RACEDB: Detecting Request Race Vulnerabilities in Database-Backed Web Applications

An Chen
University of Georgia
an.chen25@uga.edu

Yonghwi Kwon
University of Maryland
yongkwon@umd.edu

Kyu Hyung Lee
University of Georgia
kyuhlee@uga.edu

***Abstract*—Request race vulnerabilities in database-backed web applications pose a significant security threat. These vulnerabilities can lead to data inconsistencies, unexpected behavior, and even unauthorized access. Existing automated detection techniques often fall short due to the complexity of race conditions and the intricate interplay between application logic and database interactions.**

This paper introduces RACEDB, a novel system that tackles these challenges through two key innovations. Application-aware Request Race Detection (ARD) provides a comprehensive analysis of data dependencies, considering not only the database query but also the application code. This allows RACEDB to identify subtle race conditions that might be missed by existing approaches. Furthermore, RACEDB employs an automated verification technique using replay-based execution. This technique efficiently isolates true races from false positives and generates definitive exploits for verified vulnerabilities. We evaluated RACEDB on a dataset of 14 real-world PHP web applications. The results demonstrate the effectiveness of RACEDB compared to existing tools. RACEDB achieved a superior detection rate, identifying 21 known vulnerabilities and discovering 18 new vulnerabilities, significantly exceeding the performance of existing tools while also achieving a lower rate of false positives. Finally, we responsibly reported all newly discovered vulnerabilities to the corresponding developers, and 7 of them have been assigned CVE IDs.

1. Introduction

Modern web applications heavily rely on databases to store and manage data. These applications receive requests from remote users as input, process the requests, and update the database accordingly. Remote requests can occur concurrently, which can lead to unpredictable and inconsistent behavior if not handled properly. Race conditions, deadlocks, and database corruption are potential consequences of inadequate concurrency control. Several prominent incidents highlight the criticality of addressing race conditions. For instance, vulnerabilities in Instacart, Starbucks, Flexcoin, and GitLab led to issues like double coupon redemption [33], duplicate balance transfers [68], wallet overdraw [21] that lead Flexcoid bankruptcy, and account hijacking [15]. The GitLab account hijacking [15] in 2023 affects multiple versions of the application. This vulnerability allows attackers to exploit the request race condition to forge verified emails and potentially take over third-party accounts when GitLab is used as an OAuth provider [15]. The attack leverages the lack of proper synchronization during the email verification process, enabling malicious actors to intercept and manipulate the verification flow, leading to unauthorized access to user accounts.

These incidents underscore the critical importance of addressing request race conditions in web applications to prevent similar high-impact failures. Hence, testing web server applications to identify and fix concurrency bugs is critical to building secure web services. Various automated concurrency bug-finding techniques exist. Many of them focus on exploring different interleavings between executions (e.g., threads) to identify specific sequences of interleavings causing inconsistent behaviors. For example, a large body of existing work mainly explores different thread schedules by testing various permutations of threads via customized schedulers or delay injection [12], [20], [24], [40], [43], [69].

In web server applications, interleavings typically occur between request handlers. If resources like variables or database content are not properly protected during request handling, concurrent access can corrupt their values. Unlike local program variables, database content can be accessed and modified by any execution, leading to significantly more potential content states to consider. In addition, recent studies have introduced novel request race attacks, such as the Timeless Timing Attack [23], which manipulates shared resource timing to exploit race conditions without depending on network latency. Additionally, the Single-Packet Attack [38] leverages HTTP/2 multiplexing to bundle multiple requests into a single TCP packet, ensuring simultaneous processing and exposing race vulnerabilities in web applications. These approaches demonstrate the evolving sophistication of request race exploitation across modern web systems. This is further compounded by complex database queries and diverse user requests. Existing techniques often struggle with this complexity, as database content plays a crucial role in race conditions but has received less attention in prior research.

In this paper, we propose RACEDB, that systematically analyzes database-backed web server applications for

race condition issues. RACEDB goes beyond existing approaches [39], [59], [71] by analyzing dependencies not only between database fields or tables but also within the application context. It identifies dependencies introduced by the web application logic, such as inter-table dependencies mediated by application variables. This comprehensive analysis empowers RACEDB to detect silent data corruption within the database that could be missed by conventional methods.

Utilizing the identified dependencies, RACEDB employs a graph-based race detection algorithm [71] to detect potential race conditions involving user requests. To further refine the analysis and reduce false positives, RACEDB offers automated verification capabilities. This verification phase leverages a replay execution technique to isolate true races among the identified candidates. As a result, RACEDB can generate definitive exploits that demonstrate the vulnerabilities, leading to a higher accuracy in identifying and addressing real request race vulnerabilities.

We demonstrate the effectiveness of RACEDB by conducting a comprehensive study using a dataset of 14 real-world web applications. We compared RACEDB against existing tools [39], [59] in terms of both detection capability and false positive rates across all applications, where RACEDB consistently outperforms existing tools across all assessed applications. Specifically, RACEDB successfully identified a total of 39 request race vulnerabilities within the 14 applications. Among the identified vulnerabilities, 18 were previously unknown. The new vulnerabilities were confirmed by developers, and 7 of them have already been assigned CVE IDs.

Our contributions are summarized as follows:

- We propose RACEDB, an automated system designed to detect and verify request race vulnerabilities, including intricate and silent race in database-backed web applicaitons.
- We introduce Application-aware Request Race Detection (ARD), which can identify data dependencies within the web application and the database queries (e.g., inter-table dependencies through application variables).
- We present an automated verification technique that utilizes replay-based execution. This approach effectively detects divergences between serialized and concurrent executions, significantly reducing false positives. Additionally, it provides comprehensive information for verified races, enabling deeper analysis and facilitating the reproduction of the race conditions.
- Our evaluation of 14 real-world PHP web applications shows that RACEDB outperforms existing state-of-the-art techniques. RACEDB successfully detects 21 known request race cases and 18 new cases, whereas existing tools detect only 13 known cases and 6 new cases.
- We responsibly reported 18 of new vulnerabilities from 6 applications to the corresponding developers and 7 of them have been assigned with CVE numbers.
- We will release RACEDB source code and data at https://github.com/sscf224/racedb.

Threat Model. Request race vulnerabilities arise when concurrent requests lead to unintended behaviors or data states due to timing issues. Assets at risk include user data, such as personal information, payment details, and credentials, as well as the overall system integrity. Adversaries comprise external attackers who exploit race conditions to gain financial profit or manipulate data, and benign users who may accidentally trigger the request race and suffer financial loss or encounter unexpected application behavior.

Attack vectors include sending multiple simultaneous requests to the same or different API endpoints that modify the same data entity through application GUI or manually crafted. Security assumptions include the possibility that adversaries might have valid credentials and that basic network security measures like HTTPS are in place, focusing on application-level vulnerabilities. Potential impacts involve data corruption due to inconsistent data states, and unauthorized transactions leading to overpayment or duplicates.

2. Motivating Example

We use a request race vulnerability found by RACEDB on CE Phoenix Cart [3], an open-source web e-commerce application, to illustrate how RACEDB detects and verifies the vulnerability, where existing tools failed.

Vulnerable Code under Testing. Figure 1-(a) shows simplified code snippets for processing order with a coupon. A request race can occur if two users use the same coupon which should be used only once *simultaneously*.

The `SELECT` query at line 1 counts the number of rows in the `customers_to_discount_codes` table that record the coupon usage (at line 10). Then, at line 3, if the coupon is *never used* (i.e., the first query returns no record), it sets `$coupon['max_usage']` to '`0`', so that the `if` branch at line 6 takes the true branch. At line 6, it checks whether the coupon can be used by checking the following two conditions: (1) the coupon has unlimited usage (i.e., `$coupon['max_usage']` is zero), or (2) the coupon's current usage record is less than its usage limit (i.e., (`$coupon['total_usage']`<`$coupon['max_usage']`). If the coupon can be used, it checks other conditions, such as whether the coupon has not expired and whether the current order satisfies the minimum order amount to use the current coupon by running a `SELECT` query with a long `WHERE` clause at line 7. At line 8, it checks whether the query returns a row (i.e., whether there is a coupon that satisfies the conditions). If so, it obtains the discount code's id at line 9 and then records the coupon's usage at line 10 via an `INSERT` query.

