemented request handler idempotence enforcement.

Developers can also incorrectly implement request handlers making them not idempotent even under sequential executions. We are in great need of a guide to help developers build request handlers that are idempotent under concurrent executions.

5.3.3 RQ3: Fixings for Enforcing Idempotence

Finally, we discuss how different proposed patches address request races and whether they achieve request handler idempotence under concurrent executions.

Table 5.4: Proposed Patches for Idempotence Enforcement

	Proposed patch	Number
Enforce idempotence concurrently	App-side exception handling	25
	Add locks	4
	Rewrite queries	2
	Remove a transaction	1
	Ad-hoc synchronization	1
Cannot enforce idempotence concurrently	App-side or feral uniqueness validation and DB constraints	3
	Add database unique constraints only	2
	Add new logic	2
	Frontend	1
Total		41

Table 5.5: Exception Handling Logic

Framework	Exception Handling Strategy	Number
Ruby on Rails	Retry failed transaction	10
	Get and return latest data	3
	Overwrite database with current request data	3
	Catch exception and render an error message	2
Django	Retry failed transaction	1
	Get and return latest data	3
	Overwrite database with current request data	2
	Catch exception and render an error message	1

As Table 5.4 shows, app-side exception handling is used in the proposed patches for 25 request races. 23 request races happen in request handlers that are not idempotent under concurrent executions and do logic of storing objects with attributes that need to be unique. 2 happen in request handlers not idempotent under sequential executions because developers only used database unique constraints.

We further investigate the 25 patches to understand how the exception handling code block is implemented.

It is necessary to look into how exception handling is implemented because ORM APIs do not automatically retry the failed transaction that throws exceptions and it is the developer's responsibility to implement the logic. The atomicity property of the transaction aims to make transactions to be safely retried. However, implementing retry logic is not easy. Not every transaction is worth a retry [78]. For example, if the error is due to overload, simply retrying the transaction will make the problem worse. If the errors are transient, such as deadlocks or isolation violations, it is worth retrying the transaction; while for permanent errors, such as constraint violations, a retry is pointless. By studying how exception handlings are implemented in these proposed patches, we can provide comprehensive insights to developers about how to implement exception handling to build idempotent request handlers.

Table 5.5 shows the results of exception handling logic. We find that the exception-handling logic is highly related to the query logic of ORM APIs wrapped into the transaction.

The exception handling code blocks for 10 request races in Rails apps and 1 in Django apps are implemented to retry the failed transaction. Transaction retries can achieve idempotence when the queries wrapped in the transaction include a preceding SELECT query before modifying database entries. These queries could be issued by a single API, such as *find_or_create_by*, or two APIs, such as *exist?* then *save*. While retrying a failed transaction, the SELECT query will get some results, the corresponding validation will not pass, and the following queries that modify database entries will not be issued, which prevents duplicate key errors from happening again.

If the queries wrapped by the transaction do not include a preceding SELECT query before modifying database entries, retry the transaction will execute the queries modifying database entries and cause errors again. We observe several different implementations in the exception handling code block rather than simply retrying the failed transactions. A code block to get and return the latest data is used for 3 request races in Rails apps and 3 in Django apps. A code block to overwrite conflicting data with what data are in the current

request is used for 3 request races in Rails apps. A code block that renders an error message to users is used for 2 request races in Rails apps and 1 in Django apps.

By using app-side exception handling, the application no longer crashes under concurrent executions and the request handle idempotence is hence enforced.

Next, we discuss other proposed patches that help enforce request handle idempotence under concurrent executions.

Adding locks is used in the patches for 4 request handlers. The locks are applied on certain rows by a SELECT FOR UPDATE query. This method is used to help enforce idempotence for 2 request handlers sending messages filled by table data and 2 request handlers applying a voucher to the order. They all have used app-side validation. The locks are acquired before app-side validation. The queries fetching database records for validation are serialized by the locks, and only the first one passes the app-side validation. Request handlers will not send the same message repeatedly nor be able to use a one-usage voucher multiple times.

Replacing ORM APIs with manual-written SQL queries is used for 2 request races in request handler storing objects with attributes that need to be unique. The replacement SQL queries have a syntax of “INSERT ON CONFLICT DO UPDATE” and the database will not throw duplicate key errors that may crash the application. The application will not crash and there will not be duplicate entries in the database under concurrent executions.

Removing a transaction is used in the patch for 1 request race from Gitlab storing unique objects. In the database, a failing INSERT SQL query invalidates any surrounding transaction. Even though there is proper error handling for the duplicate key error from the database, the additional surrounding transaction promotes the error into an unhandled invalid transaction error. By removing the unnecessary transaction, there will not be an invalid transaction error, and there is code logic handling the duplicate key error.

Ad-hoc synchronization is used in the patch for 1 request race to synchronize authorization refreshing work. This method is similar to adding locks and helps enforce request handle idempotence under concurrent executions.

Lastly, we discuss proposed patches that **cannot** enforce request handle idempotence under concurrent executions.

For the 3 request races that happen in request handlers without using any methods to enforce request handler idempotence the patch adds app-side or feral uniqueness validations as well as database unique constraints. As discussed before, these methods cannot enforce request handle idempotence under concurrent executions.

Adding database unique constraints is used in the patch for 2 request races. Feral uniqueness validation has been used in the request handlers but duplicate entries still happen in the database. Adding database unique constraints help prevent duplicate entries from happening but do not help enforce request handler idempotence.

The patches for 1 request handler storing unique objects and 1 request handler sending messages, add new logic. For the 1 in request handler sending messages, the new logic is a condition check which skips sending the message when it is passed. For the 1 in request handler storing unique objects, the new logic is adding a suffix to an object field, making it unique. These methods can still be invalidated by request races under concurrent executions.

1 request race is fixed by changing the frontend code but requests can be sent by scripts that can bypass the frontend code, making this method not effective in enforcing idempotence.

We hope our results can draw developer's attention to the importance of request handler idempotence and help them implement idempotent request handlers.

5.4 Static Detection for Request Races

In this section, we present three rules extracted by studying 41 request races in request handlers that need to be idempotent and 7 request races in request handlers doing atomic increments. To measure the applicability of these simple rules, we build a rule-based static checker that is capable of locating these bug patterns. We discuss how the static tool is implemented and results of applying the tool to 19 real-world open-source web applications.

5.4.1 Extract Rules from Request Races

First, we discuss what ORM APIs are frequently involved in request races, how we extract rules by studying the site where APIs are called, and what the extracted rules are.

ORM APIs in Request Races. ORM APIs provide us with opportunities to do static analysis on the site where they are used. In ORM web applications, developers usually use ORM APIs to interact with the database. ORM APIs abstract away the complex query syntax and allow developers to map database tables to classes and objects in their code. Developers can use object-oriented programming concepts to manipulate data in the database.

Table 5.6: API *save* and *create* in Request Races

Label	Involve API	save	create
Need to be idempotent	41	22	10
Atomic increments are not idempotent	7	5	0
Happen between different request handlers	12	2	0
Happen between same handler but different input parameters	14	4	2
Total	74	33	12

In this work, we try to build a static tool to detect request races with a special focus on those that invalidate request handler idempotence. From the results of RQ1, 41 request races happen in request handlers that need to be idempotent.

As shown in Table 5.6, all these 41 request races involve ORM API, and *save* and *create* account for 32 of them. *save* and *create* also appear in other types of request races. They account for 5 out of 7 of request races in request handlers doing atomic increments, 2 out of 12 request races happen between different request handlers, and 6 out of 14 request races happen between instances of the same request handler but different input parameters.

Based on the distribution of *save* and *create* used in request races invalidating request handler idempotence, we focus on studying how they are used and implement our static tool to search their call sites.

Table 5.7: API *save* and *create* with Request Handler Logic

Logic	save	create
Storing objects with attributes that need to be unique	18	9
Sending messages filled by table data	3	0
Applying a voucher to the order	1	1
Atomic increments	5	0

Rules Extracted. We extract rules based on request handler logic. Table 5.7 shows how request handler logic is associated with the number of usages of these two APIs. 18 of *save* and 9 of *create* are in request handlers that store objects with attributes that need to be unique. We first present two rules extracted by studying these 27 sites.

Rule1: If a function modifies the object attributes that are mapped to fields defined as unique keys in the database and it then calls ORM APIs such as *save* or *create* to store the object in the database, it is a bug when no proper exception handling or no synchronizations are used to synchronize these operations.