Typically, after a coupon is successfully used, its usage is recorded at line 10, which is checked at line 1 to prevent a coupon from being used more than its use limit (i.e., `$coupon['max_usage']` at line 6). However, a request race can happen if two concurrent requests execute the queries at lines 1, 7, and 9 before one of the requests executes the query at line 10 (i.e., executing $1_A \rightarrow 7_A \rightarrow 9_A \rightarrow 1_B \rightarrow 7_B \rightarrow 9_B \rightarrow 10_A \rightarrow 10_B$ where the subscripts represent the two different requests A and B).

```
1  $count = tep_db_query("SELECT count(*) AS total_usage, dc.max_usage FROM discount_codes dc, customers_to_discount_codes c2dc
                          WHERE dc.discount_codes_id = c2dc.discount_codes_id AND
                                dc.discount_codes='".$_SESSION['sess_discount_code']."'");
2  if (mysqli_num_rows($count) == 0)
3     $coupon['max_usage'] = 0;
4  else
5     $coupon = $count->fetch_assoc();

6  if ($coupon['max_usage'] == 0 ? 1 : ($coupon['total_usage'] < $coupon['max_usage'] ? 1 : 0)) {
7     $condition = tep_db_query("SELECT * FROM discount_codes WHERE discount_codes = '%s' AND
                                 IF(expires_date='0000-00-00', date_format(date_add(now(), ...), '%%Y-%%m-%%d'), expires_date)
                                 >= date_format(now(), '%%Y-%%m-%%d') AND minimum_order_amount <= '%s' AND status = '1'");
8     if (mysqli_num_rows($condition) != 0) {
         ...
9        $codes = tep_db_query("SELECT discount_codes_id FROM discount_codes
                                WHERE discount_codes='".$_SESSION['sess_discount_code']."'");
10       tep_db_query("INSERT INTO customers_to_discount_codes(customers_id, discount_codes_id)
                                VALUES ('".$_SESSION['customer_id']."', '".$codes->fetch_assoc()."')");
```

(a) Program Source Code For Order Processing

```
A1  SELECT count(*) AS total_usage, dc.max_usage
       FROM discount_codes dc, customers_to_discount_codes c2dc
       WHERE dc.discount_codes_id = c2dc.discount_codes_id
             AND dc.discount_codes='CODE24'
A2  SELECT * FROM discount_codes WHERE discount_codes='CODE24'
                                       AND ...
    ...
A3  SELECT discount_codes_id FROM discount_codes
       WHERE discount_codes='CODE24'
A4  INSERT INTO customers_to_discount_codes(customers_id,
                discount_codes_id) VALUES ('2', '3')
```

(b) Request A Query Trace

```
B1  SELECT count(*) AS total_usage, dc.max_usage
       FROM discount_codes dc, customers_to_discount_codes c2dc
       WHERE dc.discount_codes_id = c2dc.discount_codes_id
             AND dc.discount_codes='CODE24'
B2  SELECT * FROM discount_codes WHERE discount_codes='CODE24'
                                       AND ...
    ...
B3  SELECT discount_codes_id FROM discount_codes
       WHERE discount_codes='CODE24'
B4  INSERT INTO customers_to_discount_codes(customers_id,
                discount_codes_id) VALUES ('1', '3')
```

(c) Request B Query Trace

Figure 1: Motivation Example

Scenario Triggering the Request Race. A malicious user first logs on to the service running CE Phoenix Cart with two browsers and two different accounts. Then, the user proceeds to the payment page, which allows the user to redeem the coupon. On the payment page, the user puts the same coupon code which can be only used once. Then, the user confirms the order (with the coupon) on the two browsers *at the same time* to cause the request race.

Figure 1-(b) and (c) show the query trace of the two requests A and B when the request race happens. The query A1 and B1 (line 1), in this example, would return the same value before executing the query A4 or B4 (line 10). As a result, both requests are processed successfully, allowing the malicious user to use the coupon twice.

Assume that the two requests are *not* concurrently executed, meaning that the entire request A is completed before the request B. Then, a usage record inserted by the query A4 will make the query B1 return a row inserted by A4, resulting the `$coupon['total_usage']` to be 1. Since `$coupon['max_usage']` is 1 (i.e., the coupon can be used once), the second condition at line 6 is not satisfied, preventing the coupon from being over-used.

Existing Request Race Detection. Existing techniques [22], [39], [54], [59], [71], [76] mainly focus on detecting races on database queries operating on the same database field such as concurrent requests reading and writing the same field of a table. Hence, they miss this request race as the race is caused between different fields: `count` and `discount_codes_id`. Specifically, we further explain how the two of the most recent approaches, Raccoon [39] and ReqRacer [59], miss the race. In particular, it is challenging to identify that the first query (line 1) and the last query (line 10) can be the target of request race, as their relationships are expressed in two different ways. First, at line 1, the `WHERE` clause and the `count` operation together indicates the relationship between the two tables `discount_codes` and `customers_to_discount_codes`. Second, at lines 9 and 10, the relationship between the two tables are established by the `discount_codes_id` from the `discount_code` table being used in the `INSERT` query at line 10. Both Racoon and ReqRacer miss them because (1) they operate on the SQL traces, which do not include the concrete values of the `WHERE` clause (at line 1), and (2) they target values appearing across multiple queries to identify queries potentially causing races while the queries A1 and A4 use two different values to indicate the same coupon, `discount_codes` and `discount_codes_id` respectively.

RaceDB on the Motivating Example. RaceDB leverages its concolic execution engine to identify (1) inter-table relationship between the `discount_codes_id` fields of the `discount_codes` and `customers_to_discount_codes` tables, and (2) the dependency between the first query's return (`$count`) and the conditional statements at line 6. The dependencies suggest to create a database with the `discount_codes` table with `discount_codes` equal to the coupon's code stored in the session variable, as well as database items with the same `discount_codes_id` in the

two tables[1]. More importantly, RACEDB identifies another inter-table relationship that the number of rows returned from the first query (i.e., `total_usage`) should be less than the `max_usage` in the same query, in order to take the true branch at line 6. Furthermore, to reach the vulnerable query at line 10, the database must satisfy the `SELECT` query at line 7 with the condition at line 8. RACEDB synthesizes the corresponding database item by examining the `WHERE` clause at line 7. In lines 9 and 10, RACEDB discovers the inter-table dependency between the `discount_codes` and `discount_codes_id` from the query return at line 9 being used in line 10's query construction.

To this end, RACEDB identifies that the program conditions and database operations (e.g., `SELECT` and `INSERT` queries) are all dependent on the first query returning `$count`. Hence, RACEDB marks the '`count(*)`' field at line 1 as a sensitive field, meaning its value should not diverge between different interleavings. RACEDB identifies the following two executions resulting in a different values at the end of the execution.

- $1_A \rightarrow 7_A \rightarrow 9_A \rightarrow 1_B \rightarrow 7_B \rightarrow 9_B \rightarrow 10_A \rightarrow 10_B$, resulting in 2.
- $1_A \rightarrow 7_A \rightarrow 9_A \rightarrow 10_A \rightarrow 1_B$, resulting in 1.

With the executions leading to different values on the sensitive field, RACEDB confirms it as a true positive.

3. Design

Request race vulnerabilities arise from various non-determinism occurring during concurrent execution. As there are many sources of non-determinism and their many combinations, static analysis tools [36], [46], [66], [76] are ineffective in identifying and detecting request races. As a result, there exist techniques leveraging traces collected from runtime executions [39], [54], [59], [71]. While they advance the state-of-the-art, they rely on data gathering and lack the dynamic analysis capabilities necessary to identify and reason the program execution and database states, missing various potential request race vulnerabilities.

Objective. RACEDB automatically detects and verifies request race vulnerabilities in database-backed web server applications. RACEDB aims to solve three fundamental challenges. First, database-backed web server applications often have complex dependencies on database contents, preventing execution from reaching the code blocks vulnerable to request races. RACEDB analyzes program and database states to generate a database that can reach the vulnerable code. Second, there exist tables and fields that are closely related, such as storing identical values or related values, which, if not correctly operating in a concurrent execution, can cause a request race. RACEDB comprehensively analyzes various dependencies in the program or query language logic to reveal such inter-table dependencies. Third, existing techniques [39], [71] often produce many false positives, requiring substantial manual effort in testing and validating the race candidates, preventing them from being practical. RACEDB implements a replay-based validation technique to automatically identify true positive request races along with a concrete input and a database.

Overview. Figure 2 presents the high-level system overview of RACEDB. First, the trace generation component leverages a concolic execution engine to explore the execution paths of a target application and identifies the required program and database states (§ 3.1). Second, the application-aware request detection component constructs an Application-aware Request Race Detection (ARD) graph by analyzing execution traces (§ 3.2). Then, it runs a request race candidate detection algorithm to identify race candidates for testing with the ARD graph. Third, the candidates are passed to the race verification component that compares the serialized and concurrent executions to detect divergences between the executions to identify true positive request races from the candidates (§ 3.3).

3.1. Trace Generation via Concolic Execution

Concolic Execution Engine. Recently, SynthDB [10] proposed a database synthesization technique to aid database-backed web applications. Their technique is based on a concolic execution engine developed for PHP applications. We obtained the implementation of SynthDB from the authors and utilized it as a foundation for RaceDB. Specifically, we used their concolic execution engine to generate query traces and employed a modified version of SynthDB to generate synthesized database states, allowing us to reach the code base related to request races.

We made a few changes to the SynthDB. First, we modify its concolic execution engine to generate a database that can fulfill a given remote request successfully. In other words, to complete a request, the database should include all the values that would satisfy all the path conditions during the execution of the request. Specifically, while SynthDB aims to create a single database that maximizes code coverage, RACEDB generates multiple databases for each request's execution path, where each path executes multiple database queries that might cause a request race. Second, we enhance SynthDB's dependency analysis between the queries. Specifically, RACEDB focuses on tracking dependencies between `SELECT` queries and `UPDATE` or `INSERT` queries, that are essentially database read and write operations. RACEDB enhances SynthDB to support complex dependencies between multiple tables expressed in `WHERE` clauses and values passed between the queries via program variables. For example, in Figure 1-(a) at line 1, the `WHERE` clause's highlighted condition indicates the two tables are closely related. In addition, at lines 9 and 10, the discount code's id returned from the `SELECT` query is used in the `INSERT` query, revealing the relationship between the two tables. With the above dependencies, RACEDB can identify the `SELECT count(*)` and the `INSERT` queries at lines 1 and 10 can be a request race candidate.