The following code snippet is the same example in RQ2. In the database, a unique key is defined on columns plannable_type, plannable_id, and user_id. From line 8 and line 9, a plannable object contains data for both plannable_type and plannable_id. In line 10, a planner_override object is built using the plannable object and data from current_user, which include a user_id, and they form the unique key. In line 11, the planner_override call *save* to store data in the database, and it could cause application crash under concurrent execution due to unhandled duplicate key errors from the database.

```
1  CREATE UNIQUE INDEX
2    index_planner_overrides_on_plannable_and_user
3    ON public.planner_overrides
4    USING btree
5    (plannable_type, plannable_id, user_id);
6
7  def create
8    plannable_type = PlannerHelper::PLANNABLE_TYPES [params
       [:plannable_type]]
9    plannable = plannable_type.constantize .find(params
       [:plannable_id])
10   planner_override = PlannerOverride
       .new(plannable:plannable, user:@current_user, ...)
11   planner_override.save
```

As shown in the above example, the unique keys do not explicitly appear as the input parameters of *save*. For example, plannable_id is inferred to be included in the plannable object. We manually identify if ORM APIs modify object attributes that are mapped to fields defined as unique keys in the database.

The patterns from 20 of 27 request races match this rule. These 20 request races all cause application crashes for the external effects, which this rule can detect. This rule does not report a bug if proper exception handling is used.

Rule2: If there are feral uniqueness validations, developers assume some properties to be unique. It is a bug if there are feral uniqueness validations but no corresponding unique keys are declared in the database schema.

The following code snippet shows how feral uniqueness validation is defined in OpenProject. As discussed in RQ2, request races break feral uniqueness validations and duplicate entries could happen if there are no database unique constraints.

```
1   validates :identifier, uniqueness: { case_sensitive: true }
```

The patterns from 3 of the remaining 7 request races match this rule. These 3 request races all cause duplicate entries for the external effects, and this rule can detect request races that may lead to duplicate entries happening in the database during runtime.

Without a sophisticated understanding of the application logic, we have a limited source to infer developer's expectation about whether an object attribute needs to be unique. Feral uniqueness validations are declared by developers and they are a convincing source to infer developer's expectation about the uniqueness of object attributes. Duplicate entries can be reliably prevented by database unique constraints, and without them, duplicate entries are possible under concurrent executions.. If we infer that some object attributes need to be unique by checking the definition of feral uniqueness validations but corresponding database unique constraints are missing, it is a bug where duplicate entries can happen during concurrent executions. This rule does not report a bug if the corresponding database unique constraints of feral uniqueness validations exist.

For the remaining 4 ORM API usages in request handlers storing objects with attributes that need to be unique, before applying the proposed patches, the related columns are not defined as unique keys in the database, and there are no feral uniqueness validations on these columns in the application source code. Without a deep understanding of applications logic, we cannot determine whether APIs modifying these object attributes need to have exception handling and they need to be unique.

For the 3 usages in request handlers sending messages filled by table data, it is necessary to do data flow tracking. For the 2 usages in request handlers applying a voucher to the order, it is required to check invariants modeling such logic. We leave these for future work because they account for a small portion of request races that invalidate idempotence enforcement.

Next, we present the third rule extracted by studying 7 request races that happen in

request handler doing atomic increments.

Rule3: If a function increments or decrements the value of an object attribute before calling ORM API *save*, it is a bug if no synchronizations are used.

In the following code snippet from Django-lfs, the stock_amount field is decremented on the application side if manage_stock_amount of the Project object is True, and the object with updated values is stored in the database by calling ORM API *save*. Without correct synchronizations, the stock_amount value can be wrong under concurrent executions.

```
1 class Product(models.Model):
2     ...
3     def decrease_stock_amount(self, amount):
4         if self.manage_stock_amount:
5             self.stock_amount -= amount
6         self.save()
```

The patterns from 5 request races match this rule. The request handlers do increments on column values but the atomicity is not enforced and the values are not correctly updated when running concurrently. This rule can detect these request races that lead to incorrect update values. This rule does not report a bug if atomic update operations, such as the F method in Django, or locks are used.

5.4.2 Implementation

The tool is implemented in Python to parse information from various types of files. It requires the application source code and its database schema files to find potential request races. The application source code is available from their public repositories. However, the database schema files may not be included in the repositories. To get the schema files, we follow the instructions to install the web applications. If the application provides a docker image, we install the docker image and dump the schema files from the database container.

The schema files could be in Ruby or SQL format and they have different syntaxes for creating a table and defining unique keys. For SQL files, The table name is extracted from CREATE TABLE query. The lines defining unique keys have a syntax of CREATE UNIQUE INDEX or ADD CONSTRAINT UNIQUE and the corresponding table name is included in these lines. For Ruby files, the table name is extracted from the create_table line. The lines defining unique keys are following the create_table line with a syntax of “table.index unique:

true". The schema-parsing module of the tool outputs a hashmap of table names to unique key names. If the table has more than one unique key, the schema-parsing module groups them by the table name.

The application source code could be in Ruby, Python, or PHP based on the ORM framework used. Different programming languages have different syntaxes and the tool has three dedicated code-parsing modules for Ruby, Python, and PHP.

The code-parsing module parses lines for feral concurrency control mechanisms that validate uniqueness before calling ORM APIs. For applications built on Ruby on Rails, the lines are declared in the model files. The validation lines usually contain keywords such as validate_uniqueness_of. For applications built on Django, the lines for feral concurrency control are right before calling APIs such as *save*, and they contain keywords of full_clean(). For applications built on Laravel, the lines are in the controller files with a syntax of "$validate_column_rules = ['unique':table_name]".

The code-parsing module divides the source code by request handlers and searches for sites where ORM-provided APIs are called. Currently, the APIs searched are *save* and *create* because they account for the majority of request races involving ORM APIs. The tool finds 3246 request handlers in 19 web applications that call *save* or *create*.

After collecting the three parsing results mentioned above, we manually check the results by applying the three rule rules extracted when studying bugs. If there are feral uniqueness validations on the application side but no unique keys on the database side, this is a bug where duplicate entries can happen in concurrent execution. If the API call sites modify object fields that are mapped to a column having unique constraints in the database, and there is no proper error handling, this is a bug that can crash the application in concurrent execution. If before the API call of *save* there is incrementing or decrementing column value on the application side, this is a bug causing wrong update results.

5.4.3 Checking Real-world Web Applications

We first evaluate the static checker on the set of request races used in this study. The static check can identify 23 modifying unique column values without proper exception handling, 2 having feral uniqueness validation without corresponding database unique constraints, and 5 incorrectly increment column values. If the static check had been used and the captured bug was fixed, 35% of the request races racing on database records in ORM web applications can be prevented.

We then run the static tool to check the latest stable version of 19 real-world open-source web applications built upon ORM frameworks. The static tool finishes within 5 seconds on a MacBook Pro laptop with 2.6GHz Intel Core i7 processor. Table 5.8 shows the results from the static tool.

Table 5.8: Tool Detection Results

Apps	Func No.	Rule1	Rule2	Rule3
Canvas-lms	586	42	4	5
Danbooru	82	9	9	0
Diaspora	79	2	1	1
Discourse	307	46	1	5
Gitlab	253	21	11	1
Openproject	103	3	4	0
Redmine	122	12	17	0
RoR ecommerce	89	4	3	1
Sharetribe	142	9	2	1
Spree	50	4	10	3
Django-lfs	140	2	-	38
Oscar	136	3	-	8
Posthog	139	23	0	1
Redash	3	0	-	0
Saleor	184	0	-	4
Sentry	339	48	-	0
Weblate	206	12	0	0
Zulip	191	33	0	5
Pixelfed	95	31	0	8
Total	3246	304	62	81

Bugs Related to Idempotence

The tool detects request races that cause the request handler to be not idempotent by using Rule1 and Rule2.

The tool finds 304 bugs using Rule1, which include 152 in Ruby on Rails applications, 121 in Django applications, and 31 in Laravel applications. In these 304 bugs, the request handler modifies object attributes that are mapped to unique keys in the database, but

proper exception handling code block is missing, and the application could crash under concurrent executions.

The bugs detected by Rule1 are those that miss exception handling. The tool does not check if the exception handling code block is correctly implemented. For example, a 'pass' statement in exception handling is not appropriate if the request handler before fixing returns some results. To prune such false negatives, we need to parse and analyze the exception handling code block to make sure the implementation is correct and can help enforce idempotence.

The tool finds 60 request races where there are feral uniqueness validations in some model files but there are no unique constraints on the corresponding columns in the database. All 10 Ruby on Rails web applications, three out of eight Django web applications, and one Laravel web application use feral uniqueness validation. For those applications that we do not find usage of feral uniqueness validation, a '-' is put in the table; If we find usage of feral uniqueness validation and corresponding database unique constraints exist, a '0' is put in the table. We do not find this type of request race from Django and Laravel, and all these 60 bugs are from Ruby on Rails applications.

The bugs detected by Rule2 are based on our inference of developer's expectations. Developers may define feral uniqueness validations by mistake, and those attributes may not need to be unique based on application logic. We let developers decide if detected request races are false positives.

Bugs Related to Concurrency Only

The tool detects 83 request races that will cause wrong results for concurrent updates, including 19 in Ruby on Rails applications, 56 in Django applications, and 8 in Laravel applications. We believe all these 83 are true positives because this pattern is similar to concurrency bugs in multi-threaded programs where a shared variable is read and then updated by two or more concurrent threads.