Database Operations. RACEDB extracts database read and write operations issued during execution from the execution traces. For each database operation, RACEDB records

1. The `discount_codes` and `customers_to_discount_codes` tables.

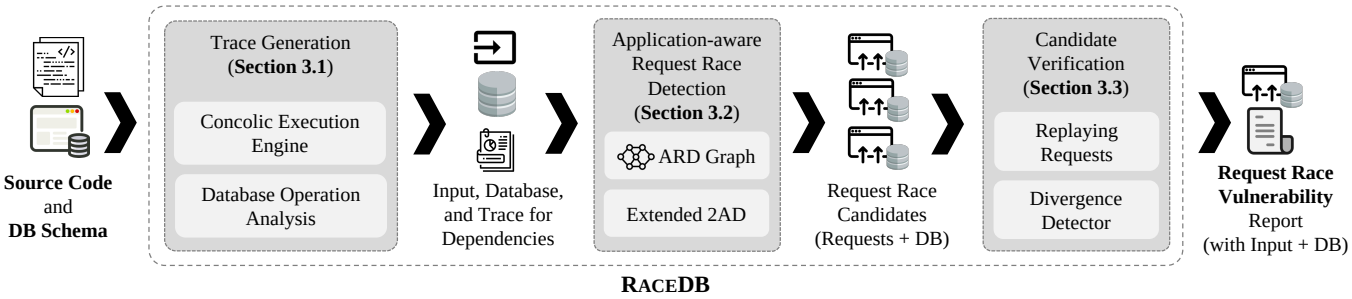

Figure 2: System Overview of RACEDB.

the database fields—such as tables and columns—affected by the operation. Analyzing the program dependencies related to the database operations (e.g., program statements and queries dependent on `SELECT` query's return values) allows RACEDB to identify implicit inter-table relationships. For example, in Figure 1-(a), the `SELECT` query return value at line 9 is eventually used during the construction of the `INSERT` query at line 10. This inter-table usage, including the `WHERE` clause at line 9, implies three related database field pairs: (1) `discount_codes_id` and (2) `discount_codes` of `discount_codes` and (3) `customers_to_discount_codes.discount_codes_id`.

Note that identifying interdependent tables extends the search space of request race candidates. Specifically, previous approaches focus on finding races between accesses to the same table and field, while RACEDB discovers the interdependent tables and includes them in the search space of request races.

Trace Generation Outcomes. This step has three outputs: (1) synthesized remote input such as `$_POST` and `$_GET` values, (2) a synthesized database, and (3) a trace for database operations and the corresponding affected database fields, including revealed inter-table usages. The outputs will be used to identify relationships between the database operations that can potentially cause request races in § 3.2.

3.2. Application-aware Request Race Detection

This section introduces our application-aware request race detection algorithm. Previous studies have proposed request race detection algorithms based on dependency graphs [39], [59], [71]. In particular, they focus on database traces and request history to identify dependencies between database queries and detect potential race conditions. Unfortunately, they do not consider application logic, such as dependencies introduced within the web application and the database queries (e.g., inter-table dependencies through application variables), missing various request races.

Graph Construction. To construct the application-aware request race detection (ARD) graph, we collect a set of concrete execution traces generated by concolic execution. The ARD graph is a finite multigraph, allowing multiple edges between the same pair of nodes. An ARD graph consists of two types of nodes, operation nodes and request nodes, and two types of edges, *r-w* edge and *w-w* edge. An

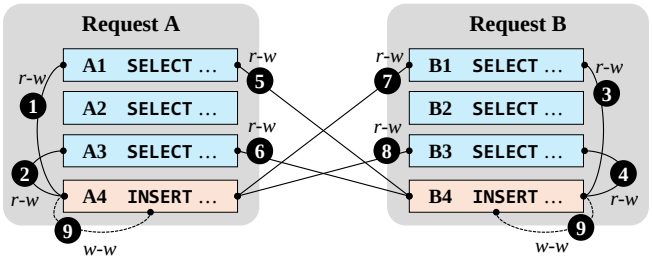

Figure 3: ARD graph generated the motivation example.

operation node represents a database operation (e.g., an SQL query), and a request node represents a request received by the application. A request node acts as a supernode, encapsulating all the operations executed within that request. Both *r-w* and *w-w* edges are undirected edges. The *r-w* edge represents data dependencies between two operations where one of them is a write operation, while the *w-w* edge shows data dependencies between two write operations.

To create an ARD graph, we first create an operation node for each database operation and a request node for each request execution. Then, we group the operations by execution of a request. We consider two database operations to be a potential request race candidate if they access the correlated data field (i.e., the same table/column or a relationship identified through application-level data dependencies) and, at least one of the operations is a modification (i.e., a write). For each identified potential race between two operations, we create an *r-w* edge if one is a read and the other is write, and add an *w-w* edge if both are writes.

Figure 3 shows an ARD graph generated from the motivating example § 2. There are two requests (Request `A` and `B`) in this example, where each request containing four database operations (`A1`∼`A4` and `B1`∼`B4`). Thus, the graph contains 2 request nodes and 8 operation nodes. Next, RACEDB identifies the data dependencies between the database operations (i.e., queries).

Specifically, we derive an *r-w* edge between `A1` and `A4` (❶) through two dependencies: (1) the inter-table dependency between the `discount_codes` and `customers_to_discount_codes` tables through the `discount_codes_id` field in `A1`, and (2) the dependency through the return value of `A1` (i.e., the `count(*)` query on the two tables) being dependent on the number of records

inserted by A4. Next, as discussed in § 2, we introduce an *r-w* edge (❷) from the identified data dependency between queries A3 and A4 through the application variable $codes at line 9 in Figure 1. Note that the dependencies between A1 and A4 as well as between A3 and A4 exist in the B1, B3, and B4, introducing the *r-w* edge❸ and ❹. Moreover, between two requests, there exist cross request dependencies such as between A1 and B4 as well as between A3 and B4, introducing the edges ❺ and ❻. Also, the vice versa holds, adding the edges ❼ and ❽. Lastly, as previously discussed, all write operations have a self-loop *w-w* edge (❾).

Extending 2AD Algorithm for Race Detection. To identify potential race conditions within request sequences, we leverage the Abstract Anomaly Detection (2AD) algorithm [71] with a slight extension. Unlike traditional approaches relying on concrete concurrent traces [4], 2AD generalizes the reasoning of potential races. It analyzes collected serial requests to determine if vulnerabilities could arise from potential concurrent execution scenarios. 2AD reasons about the space of possible concurrent interleavings by analyzing a finite graph representing a given trace.

RACEDB extends the concept of edges in the 2AD algorithm by incorporating application-level dependencies, such as inter-table relationships. This allows us to capture a richer context for potential request races. Fortunately, the core race detection algorithm from 2AD remains directly applicable to our enhanced detection graph. Therefore, we leverage 2AD's algorithm to identify race candidates within our constructed detection graph. For a detailed explanation of the algorithm, we refer readers to the original paper [71].

Detecting Cycles in Request Nodes. We apply the 2AD algorithm to identify request race candidates. Specifically, we identify the cycles between the request nodes in an ARD graph. For example, from the motivating example (Figure 3), we check whether there are edges forming cycles between the request A and B nodes, by following the steps below. First, RACEDB randomly selects the query A4 and attempts to build a cycle between request nodes (i.e., Request A and B), which can indicate a potential request race. We can then traverse edges (❼-❽), (❻-❽), (❺-❻) or (❺-❼) to form an inter-request cycle between the request A and B, resulting in a candidate pair. Note that the cycle (❺-❼) is the root cause of the request race case discussed in § 2. Additionally, we can traverse edge (❶ or ❷) and the dotted self-edge (❾) to form an intra-request cycle, indicating that two requests A can also race with each other, making them another candidate. In the end, we can identify three request race candidates: (1) Request A and B, (2) Two requests A, and (3) Two requests B. Along with the edges used to form the cycle, these candidates will be examined in the next phase of candidate verification.

3.3. Candidate Verification with Replay Execution

Unlike Raccoon [39] and ReqRacer [59], which focus on database access patterns or rely on error messages for verification, RACEDB employs an automated approach based on detecting execution divergence across different interleavings. Specifically, RACEDB uses an automated technique to verify race candidates identified by ARD graph. RACEDB achieves this by executing each candidate race in both serialized and concurrent manners. We then monitor each execution to detect *divergences* across the executions to detect a request race. The divergences can include differences in the following data:

- **Database State:** Inconsistencies in the actual data stored in the database after serialized and concurrent executions are a clear indicator of a race condition. However, comparing complete database states (including all tables and fields) can lead to false positives due to fields that are not relevant to the potential race. This can occur due to non-determinism (e.g., random value-involved fields) or values dependent on timing (e.g., timestamp-related fields), changing values regardless of the race condition. To avoid such false positives, we leverage the data-dependence analysis results obtained during the construction of the ARD graph. RACEDB focus only on database fields that are data-dependent on or may modified by one or more other writing operations, identified by the edges in the detection graph. Values of these fields are directly affected by race conditions and provide a more targeted comparison for identifying true races.
- **Application State:** This encompasses variations in the application's internal state, such as error messages generated during execution or application crashes.
- **Database Access Patterns:** Divergences in how the application accesses the database during serialized and concurrent executions can also signify a race condition. For instance, if a serialized execution performs a single read operation, while a concurrent execution performs multiple reads followed by a write, this difference in access patterns could lead to inconsistencies.

Replaying Serialized Requests. For each pair of requests (r1, r2) identified as a race candidate, RACEDB prepares the required database state to replay the requests. To collect results from an execution *without* a request race, RACEDB replays the requests by serializing each request's execution. Specifically, it first executes the request r1 and waits until it finishes. Upon r1's completion, it executes the request r2 (r1→r2). Additionally, RACEDB collects results from the (r2→r1) execution order as well. This process results in two serialized executions generating two database states D_1^s and D_2^s: D_1^s from r1→r2 and D_2^s from r2→r1.

Replaying Concurrent Requests. Now, RACEDB aims to try all possible interleavings of the database operations in r1 and r2. Specifically, RACEDB controls the execution of individual database operations in r1 and r2 to examine all possible interleavings between operations identified as a cycle in the graph. To control the order of database operations (i.e., queries), RACEDB leverages a library called ProxySQL [58], which can insert delays in each query's execution. We assign delays to each query to enforce these interleavings during replay.

For example, consider the motivating example where

RACEDB identified three request race candidates: (1) Request A and B, two request As, and two request Bs. Focusing on the race candidate involving requests A and B, the detection graph contains multiple cycles (represented by edges (❼-❽), (❻-❽), (❺-❻), or (❺-❼). For each cycle, a limited number of interleavings exist between the involved operations. Take the cycle formed by edges (❺ ❼). The operations involved are A1, A4, B1, and B4 and the possible interleavings include: (A1, B1, A4, B4), (B1, A1, B4, A4), (A1, B1, B4, A4), and (B1, A1, A4, B4). We can exclude (A1, A4, B1, B4) and (B1, B4, A1, A4) as they are equivalent to serialized execution. Additionally, (A1, B1) and (B1, A1) involve only read operations. Hence, swapping their order will not introduce races. By executing each of these valid interleavings, RACEDB obtain four database states from the concurrent executions: D_1^c, D_2^c, D_3^c, and D_4^c.

Finally, we compare these database states from the concurrent executions ($D_{1\sim4}^c$) with the database states from serialized executions ($D_{1\sim2}^s$). If D_1^c and D_2^c matches one of the database states from the serialized executions ($D_{1\sim4}^s$), it indicates no race occurred. However, if any one of $D_{1\sim4}^c$ diverges from both of D_1^s or D_2^s, it suggests the execution order led to a race condition.

In the motivating example, RACEDB successfully detected such a divergence between serialized and concurrent database states. Consequently, it reports the race to the user, providing complete information for reproduction, including requests involved, database states, and exact order of database query executions.

4. Evaluation

This section details the evaluation of RACEDB's effectiveness in detecting request race vulnerabilities. In § 4.1, we describe the evaluation methodology, including the dataset of real-world web applications, collecting reported vulnerabilities, and the configuration of compared tools. § 4.2 evaluates RACEDB's ability to identify vulnerabilities compared to existing tools (e.g., Raccoon [39], ReqRacer [59]). To illustrate RACEDB's capabilities in detail, § 4.3 discusses two specific vulnerabilities detected by RACEDB. § 4.4 analyzes false positives reported by RACEDB and compares them to the false positive rates of existing tools.

4.1. Experimental Setup

To demonstrate the feasibility of our methodology in a practical setting, we developed a prototype of RACEDB in Python. RACEDB is designed to integrate seamlessly into existing web application testing procedures. Its design principles are applicable to any PHP web application that utilizes MySQL for persistent data storage. Our current implementation specifically targets web applications based on the LAMP stack (Linux, Apache, MySQL, PHP). This choice reflects the widespread popularity of LAMP as a web application deployment method [18], [35], [50]. We discuss future extension plans to broaden support for additional web application frameworks and database technologies in § 6.

Application Selection. To thoroughly evaluate RACEDB, we selected 14 popular web server applications that are tightly connected to databases. Our selection criteria include: 1) popularity, 2) complexity and reliance on databases, and 3) previous evaluation by other studies [39], [59]. We first chose four popular categories of web server applications: Ecommerce platforms, Online forums, Content Management Systems, and Web Games. To assess the real-world popularity and adoption rates of these technologies, we leverage data from BuiltWith.com [7], a website profiler that tracks backend technologies and analytics. For example, web applications such as WordPress [74], phpBB [57], InvoicePlane [34], and Zen Cart [3] have thousands of deployments on the Internet. We also included web applications previously tested by other studies [39], [59], such as Open Cart [51], MyBB [49], OXID eShop [53], Moodle [48], and Drupal [2]. In addition, we include SchoolMate, which has been popularly evaluated by previous studies as evidenced by more than 1,000 search results from Google Scholar [25] since 2020.

We excluded certain applications despite their popularity or previous evaluations. First, some applications are too simple to contain race vulnerabilities or have limited database interactions. Such applications are not suitable for our evaluation, which focuses on race conditions caused by database interactions. Additionally, several applications are outdated and not supported by the current implementation of RACEDB. For instance, MediaWiki-1.19 [59] and Moodle-2.0.10 [59] only run on PHP 5.2 or earlier versions, which have become obsolete since 2011 and are not supported by RACEDB. Furthermore, we encountered a few outdated applications that could not be installed due to unresolved dependency issues.

In this paper, we include 14 web applications, as shown in Table 1, containing 23,403 files and 1263k Logic Lines of Code (LLoC). We installed these web applications in our testbed and initialized the databases with default or recommended settings. When necessary, we created admin and/or user accounts, which were primarily used for our automated authentication phase discussed in § 3.3. Our testbed runs on Ubuntu 22.04 with a 20-core Intel i7 CPU and 32GB RAM.

Vulnerability Collection. For the 14 applications, we collect reported request race vulnerabilities from various sources, including the CVE repository [13], official vulnerability reports for each application, and GitHub issue pages. We used specific search keywords, such as "request race", "race condition", and ".php race", to identify request race vulnerabilities along with the version information of the target applications.

We excluded 11 reported races from our evaluation due to the following reasons. Some reports lacked sufficient details or contained inaccuracies, some races relied on interactions with real payment gateways (e.g., Paypal, BOA) which were outside the scope of our simulated environment. Finally, a few races involved resources other than databases

Table 1: List of PHP Applications.

Id	Application	Source Code		Database		# SQL Query				Description
		# Files	LLOC	# Tables	# Columns	INSERT	UPDATE	SELECT	Total	
s1	SchoolMate-1.5.4 [64]	63	1,587	15	95	17	32	214	263	Content management system
s2	PHP7-Webchess [73]	29	1,505	7	48	14	20	60	94	Web game
s3	OsCommerce-2.4.0 [52]	422	15,809	49	343	529	10	377	916	Ecommerce platform
s4	CE Phoenix Cart-1.0.7 [1]	1,361	23,938	55	369	149	101	436	686	Ecommerce platform
s5	OpenCart-3.0.3.8 [51]	1,932	60,515	136	834	246	111	586	943	Ecommerce platform
s6	MyBB-1.8.15 [49]	312	49,390	75	824	133	379	2,330	2,842	Online forum
s7	OXID eShop-6.0.2 [53]	663	29,021	38	397	43	58	795	896	Ecommerce platform
s8	Moodle-3.11.8 [48]	11,695	741,387	444	4,077	2,138	1,849	12,219	16,206	Ecommerce platform
s9	Drupal-7.6.9 [2]	148	3,315	62	488	230	140	496	866	Content management system
s10	SMF-2.1.2 [65]	316	45,641	73	525	7	270	929	1,206	Online forum
s11	Zen Cart-1.5.7 [3]	1,829	74,960	103	848	394	215	1,311	1,920	Ecommerce platform
s12	phpBB-3.3.8 [57]	1,091	40,612	69	601	64	341	938	1,343	Online forum
s13	WordPress-5.1.2 [74]	901	84,891	12	94	12	32	271	315	Content management system
s14	InvoicePlane-1.5.11 [34]	2,641	91,036	41	292	29	44	243	316	Ecommerce platform
Total		23,403	1263k	1,179	9,835	4,005	3,602	21,205	28,812	

(e.g., file or cache), which our system is not currently designed to analyze.

Additionally, we included request races reported by previous studies. In total, we identified 21 request race vulnerabilities from 8 applications. For the remaining six applications, we could not find any reported request race vulnerabilities as of May 2024.

Note that ReqRacer can detect request races in the cache. However, this requires modifying the application's cache API, necessitating non-trivial manual effort and a understanding of application-specific details. Additionally, a recent study [61] reports that request races caused by the cache represent a smaller portion (7.2%, 18 out of 249 races they studied) of request races compared to database races (71.5%, 178/249). Considering the combined challenges of cache analysis complexity and the substantial human effort required for modifications, we have opted to exclude them from the scope of this work. Our evaluation will therefore focus on request races arising from the database.

Setup Tools from Previous Studies. To compare RACEDB with previous studies, we first obtained the implementations of Raccoon and ReqRacer from their official sites [62] and installed them in our testbed. We followed the instructions provided on their official sites and in their respective papers.

Raccoon collects database query logs for each request and analyzes them to identify pairs of queries with intersecting read and write columns. It then conducts replay-based verification by running the flagged request consecutively and concurrently against the web application to exploit potential vulnerabilities. By inserting a delay before the vulnerable writing query, the oracle counts the occurrences of the writing query in both serialized and concurrent executions. If the count is higher in the concurrent execution, a vulnerability is confirmed.

In contrast, ReqRacer uses the open-source tool Gor [26] to capture and replay HTTP requests. Reqracer leverages this collected information to identify shared-resource accesses, reason about the happen-before relationship between requests, and enable execution replay. It then replays the inferred racing request candidates, enforcing the identified unserializable interleavings, and observes their effects. Only request races that trigger error messages are reported.

We leveraged the collected known vulnerabilities to evaluate RACEDB and compare it with previous studies [39], [59]. Additionally, we evaluated the total 14 applications with RACEDB to identify any new request race vulnerabilities. The results are reported in the following sections.

4.2. Detection Results

Table 2 presents the request race vulnerability detection results from the 14 applications we tested. Overall, RACEDB successfully detected all 21 previously reported vulnerabilities and identified 18 new vulnerabilities from 14 applications. Meanwhile, Raccoon detected 12 known vulnerabilities out of 21 and identified 6 previously unknown vulnerabilities. Reqracer detected 13 known vulnerabilities and 4 new vulnerabilities. Notably, all vulnerabilities detected by Raccoon and Reqracer were also successfully detected by RACEDB.