5.4.4 Limitations

The schema-parsing module only supports Ruby and SQL format schema files. The code-parsing module only supports the three programming languages, which are Ruby, Python and PHP. For schema files in other types and web applications in other programming

languages, the parsing module needs to be extended to support new syntaxes.

Our tool has possible false positives. Our tool does not take into account ad-hoc synchronizations that may be used by developers in the application source code. When ad-hoc synchronizations are used, the request handlers cannot execute concurrently, and duplicate key errors may not happen. Our rules are defined on *save* and *create*. While extending our rules to more ORM APIs, there could be false positives. Right now, we search *save* and *create* which do not handle exceptions in the API source code. APIs such as get_or_create in Django have exception-handling code blocks embedded in the API source code, and the application will not crash without app-side exception handling. To address the possible false positives, we need to look into more ORM API source code and build a whitelist of APIs that do not need app-side exception handling to avoid application crashes. The tool should not report a bug if the API that misses app-side exception handling is from the whitelist.

Our tool has possible false negatives. The static tool cannot detect bugs not covered by the three rules. For example, the tool cannot detect bugs that could cause voucher multiple redemptions. It is necessary to have a sophisticated understanding of the application business logic to extract rules for such bugs.

The current implementation of the tool is a combination of automation and manual work. The tool automatically parses application source code to get request handlers calling specified ORM APIs and code lines of feral uniqueness validations. It also automatically parses database schema files to get a map between the table name and unique key names.

The following are done manually. Rule1: For each request handler calling the specified ORM APIs, We start by checking what object calls the API. We check the results after parsing database schema files to find the corresponding unique keys. We check if the attributes of the object that are mapped to the unique keys are modified in the request handler and if exception handling is used. Apart from the rescue block in Rails, the except block in Python, and the catch block in PHP, ORM applications could implement methods for exception handling. For example, Canvas-lms provide a function unique_constraint_retry that handles the exception and retries the failed transaction. We manually determine if exception handling is used in the parsed request handlers. Rule2: Rails apps: For each model file that has feral uniqueness validations defined, We check if the corresponding table has unique keys defined on the attributes validated. Django: We search for sites where full_clean is called. We check what fields are modified in the object calling full_clean and if the modified attributes have corresponding unique keys defined in the database. Laravel apps: We search lines of validation rules which define a column need to be unique in table_-

name. We check if the column is defined as a unique key in the table. Rule3: We manually check if there is a pattern of reading object attribute value, then calling "+1", "-1", "+=", "-=" on an object attribute, and finally *save* is called.

5.5 Discussion

In this section, we discuss how our study results would benefit ORM framework API development and ORM web application developers.

5.5.1 Suggestions of ORM Framework API Development

Embedding exception handling within the API of an ORM framework is a beneficial practice. It allows developers to make the code more readable and reduce the need for scattered or duplicated error-handling code in application codebases. Developers can benefit from a consistent and predictable error-handling experience across different parts of the ORM framework. This simplifies error recovery, enables proper logging and auditing, and ensures that errors are appropriately propagated to the calling code for proper handling.

We found two bugs from applications built on Django that have contradictory fixing strategies. One bug from Zulip is fixed by adding exception handling after calling get_or_create API. One bug from Sentry is fixed by using get_or_create API without exception handling. By checking the source code of the latest Django framework, we noticed that Django API get_or_create handles the possible database exceptions. The Zulip bug is reported in 2012 and the Sentry bug is reported in 2021. We believe the Django framework includes exception handlings during these years to make their API more concurrency safe to use.

ORM framework should support more query syntaxes and consider the performance when implementing APIs. The support for a wide range of query syntaxes within an ORM framework is highly desirable. This allows developers to express complex queries more flexibly and intuitively, accommodating diverse database requirements and optimizing data retrieval.

We remove one bug from our previous study because we think it is not a race bug. Before applying the patch, developers used *create* from Ruby on Rails and catch and handle the ActiveRecord::RecordNotUnique exception. The bug report claims that the database “errors out when the draft is being created by multiple backends”. In the fixing patch, developers

use a manually written query INSERT ON CONFLICT DO UPDATE to replace the ORM API calls which “moves the new draft creation concurrency handling to PostgreSQL”.

From this bug, we can infer that the overhead of database-throwing errors cannot be ignored. ORM frameworks should take into account the performance issue when developing APIs, and provide more APIs to support syntaxes such as “INSERT ON CONFLICT DO UPDATE” to avoid efforts to manually write queries by developers. It will be easier for developers to maintain the web application if the ORM framework can provide more APIs that fit their needs.