In Table 2, we provide detailed information for each vulnerability identified. The first column shows the application id where the vulnerability resides, and the second column lists a simplified ID for the vulnerability. The third column indicates the type of vulnerability (i.e., inter-request race or intra-request race) as discussed in § 3.2. The next column displays the number of database tables involved in each race. The fifth column shows the type of data divergences detected during the verification phase, as discussed in § 3.3. The subsequent three columns present the detection results of each tool tested. The ninth column indicates whether the vulnerability is already reported or newly detected. For new vulnerabilities, we also note the status of our CVE submission: "Known" for already reported vulnerabilities,

Table 2: List of Detected Vulnerabilities

	Vul.	Race type	# Tables involved	Detected divergence	Raccoon	ReqRacer	RACEDB	Reported Vul? (Abusable?)	Brief description
s1	v1	Intra	2	Database state			✓	CVE submitted (N)	incorrect grades
	v2	Intra	1	Access pattern/Database state	✓		✓	CVE submitted (N)	incorrect points
	v3	Intra	1	Access pattern/Error message	✓	✓	✓	CVE submitted (Y)	DB insertion error
	v4	Inter	2	Database state			✓	CVE submitted (N)	incorrect parent/student pair
	v5	Intra	1	Access pattern/Error message	✓	✓	✓	CVE submitted (Y)	DB insertion error
	v6	Inter	1	Error message		✓	✓	CVE submitted (Y)	DB insertion error
s2	v7	Inter	2	Database state			✓	CVE submitted (Y)	2 queens or game freezed
	v8	Intra	1	Access pattern/Error message	✓	✓	✓	CVE submitted (Y)	DB insertion error
s3	v9	Intra	1	Access pattern/Database state	✓		✓	CVE assigned (Y)	download more than its limitation
	v10	Inter	1	Database state			✓	CVE assigned (Y)	oversell
s4	v11	Inter	2	Database state			✓	CVE assigned (Y)	coupon overusage
s5	v12	Intra	1	Access pattern/Error message	✓	✓	✓	Known (Y)	incorrect login attempts
	v13	Intra	1	Access pattern/Database state	✓		✓	Known (Y)	coupon overusage
	v14	Intra	1	Access pattern/Database state	✓		✓	Known (Y)	coupon overusage
s6	v15	Intra	1	Access pattern	✓		✓	Known (Y)	post spam
	v16	Intra	1	Access pattern	✓		✓	Known (Y)	post spam
	v17	Intra	1	Access pattern	✓		✓	Known (Y)	post spam
	v18	Intra	1	Access pattern/Error message	✓	✓	✓	Known (Y)	pm spam
	v19	Intra	1	Access pattern/Error message	✓	✓	✓	Known (Y)	pm spam
s7	v20	Intra	1	Access pattern/Database state	✓		✓	Known (Y)	coupon overusage
s8	v21	Inter	1	Error message		✓	✓	Known (Y)	DB insertion error
	v22	Intra	1	Error message		✓	✓	Known (Y)	DB insertion error
	v23	Inter	1	Error message		✓	✓	Known (Y)	DB fetch error
	v24	Inter	1	Error message		✓	✓	Known (Y)	DB insertion error
s9	v25	Inter	1	Error message		✓	✓	Known (Y)	DB fetch error
s10	v26	Inter	1	Error message		✓	✓	Known (Y)	delete before create
s11	v27	Inter	2	Database state			✓	CVE assigned (Y)	coupon overusage
	v28	Inter	2	Database state			✓	CVE assigned (N)	lost credits
	v29	Inter	2	Database state			✓	CVE assigned (Y)	extra credits through gifting coupon
	v30	Intra	1	Database state			✓	CVE assigned (Y)	extra credits through gifting coupon
s12	v31	Inter	1	Error message		✓	✓	Known (Y)	app error
s13	v32	Inter	1	Database state/Error message		✓	✓	Known (Y)	incorrect rating
	v33	Intra	1	Access pattern/Error message	✓	✓	✓	Known (Y)	incorrect subscribing
	v34	Intra	1	Access pattern/Database state	✓		✓	Known (Y)	incorrect votes
	v35	Intra	1	Access pattern/Database state	✓		✓	Known (Y)	incorrect votes
	v36	Inter	1	Database state/Error message		✓	✓	Known (Y)	incorrect rating
s14	v37	Inter	2	Database state			✓	CVE submitted (Y)	incorrect payment
	v38	Intra	1	Access pattern/Database state	✓		✓	CVE submitted (Y)	incorrect payment
	v39	Inter	2	Database state			✓	CVE submitted (Y)	incorrect payment
	Total				**18**	**17**	**39**		

"CVE submitted" for confirmed vulnerabilities with CVEs submitted, and "CVE assigned" for submitted and assigned CVEs (due to anonymity, CVE numbers are not disclosed in this submission). The next column indicates whether the race can be abused and exploited by a malicious actor to gain an advantage, or if it only negatively impacts legitimate users. The last column provides a brief description of the vulnerability.

We observed limitations in both Raccoon and ReqRacer's ability to detect vulnerabilities involving multiple database tables. These limitations appear to stem from their specific design choices. First, both tools primarily rely on analyzing the `WHERE` clauses of SQL statements to identify interleaving database operations. This approach, while effective in cases they focused, overlooks dependencies that exist between tables at the application level (as discussed in § 2). These application-level dependencies can create additional interleaving opportunities that lead to race conditions. Consequently, this limitation can lead to missed vulnerabilities. 9 out of 39 vulnerabilities involve multiple DB tables, thus both Raccoon and ReqRacer failed to detect them.

Additionally, Raccoon's design appears to focus primar-

ily on a specific type of request race vulnerability, Guarded Race Conditions (GRC). This focus could potentially lead to missing other types of vulnerabilities, such as those involving distinct requests (inter-request race). 18 out of 39 cases belonged to these categories, and Raccoon failed to detect them. In addition, Raccoon's detection strategy primarily relies on comparing database access patterns, such as the number of write operations, between serialized and concurrent executions. While this approach can be effective in some cases, it has limitations. If the database access patterns happen to be identical between the two scenarios, Raccoon fails to detect potential race conditions. This limitation becomes evident in our evaluation. For example, in case of vulnerability v21, both the serialized and concurrent executions invoke the exact same number of database write operations. Consequently, Raccoon fails to detect this race condition. § 4.3.2 discusses a similar case, v30, in detail.

ReqRacer's detection relies on error messages from the database and application, which resulted in it failing to detect five cases (v9, v10, v15, v16, v17, v34, v35 and v38) that corrupt data without emitting any errors. Also, we observe three cases (v13, v14, and v20) that evade ReqRacer's detection algorithm. ReqRacer effectively detects interleavings between database accesses when both read and write operations utilize the same `WHERE` clause. However, the three identified cases employ different `WHERE` clauses in the read and write queries (e.g., using "`coupon_code`" for read and "`coupon_id`" for write), and the dependencies are introduced through the application variables. This allows them to bypass ReqRacer's detection mechanism. These results demonstrate that RACEDB outperforms existing tools in detecting request races in real-world applications, thereby enhancing the security of web applications.

As discussed in the previous section, some of the tested applications are not the most recent versions, as we used the same versions that previous studies [10], [39], [59] have tested, for fair comparison (s3∼s11, s14). In addition, we chose slightly older versions of applications s12 and s13 due to the availability of race bug reports. For these older versions of applications, we studied whether the request races detected by RACEDB still exist in the most recent versions. We found that vulnerabilities v1∼v11 and v26∼v30 still exist in the new versions of the applications, but others (v12∼v25 and v31∼v39) have been fixed by the developers.

4.3. Case Study

In this section, we present detailed analyses of two vulnerability cases (v7 and v39) and compare the performance of RACEDB against previous techniques [39], [59].

4.3.1. Webchess Game-breaking (v7). RACEDB identified a request race vulnerability in Webchess [73], an open-source web chess game, that can lead to game corruption. This vulnerability arises during pawn promotion, a chess rule that allows a pawn reaching the final rank to be upgraded to another piece (e.g., queen). In Webchess, when a white pawn reaches the final rank, the game pauses for

Figure 4: Webchess - Black Player has extra queen.

```php
<?
$history = mysqli_query($dbh, "SELECT * FROM history WHERE gameID =
  ↪  (...)");
if ($isMoving){
    $tmpQuery = "INSERT INTO history (...) VALUES (...)";
    doMove();
    saveGame();
}
elseif($history[$numMoves]['curPiece'] == 'pawn' &&
  ↪  $history[$numMoves]['promotedTo'] == null)
{
    if($history[$numMoves]['toRow'] == 7 ||
    ↪  $history[$numMoves]['toRow'] == 0)
    {
        mysqli_query($dbh, "UPDATE history SET promotedTo =
        ↪  '".getPieceName($_POST['promotion'])."' WHERE gameID =
        ↪  ".$_SESSION['gameID']." AND timeOfMove =
        ↪  '".$history[$numMoves]['timeOfMove']."'");
        saveGame();
    }
}

function saveGame(){
    $values[] = collect_pieces_information();
    // clear old data, then insert new data
    mysqli_query($dbh, "DELETE FROM pieces WHERE gameID =
    ↪  ".$_SESSION['gameID']);
    mysqli_query($dbh, "INSERT INTO pieces (gameID, color, piece,
    ↪  row, col) VALUES ($values));
}
```

Listing 1: Vulnerable code in Webchess game (v7).

the white player to choose the promotion piece. While the white player makes this decision, the black player cannot make moves through the WebChess interface. However, the server still accepts requests from the black player during this window. An attacker (playing black) can exploit this by crafting a network request to move one of their pieces which is straightforward due to the absence of encryption in Webchess. This creates a race condition between the black player's `move` request and the white player's `promotion` request. If the race is successfully exploited, one of the following two scenarios can occur. 1) the black player gains an additional piece (e.g., extra queens) as shown in Figure 4. 2) The game becomes permanently frozen and cannot be resumed. This vulnerability allows the black player to gain an unfair advantage by manipulating the game state during white's pawn promotion. White players expecting to promote a pawn typically have a strategic advantage, making this exploit particularly disruptive.