5.5.2 Suggestions to Web Application Developers

Developers should be encouraged to utilize feral validations in their applications. Our study findings indicate that request races cannot be completely prevented when relying solely on feral uniqueness validations and app-side validations. This may raise the question of whether these approaches are still necessary. The answer is yes.

Under most isolation levels, when the database handles an INSERT query, internal locks are utilized to enforce ACID properties. However, a SELECT query typically does not request any locks under these isolation levels. Without employing feral uniqueness validations and app-side validations, the absence of locks on the database side can lead to increased lock contention and performance degradation. Furthermore, these approaches play a crucial role in preventing duplicate operations when requests are sent sequentially. Therefore, we believe it is still beneficial to incorporate feral uniqueness validations and app-side validations in web applications. These measures contribute to reducing lock contention, enhancing performance, and ensuring data integrity.

Additionally, developers should pay close attention to exception handling. Neglecting proper exception handling can result in application crashes and the loss of user input. Unlike locks, the use of exception handling does not adversely affect request handling throughput. Instead, it helps enforce the idempotence of request handlers, promoting stability and reliability in the application.

In summary, the adoption of feral validations, along with careful attention to exception handling, is essential in maintaining a robust and high-performing web application while ensuring data integrity and minimizing the risk of crashes.

CHAPTER

6

RELATED WORK

In this chapter, we provide a discussion of related work.

Reliability and Security in Web Applications 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 [40, 43, 59, 96, 97, 106, 122], but none of them handles the concurrency aspect. Some of these existing techniques handle the input generation problem, and ReqRacer complements these techniques by exposing the buggy-interleavings.

Techniques focusing on the security aspect of web applications have been proposed, e.g., auditing [77, 112], intrusion detection and recovery [52, 53], and identifying information disclosure [54]. Races are considered severe security vulnerabilities [35], and they can enable concurrency attacks [129]. 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 [47, 68, 69, 93, 105]. Some Android applications also have a client-server structure, and techniques developed in Chapter 3 could also be leveraged to handle races on their server-side applications.

Study of Concurrency Bugs in Multi-threaded Programs A lot of research efforts have been spent on races and concurrency bugs in multi-threaded programs. Researchers have conducted thorough empirical characteristic studies [61, 90] and the study results guide the development of tools for various purposes, e.g., bug detection [49, 50, 60, 71, 91, 107, 127, 128], program testing [67, 84, 99, 108, 113, 121], failure diagnosis [41, 42, 74, 76, 92], and fixing [73, 75, 87, 88]. Researchers have also proposed techniques targeting process races on the operating-system level [81] and distributed concurrency bugs in distributed and cloud systems [82, 83, 85, 86, 89]. As argued in our previous study [102], 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 [61] 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 [102]. Nevertheless, we follow their methodology and get inspiration from their study to conduct ours.

Study of Concurrency Bugs in Web Applications 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 [37, 48, 72, 94, 100, 104, 125] and fixing [36].

In the studies of concurrency bugs in web applications, while the one by us [102] 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 [57, 115]. However, the number of request races in web applications covered by these two studies is only 11 [102], 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 [101].

Study on error handling code Error handling is a critical aspect of software development, and incorrect or inadequate error handling can introduce vulnerabilities or unexpected behavior in an application. Gunawi et al. found that file systems and storage device drivers often do not correctly propagate error code [64]. error codes that are not checked or not further propagated. Yuan et al. studied failures in non-distributed systems [124]. They developed a static checker, Errlog, to find places where the error handling code is not logged and adds appropriate logging statements with low overhead. Yuan et al. studied catastrophic failures in production quality distributed systems [123]. They extracted simple

rules from bugs and developed a static checker, Aspirator, to locate these bugs leading to catastrophic failures.

In this work, our static tool can detect request races that could lead to application crashes in database-backed web applications built on ORM frameworks. These races can be fixed by adding an error-handling code. We also discuss how to implement the error-handling logic based on ORM APIs.