Listing 1 shows a code snippet from WebChess that illustrates the race condition. When the white player selects a promotion piece, Webchess reads the current game infor-

mation from the `history` table (line 2). Then updates the promotion information in the database, reflecting the white player's choice (line 12). Finally, the `saveGame()` function is called (line 17). At this point, the attacker (black player) crafts and sends a `move` request. This request also fetches the current game information (line 2). The black player's move is then updated in the database (line 4), followed by calling `saveGame()` (line 6). Following these initial actions, the white player's promotion request executes queries at lines 20 and 21. These queries are designed to update the board state in the database. Specifically, they might delete the old board information from the `piece` table and insert a new entry representing the all pieces on the board, which includes the newly promoted piece. However, due to the race condition, the black player's `move` request might also execute these same queries (lines 20 and 21) concurrently. This creates the potential for data corruption. Specifically, both requests insert the entire board state into the `piece` table, resulting in two entries with distinct information in the table. After the race, the application tries to resume the game by retrieving information from the `history` and `piece` tables. If the application successfully resumes the game from this corrupted data, the black player might has extra pieces (queen) due to the distinct information in the `piece` table. In another scenario, the corrupted data retrieved from the database might prevent the application from successfully resuming the game, leading to a permanent game freeze.

This vulnerability presents three challenges that hinder detection by existing tools like Raccoon and ReqRacer. First, these tools are primarily designed to identify race conditions in a single database table. However, in this case, the race condition involves two separate tables: `history` and `piece`. This multi-table aspect falls outside the scope of what these tools are designed to handle. Second, the vulnerability exploits an inter-request race condition. It involves two distinct types of requests: the black player's `move` request and the white player's `promotion` request. Raccoon's focus on single-request type races makes it unsuitable for detecting this. Third, ReqRacer relies on detecting error messages from the application or database to identify race conditions. Unfortunately, this vulnerability does not generate any such error messages. This limitation in ReqRacer's approach prevents it from detecting this race. Furthermore, this case requires message crafting by the attacker, making it challenging to collect query traces without analyzing the application code, a capability unique to RACEDB. Although we provided query traces that included the crafted message to Raccoon and ReqRacer for a conservative comparison, they still failed due to the aforementioned reasons.

RACEDB successfully identified this race condition due to the following reasons. First, sys's ARD technique effectively captured the data dependence across the two tables, `history` and `piece`, through its dependency analysis graph. Then, the verification phase of RACEDB played a key role in confirming the race. It successfully detected a divergence in the `piece` table between the serialized and concurrent executions. This divergence provides concrete evidence of a race condition that could lead to data inconsistencies.

```
1 <?php
2 $q2 = $db->Execute("SELECT * FROM COUPON_GC_CUSTOMER WHERE
  ↪  customer_id='".$_SESSION['customer_id']."'");
3 $new_amount = $q2['amount'] - $_POST['amount'];
4 if ($new_amount < 0) {
5     zencart_redirect('error (gift credits not enough)');
6 }
7 $db->Execute("UPDATE COUPON_GC_CUSTOMER SET amount='".$new_amount."'
  ↪  WHERE customer_id='".$_SESSION['customer_id']."'");
8 $db->Execute("INSERT INTO COUPONS (..., coupon_code, coupon_amount,
  ↪  ...) VALUES ...");
9 $insert_id = $db->Insert_ID();
10 $db->Execute("INSERT INTO COUPON_EMAIL_TRACK(coupon_id,
  ↪  customer_id_sent,...) VALUES ... ");
11 ...
```

Listing 2: Request Race in Zen Cart (v30).

4.3.2. Zen Cart Double Gifts (v30). Zen Cart, a popular e-commerce platform used by over 6,900 stores [14], is vulnerable to a request race vulnerability identified by RACEDB. This vulnerability allows an attacker to exploit the system and send credit coupons to multiple accounts while only deducting the credit value once from their own account. The process of sending gift credit in Zen Cart involves creating a new coupon and sending the code to the recipient, and the sender's credit balance is adjusted accordingly. However, an attacker can exploit a race condition in this process to send multiple coupon codes (to accounts they control) while only deducting the credit value once from their own account.

As shown in Listing 2, a potential race condition exists due to the execution of a `SELECT` query (line 2) and a subsequent `UPDATE` query (line 7) that relies on the previous `SELECT` result. The code executes a `SELECT` query (line 2) to retrieve the sender's current credit balance from the `COUPON_GC_CUSTOMER` table. The retrieved value is stored in `$q2`. A new variable, `$new_amount`, is calculated by subtracting the sending amount from the retrieved credit balance. The code then attempts to update the sender's credit balance in the database (line 7), followed by creating a new coupon for the recipient (line 8). Finally, an email containing the coupon information (line 10) is sent to the recipient.

Imagine an attacker with $100 credit attempting to send $50 to two accounts they control. Both requests would concurrently read the same initial credit balance (e.g., $100) from the database due to the `SELECT` query (line 2). Based on the initial balance, both requests would calculate a new balance of $50 (original balance - sending amount). The race condition arises because the update to the sender's credit balance (line 7) might not occur before both requests proceed. This could result in both requests using the outdated balance of $100, leading to an update of $50 instead of $0. Consequently, both requests might successfully create new coupons for the intended recipients, essentially duplicating the credit transfer. This leaves the sender's account with only $50 instead of the expected $0 balance.

While this vulnerability appears straightforward, existing tools like Raccoon and ReqRacer fail to detect it. Raccoon's detection mechanism relies on identifying differences in the number of database write queries between serialized and concurrent executions of the query trace. However, if the

Table 3: Manifested and False Positives (TP represents exploitable true positives; M denotes manifested races; FP indicates false positives).

	Raccoon			ReqRacer			RACEDB		
	TP	M	FP(%)	TP	M	FP(%)	TP	M	FP(%)
s1	3	8	10 (47.6%)	3	0	4 (57.1%)	6	3	4 (30.8%)
s2	1	5	2 (25.0%)	1	1	1 (33.3%)	2	3	1 (16.7%)
s3	1	5	4 (40.0%)	0	3	6 (66.7%)	2	5	4 (36.4%)
s4	0	1	1 (50.0%)	0	2	2 (50.0%)	1	2	2 (40.0%)
s5	3	1	1 (20.0%)	1	0	1 (50.0%)	3	6	1 (10.0%)
s6	5	3	1 (11.1%)	2	0	4 (66.67%)	5	6	2 (15.4%)
s7	1	0	1 (50.0%)	0	1	4 (80.0%)	1	1	1 (33.3%)
s8	0	8	15 (65.2%)	4	2	1 (14.3%)	4	4	1 (11.1%)
s9	0	11	15 (57.7%)	1	0	1 (50.0%)	1	1	1 (33.3%)
s10	0	7	5 (41.7%)	1	0	2 (66.7%)	1	2	0 (0%)
s11	0	10	9 (47.4%)	0	1	4 (80.0%)	4	2	1 (14.3%)
s12	0	2	3 (60.0%)	1	2	6 (66.7%)	1	4	3 (37.5%)
s13	3	2	5 (50.0%)	3	1	4 (50.0%)	5	3	2 (20.0%)
s14	1	1	1 (33.3%)	0	0	4 (100%)	3	1	1 (20.0%)
Total	18	64	73 (47.1%)	17	13	44 (59.5%)	39	43	24 (22.6%)

attacker's credit balance exceeds the total amount they are sending, both concurrent and serialized executions would result in the same number of writes, causing Raccoon to miss the issue. ReqRacer, on the other hand, depends on error messages emitted by the application or database to detect races. In this scenario, no errors occur regardless of the race, causing ReqRacer to fail as well.

4.4. Analysis of False Positives

We discuss false positive cases reported by RACEDB and compare them with Raccoon and ReqRacer. As shown in Table 3, RACEDB identified 106 potential request race bugs. Through manual analysis, we confirmed 39 of these to be actual race conditions that could lead to permanent data corruption, application errors, or database errors. Additionally, 43 of them caused deviations in execution states or data corruption, however, we failed to confirm any negative consequences resulting from these deviations. We mark them as *manifested* races because, although we could not exploit them, they signal unintended behavior and *could become exploitable in the future*. The remaining 24 out of 106 reports were classified as false positives as we could not observe clear deviations in execution or data corruption.

We further investigated the root causes of these manifested and false positive races and identified two main factors.