Program idempotence Idempotence holds immense significance in the realm of programming and has been extensively explored across various domains. By adhering to idempotent principles, developers can simplify system design, leading to cleaner and more manageable codebases. Idempotence fosters clarity and promotes ease of understanding, allowing developers to comprehend and maintain the system with greater efficiency. Furthermore, idempotence contributes to system robustness by providing resilience against unintended consequences or duplicate operations. It empowers developers to design functions, methods, and operations that can be executed multiple times without causing undesirable effects or producing inconsistent outcomes. Naik analyzed important properties, including idempotence, in distributed systems [95] and explained the requirements and compromises in distributed systems. Surbatovich et al. investigated non-idempotent behaviors caused by repeated I/O operations [111]. They provided characterization of input-dependent idempotence bugs, and developed tools for detecting such bugs at compile time and runtime. Zhang et al. took advantage of idempotence property to help software survive concurrency bugs during production run [126]. Based on an observation that reexecuting an idempotent region is sufficient to recover from many concurrency-bug failures, they built ConAir to run static analysis to identify potential failure sites and idempotent code regions, and insert recovery code around the identified idempotent code regions.

Our work revolves around introducing the concept of idempotence to database-backed web applications. In our research, we delved into the reasons why previous approaches fell short in guaranteeing idempotence. By thoroughly analyzing these shortcomings, we were able to extract valuable rules and insights from the bugs encountered during our investigation. These rules serve as a foundation for developing effective detection mechanisms, which are vital in ensuring the idempotence of database-backed web applications.

Web application code comprehension A web application is a multifaceted creation that integrates various technologies, including scripting languages, middleware, web services, data warehouses, and databases. Due to the intricate nature of these components, it becomes challenging for human developers to fully grasp the intricate relationships between

the modules. Recognizing this challenge, numerous studies and projects have emerged to assist developers in gaining a deeper understanding of web application code. Hassan et al. [65] proposed an approach to extract code structure and show the interaction between various components. WANDA [39] instruments web applications and combines dynamic and static information to support maintenance and evolution tasks. The integration [58] of WARE and WANDA combines static and dynamic analysis the further improve web application comprehension.

In this work, we focus on request races that happen under concurrent execution and invalidate request handler idempotence. The above-mentioned works are too general and cannot efficiently and effectively solve the request race problem.

CHAPTER

7

CONCLUSION

Modern websites commonly utilize web applications on the server side to handle incoming HTTP requests from users and generate dynamic responses. However, these web applications, by nature, operate concurrently and are susceptible to server-side request races. Request races refer to situations where multiple requests to the server compete for shared resources or execute in an interleaved manner, leading to unexpected and potentially erroneous outcomes. These races can occur when multiple users simultaneously interact with the web application or when a single user sends multiple requests in quick succession. The prevalence of web applications and the convenience offered by ORM frameworks exacerbate the problem of request races. As web applications become increasingly popular, the number of concurrent requests they handle grows, amplifying the likelihood of request races. Additionally, ORM frameworks provide developers with powerful tools for database interactions, but they also introduce complexities that can contribute to race conditions if not managed properly.

Addressing the issue of request races is crucial to ensure the reliability, consistency, and security of web applications. In this thesis, we conducted a comprehensive characteristic study on request races, encompassing various types of web applications: those built using

raw-SQL, ORM-based frameworks, and Node.js. By examining a diverse range of web applications, we aimed to gain a deeper understanding of the nature and occurrence of request races in different development contexts. Furthermore, we delved into the effects of request races and explored potential fixes for mitigating their impact. By thoroughly analyzing the consequences of request races, we sought to identify effective strategies and best practices for addressing and resolving them. A significant aspect of our study focused on the idempotence property to request races. Idempotence is a crucial concept in ensuring the reliability and consistency of web applications. We specifically investigated how request races can impact the idempotence of request handlers and provided valuable guidance to developers on how to build idempotent request handlers effectively.

Based on the study results, we built two race detection tools. The first tool, called REQRACER, is a dynamic detection tool specifically designed for scope-based request races in raw-SQL web applications. By leveraging dynamic analysis techniques, REQRACER is capable of detecting and highlighting areas within the codebase where request races can occur. This tool aids developers in identifying potential race conditions. The second tool is a static detection tool tailored for request races in ORM-based web applications, with a specific emphasis on enforcing idempotence. This tool utilizes static analysis techniques to analyze the codebase and identify potential areas where request races can manifest. It pays particular attention to ensuring the idempotence property of request handlers, which is crucial for maintaining consistent and reliable application behavior.

Future work in the field of request race detection can focus on enhancing the efficiency and effectiveness of existing tools. For instance, in the case of REQRACER, the modeling of happens-before relationships can be further extended to improve the accuracy and speed of request race detection. By refining the modeling techniques, the tool can provide more precise results while reducing false positives. In addition, the static detection tool used for ORM-based web applications can be expanded to support a broader range of rules. By incorporating more rules, such as those related to invariants modeling voucher redemption, the tool will be able to detect a wider variety of request races. This expansion enables developers to identify potential issues and race conditions in different areas of their codebase, improving the overall effectiveness of the tool.