First, as discussed in § 3.2, RACEDB employs automated program analysis to identify data dependencies. These dependencies are crucial in identifying races that cause data corruption. However, without application-specific knowledge, it is difficult to fully understand the consequences of this data corruption. We have observed cases where data corruption does not result in any negative consequences for users or the application, and we classify these cases as

```
1 <?
2 $counter_query = "select startdate, counter from COUNTER";
3 $counter = $db->Execute($counter_query);
4 if ($counter->RecordCount() > 0) {
5   ...
6   $counter_now = ($counter->fields['counter'] + 1);
7   $sql = "update COUNTER set counter = '".$counter_now."'";
8   $db->Execute($sql);
9 }
```

Listing 3: Manifested Race Example - Zen Cart

false positives. For example, in Listing 3, RACEDB analyzes application-level data dependency in Zen Cart application, appeared at lines 3, 6, and 7. The return value of the `SELECT` query at line 3 is used to modify a variable (`$counter_now` at line 6) and then the variable is used in the UPDATE query at line 7. This creates a data dependency where the update relies on the initial retrieved value. During the replay phase of analysis, RACEDB monitors a specific database field, `counter` in `COUNTER` table, for any discrepancies between serialized and concurrent executions. For instance, imagine the initial value of the `counter` is '1', and two requests execute the code concurrently. Both read '1' from the table (line 1), and store the incremented value of '2' in `$counter_now`. Then, update the `counter` in the database at line 7. In this scenario, the final counter value would be '2'. However, in a serialized execution, the final value would be '3' since each request updates the `counter` independently.

This example demonstrates a race condition that corrupts the counter value. However, RACEDB categorizes it as a manifested rather than a race vulnerability for the following two reasons. First, despite the data corruption, we fail to identify observable consequences for users or the application. Also, we observe that the counter value is routinely reset, suggesting that the inconsistency is temporary.

Second, RACEDB utilizes a static analysis technique to identify error handling-logic within an application. It then checks for specific error messages via text matching. This approach can lead to false positives as error messages can differ across different applications. For instance, consider a false positive scenario in the WebChess application. The relevant code snippet is listed in Listing 4. The WebChess allows the user to send a refresh request (executing `loadGame` at line 7) to update the board state. Suppose a white player moves a piece, triggering `saveGame` at line 2 to clear old data and insert the new data. Concurrently, a refresh request arrives from the black player, causing `SELECT` at line 8 to execute right after the `DELETE` at line 3 but before the `INSERT` at line 4. This might lead to an error message at line 14, which would not occur in serialized execution. According to the definition, RACEDB detects this scenario as a race due to the error message which only occurs in the concurrent execution. However, this essentially is a warning message, it cannot be abused by a malicious user.

Table 3 also presents manifested and false positive races reported by Raccoon and ReqRacer. Raccoon identified a total of 155 races, of which only 18 were confirmed to be actual races (resulting in 64 manifested and 73 false positives). Similarly, ReqRacer reported 74 cases, with 17

```php
<?
function saveGame(){
    mysqli_query($dbh, "DELETE FROM pieces WHERE gameID =
    ↪  ".$_SESSION['gameID']);
    mysqli_query($dbh, "INSERT INTO pieces '(...) VALUES (...);")
}

function loadGame(){
    $pieces = mysqli_query("SELECT * FROM pieces WHERE gameID =
    ↪  $_SESSION['gameID']");
    isInCheck();
}
function isInCheck(){
    if($findking){return true;}
    else{
        echo("CRITICAL ERROR: KING MISSING!");
        return false;
    }
}
```

Listing 4: False Positive Example - Webchess

confirmed races, 43 manifested, and 24 false positives.

4.5. Performance Evaluation

The setup process for evaluating each application typically required between 2 to 4 hours of effort by a single person. This process involved the following steps:

1. **Installation:** Installing the target application following the vendor's instructions.
2. **Account Creation:** For applications requiring user accounts, setting up at least two regular user accounts and one administrator account.
3. **Operation Simulation:** Simulating standard operations within the web application using each created account. For example, for e-commerce applications, this included actions such as adding items to the cart, completing orders, and redeeming coupons. For forums, this involved posting topics and commenting on discussions.

These steps were essential for SynthDB's concolic execution [10], as we used its implementation to generate query traces and synthesize database states. Additionally, note that both Raccoon [39] and ReqRacer [59] require manual collection of query traces as part of their setup procedures.

For the performance evaluation, we excluded the manual setup steps described above. Figure 5 presents the results. On average, RACEDB takes 77.3 minutes to test each application, compared to 34.9 minutes for Raccoon and 19.8 minutes for ReqRacer. As expected, RACEDB requires more time because it identifies a greater number of potential data races and verifies them through replay-based techniques.

5. Related Work

Concurrency Bugs in Web Applications. Throughout this paper, we comprehensively discuss the most closely related works, Raccoon [39] and ReqRacer [59], and their limitations. In addition to these two, there are a few other works focusing on race detection in web applications. The approaches and algorithms proposed in earlier works [54], [71] have been adopted in Raccoon. Zheng et al. [76] proposed a static approach to detect race problems in server-side scripts. Furthermore, recent studies [60], [61] have conducted thorough investigations into concurrency problems, including races [61] and deadlocks [60], and their effects on web applications.

Traditional Race Detection. Race conditions have been widely studied in multi-threaded applications [6], [19], [44] and distributed systems [9], [11], [30]. Thread-race detection techniques typically focus on identifying data races in shared memory, while process-race detection techniques target race conditions across distributed nodes in cloud environments.

However, advancements in thread-level and process-level race detection are not directly applicable to database-backed web applications. The key challenge lies in the fundamental difference between concurrency models used in web applications (often centered around database interactions) and those employed in multi-threaded or distributed systems.

Web Application Testing Techniques. Static code scanning is a widely used technique for identifying security vulnerabilities in web applications [5], [16], [29], [32], [41], [47], [70], [72], [75]. This approach analyzes the application code without executing it, thus not requiring dynamic resources such as databases. However, static analysis tools often struggle with web applications written in dynamic languages like PHP due to the inherent challenges of interpreting code behavior without actual execution. Dynamic testing involves executing the web application and analyzing its behavior for vulnerabilities [8], [17], [27], [28], [31], [37], [45], [55], [56], [56], [63], [67], [77]. This method can effectively analyze dynamic execution environments and user interactions. However, dynamic testing has difficulty achieving high code coverage due to the lack of dynamic resources like databases.

To address the limitations of dynamic analysis, SynthDB [10] proposes a technique for generating synthetic databases. SynthDB leverages concolic execution to identify interactions between web applications and databases, generating synthetic database states. These states can then be used to execute the application code and potentially reveal vulnerabilities that rely on specific database interactions, including request races. Our work uses SynthDB to generate synthesized database states for testing web applications. This approach allows us to access the web application code related to request races and generate query traces caused by these requests. In addition, R3 [42] proposed a record-and-replay technique for database-backed web applications, faithfully replaying concurrent bugs.

6. Discussion and Future Work

Request Race in Other Resource Types. The current design of RACEDB focuses on verifying request races by detecting divergences between serialized and concurrent executions. This verification process considers database state, application error messages, and database access patterns. However, if the impact of a race condition does not directly affect the data we monitor, RACEDB might miss it. As discussed in § 4.1, an example of this limitation is the

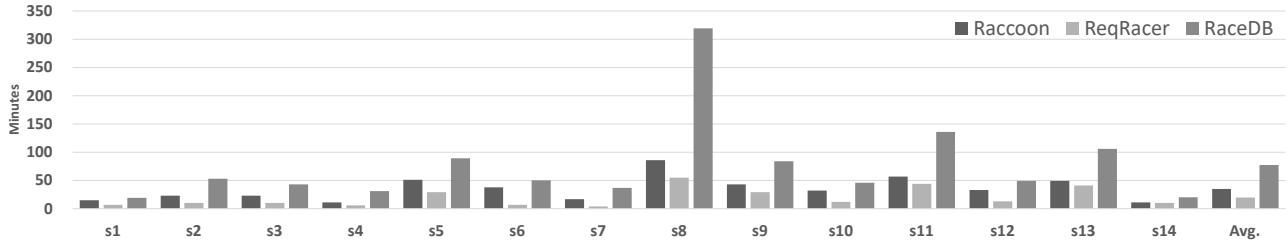

Figure 5: Performance Evaluation Results.

inability to detect cache-related races. Existing tools like ReqRacer [59] can address cache-based races, but they often require non-trivial manual effort to modify the application's cache API. To overcome this limitation, we plan to explore automated techniques for identifying cache data to be monitored. This involves leveraging program analysis techniques to automatically pinpoint cache data that needs to be monitored during verification. This investigation into automated cache data identification is one of our long-term development roadmap for RACEDB.

False Positive Issues. As discussed in § 4.4, RACEDB currently generates some manifested races and false positives even after the automated verification step. These cases primarily come from non-harmful request races. In manifested cases, although a race condition is detected and a divergence is observed in the database or application states, we could not confirm any exploitations that negatively impact user functionality. False positives are cases where we could not observe any clear deviations in execution or data corruption. Distinguishing between truly harmful and non-harmful races remains a significant challenge. To address this, we plan to leverage concolic execution as a mitigation strategy. This technique involves systematically exploring different execution paths from the identified race condition. During this exploration, we will track the divergent data and observe whether its inconsistency disappears automatically or persists. Additionally, we will monitor the downstream impacts of the corrupted data to infer potential damage to the application or user experience.

Undirected Graphs in ARD. In our approach, ARD (Application-Aware Request-race Detection) graphs are undirected. These graphs represent potential conflicts between database operations that interact with overlapping rows, without specifying the exact order of these interactions. This conservative modeling helps RACEDB avoid false negatives, as any of these interactions could potentially lead to a request race. Potential false positives are further filtered out during dynamic verification. This design choice is inherited from the 2AD framework, as discussed earlier in the paper. Extending ARD to directed graphs could improve accuracy by capturing the precise order of operations. We leave this enhancement as a future work.