Furthermore, it is important to raise awareness about the severity and significance of request races within both the research community and industry. By disseminating the findings and insights from our study, we aim to highlight the importance of addressing request races in web application development. This increased awareness can lead to more

focused research efforts, as well as encourage industry practitioners to prioritize race detection and mitigation strategies in their development processes. By leveraging the characteristic results obtained from our study, developers can gain valuable insights into the common causes and patterns of request races. Armed with this knowledge, they can proactively avoid writing code that may lead to such race conditions, enhancing the overall robustness and reliability of their applications. Additionally, the results regarding fixing request races provide developers with guidance on how to quickly and effectively resolve race-related issues when they arise.

REFERENCES

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

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

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

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

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

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

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

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

[9] [n.d.]. DNN Platform Issue Tracker. https://dnntracker.atlassian.net.

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

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

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

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

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

[15] [n.d.]. Moodle Tracker. https://tracker.moodle.org/.

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

[17] [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.

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

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

[20] [n.d.]. OpenMRS. http://openmrs.org/.

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

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

[23] [n.d.]. Pixelfed - Photo Sharing. For Everyone. https://github.com/pixelfed/pixelfed.

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

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

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

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

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

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

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

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

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

[33] [n.d.]. WordPress Trac. https://core.trac.wordpress.org/.

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

[35] 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.

[36] 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

[37] 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

[38] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 67–78.

[39] Giuliano Antoniol, Massimiliano Di Penta, and Michele Zazzara. 2004. Understanding web applications through dynamic analysis. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004*. IEEE, 120–129.

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

[46] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[47] 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

[48] 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

[49] 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

[50] 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

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

[52] 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

[53] 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

[54] 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

[55] 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.

[56] 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

[57] 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

[58] Giuseppe A Di Lucca and Massimiliano Di Penta. 2005. Integrating static and dynamic analysis to improve the comprehension of existing web applications. In *Seventh IEEE International Symposium on Web Site Evolution*. IEEE, 87–94.

[59] 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

[60] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Pro-*

gramming Language Design and Implementation (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[61] P. Fonseca, Cheng Li, V. Singhal, and R. 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

[62] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2011. 2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs. 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, 239–250. https://doi.org/10.1145/1950365.1950394

[63] 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

[64] Haryadi S Gunawi, Cindy Rubio-González, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct.. In *FAST*, Vol. 8. 1–16.

[65] Ahmed E Hassan and Richard C Holt. 2002. Architecture recovery of web applications. In *Proceedings of the 24th International Conference on Software Engineering*. 349–359.

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

[67] 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

[68] 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

[69] 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

[70] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. 2016. Automatically Verifying and Reproducing Event-based Races in Android Apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 377–388. https://doi.org/10.1145/2931037.2931069

[71] 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

[72] 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

[73] 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

[74] 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

[75] 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

[76] 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

[77] 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

[78] Martin Kleppmann. 2017. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.".

[79] 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

[80] 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

[81] 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

[82] 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

[83] 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

[84] 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

[85] 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

[86] 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

[87] 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

[88] 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

[89] 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

[90] 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

[91] 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

[92] 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

[93] 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

[94] 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

[95] Nitin Naik. 2021. Demystifying properties of distributed systems. In *2021 IEEE International Symposium on Systems Engineering (ISSE)*. IEEE, 1–8.

[96] 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

[97] 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

[98] 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

[99] 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

[100] 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

[101] 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

[102] 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

[103] 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 *Proceedings of the 19th International Conference on Mining Software Repositories*. 744–756.

[104] 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

[105] 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

[106] 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

[107] 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

[108] 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

[109] 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

[110] 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

[111] Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2019. I/O dependent idempotence bugs in intermittent systems. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–31.

[112] 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

[113] 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

[114] 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

[115] 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

[116] Todd Warszawski and Peter Bailis. 2017. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 5–20.

[117] ANSI X3. 1992. American National Standard for Information Systems- Database Language-SQL.

[118] 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

[119] 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

[120] 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

[121] 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

[122] 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

[123] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems.. In *OSDI*, Vol. 10. 2685048–2685068.

[124] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: Enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th* {*USENIX*} *Symposium on Operating Systems Design and Implementation (*{*OSDI*} *12)*. 293–306.

[125] 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

[126] Wei Zhang, Marc De Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. 2013. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. 113–126.

[127] 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

[128] 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

[129] 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

[130] 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