Support Other Languages and DBMS. RACEDB leverages SynthDB [10] for concolic execution and database synthesis. Consequently, RACEDB inherits SynthDB's limitations in terms of language and database support. Currently, RACEDB is limited to PHP applications and MySQL databases. Throughout this project, we gained a clear understanding of SynthDB's implementation details, which instills confidence in our ability to extend its capabilities. We plan to address these limitations by extending the capabilities of SynthDB. Specifically, we plan to develop an instruction-level trace and parser specifically for JavaScript applications. We also plan to enhance the current query analyzer to support PostgreSQL databases in addition to MySQL. By expanding SynthDB's functionalities, we aim to significantly broaden the applicability of RACEDB to a wider range of web application and database.

7. Conclusion

We propose RACEDB, a novel system that automatically detects and verifies request races in database-backed web applications. RACEDB analyzes diverse data dependencies within both the application and database, enabling the identification of intricate race conditions. Furthermore, automated verification with replay-based execution significantly reduces false positives. Evaluation on 14 real-world web applications demonstrates that RACEDB outperforms state-of-the-art techniques in terms of detection rate, encompassing both known and new vulnerabilities, with a lower false positive rate than existing tools. By automating race condition detection and verification, RACEDB is expected to enhance the security of web applications.

Acknowledgment

The authors would like to express their appreciation to the anonymous reviewers for their valuable and constructive feedback, as well as to the shepherd for their guidance in improving the paper during the revision process. The authors gratefully acknowledge the support of NSF 1916500, 2426653, and 2427783. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] "CE Phoenix Cart," https://phoenixcart.org/.

[2] "Drupal," https://www.drupal.org/.

[3] "Zen Cart," https://www.zen-cart.com/.

[4] A. Adya, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," 1999.

[5] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," *IEEE EuroS&P'17*, pp. 334–349.

[6] M. D. Bond, K. E. Coons, and K. S. McKinley, "Pacer: Proportional detection of data races," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2010, pp. 255–268.

[7] "BuiltWith," 2024, https://builtwith.com/.

[8] A. Bulekov, R. Jahanshahi, and M. Egele, "Saphire: Sandboxing PHP applications with tailored system call allowlists," in *30th USENIX Security Symposium (USENIX Security 21)*.

[9] Y. Cao, X. Zhang, H. Chen, and B. Zang, "Racer: Effective data race detection for the cloud," in *Proceedings of the 2020 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2020, pp. 1–14.

[10] A. Chen, J. Lee, B. Chaulagain, Y. Kwon, and K. H. Lee, "Synthdb: Synthesizing database via program analysis for security testing of web applications." in *NDSS*, 2023.

[11] G. Chen, S. Lu, S. Krishnan, S. Xanthos, and S. Thummalapenta, "Pacer: Proportional detection of data races," in *Proceedings of the 2019 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2019, pp. 255–268.

[12] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2325–2342. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[13] "CVE," https://cve.mitre.org/.

[14] "Store Leads," https://storeleads.app/reports/zencart.

[15] "CVE-2022-4037," 2023, https://nvd.nist.gov/vuln/detail/CVE-2022-4037.

[16] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *USENIX Security Symposium*, 2014.

[17] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *21st USENIX Security Symposium*, Aug. 2012, pp. 523–538.

[18] M. Elahi, F. Jahan, M. R. Shahriar, and M. Ahsan, "Performance evaluation of web servers for lamp stack web applications," *International Journal of Computer Applications*, vol. 166, no. 11, pp. 20–24, 2017.

[19] C. Flanagan and S. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009, pp. 121–133.

[20] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 121–133. [Online]. Available: https://doi.org/10.1145/1542476.1542490

[21] "Withdrawal vulnerabilities enabled bitcoin theft from flexcoin and poloniex." 2014, https://www.pcworld.com/article/444202/withdrawal-vulnerabilities-enabled-bitcoin-theft-from-flexcoin-and-poloniex.html.

[22] M. Gligoric and R. Majumdar, "Model checking database applications," in *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19*. Springer, 2013, pp. 549–564.

[23] T. V. Goethem, C. Pöpper, W. Joosen, and M. Vanhoef, "Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1985–2002. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/van-goethem

[24] S. Gong, D. Peng, D. Altınbüken, P. Fonseca, and P. Maniatis, "Snowcat: Efficient kernel concurrency testing using a learned coverage predictor," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 35–51. [Online]. Available: https://doi.org/10.1145/3600006.3613148

[25] "Google Scholar," https://scholar.google.com/.

[26] "Gor," 2024, https://github.com/adjust/gor.

[27] W. G. J. Halfond, A. Orso, and P. Manolios, "Wasp: Protecting web applications using positive tainting and syntax-aware evaluation," *IEEE Transactions on Software Engineering*, vol. 34, pp. 65–81, 2008.

[28] B. Hawkins and B. Demsky, "Zenids: Introspective intrusion detection for php applications," *IEEE/ACM 39th International Conference on Software Engineering*, pp. 232–243, 2017.

[29] M. Hills, P. Klint, and J. J. Vinju, "An empirical study of php feature usage: a static analysis perspective," *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013.

[30] X. Huang, J. Chen, W.-C. Chuang, and Y. Shoshitaishvili, "Order-aware race detection in distributed systems," in *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE/ACM, 2021, pp. 250–262.

[31] Y.-W. Huang, C.-H. Tsai, T.-P. Lin, S.-K. Huang, D. T. Lee, and S.-Y. Kuo, "A testing framework for web application security assessment," *Comput. Networks*, vol. 48, pp. 739–761, 2005.

[32] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *WWW '04*, 2004.

[33] "Jack cable. [n.d.]. race condition in redeeming coupons." 2016, .https://hackerone.com/reports/157996.

[34] "InvoicePlane," https://www.invoiceplane.com/.

[35] P. Jayaweera and S. Perera, "Implementation of lamp stack for cloud computing," in *2014 International Conference on Advances in ICT for Emerging Regions (ICTer)*. IEEE, 2014, pp. 181–188.

[36] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 6–pp.

[37] S. Kals, E. Kirda, C. Krügel, and N. Jovanovic, "Secubat: a web vulnerability scanner," in *WWW '06*, 2006.

[38] J. Kettle, "Smashing the state machine: The true potential of web race conditions," in *BlackHat USA 2023*, https://www.blackhat.com/us-23/briefings/schedule/index.htmlsmashing-the-state-machine-the-true-potential-of-web-race-conditions-31712.

[39] S. Koch, T. Sauer, M. Johns, and G. Pellegrino, "Raccoon: automated verification of guarded race conditions in web applications," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1678–1687.

[40] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 162–180. [Online]. Available: https://doi.org/10.1145/3341301.3359638

[41] P. Li and W. Meng, "Lchecker: Detecting loose comparison bugs in php," *Proceedings of the Web Conference 2021*, 2021.

[42] Q. Li, P. Kraft, M. Cafarella, c. Demiralp, G. Graefe, C. Kozyrakis, M. Stonebraker, L. Suresh, X. Yu, and M. Zaharia, "R3: Record-replay-retroaction for database-backed applications," *Proc. VLDB Endow.*, vol. 16, no. 11, p. 3085–3097, jul 2023. [Online]. Available: https://doi.org/10.14778/3611479.3611510

[43] C. Liu, D. Zou, P. Luo, B. B. Zhu, and H. Jin, "A heuristic framework to detect concurrency vulnerabilities," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 529–541. [Online]. Available: https://doi.org/10.1145/3274694.3274718

[44] U. Mathur and M. Viswanathan, "Optimal prediction of synchronization-preserving races," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2020, pp. 257–271.

[45] S. McAllister, E. Kirda, and C. Krügel, "Leveraging user interactions for in-depth testing of web applications," in *RAID*, 2008.

[46] I. Medeiros, N. Neves, and M. Correia, "Dekant: a static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 1–11.

[47] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan, "Mace: Detecting privilege escalation vulnerabilities in web applications," *Proceedings of the ACM CCS'14*.

[48] "Moodle," https://moodle.org/.

[49] "MyBB," https://mybb.com/.

[50] E. Naramore, J. Gerner, Y. L. Scouarnec, J. Stolz, and M. Glass, *Beginning PHP5, Apache, MySQL Web Development*. John Wiley & Sons, 2005.

[51] "OpenCart," https://www.opencart.com/.

[52] "OsCommerce," https://www.oscommerce.com/.

[53] "OXID eShop," https://www.oxid-esales.com/en/.

[54] R. Paleari, D. Marrone, D. Bruschi, and M. Monga, "On race vulnerabilities in web applications," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 5th International Conference, DIMVA 2008, Paris, France, July 10-11, 2008. Proceedings 5*. Springer, 2008, pp. 126–142.

[55] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications," in *NDSS*, 2014.

[56] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting csrf with dynamic analysis and property graphs," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, p. 1757–1771.

[57] "phpBB," https://www.phpbb.com/.

[58] "proxySQL," https://proxysql.com/.

[59] Z. Qiu, S. Shao, Q. Zhao, and G.Jin, "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*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 842–854. [Online]. Available: https://doi.org/10.1145/3468264.3468594

[60] Z. Qiu, S. Shao, Q. Zhao, and G. Jin, "A characteristic study of deadlocks in database-backed web applications," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 510–521.

[61] Z. Qiu, S. Shao, Q. Zhao, H. A. Khan, X. Hui, and G. Jin, "A deep study of the effects and fixes of server-side request races in web applications," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 744–756.

[62] "Reqracer artifact," 2024, https://github.com/caseqiu213/reqracer_fse_artifact.

[63] P. Saxena, D. A. Molnar, and B. Livshits, "Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications," in *CCS '11*, 2011.

[64] "SchoolMate," https://sourceforge.net/projects/schoolmate/files/SchoolMate/.

[65] "Simple Machines Forum," 2022, https://www.simplemachines.org/.

[66] J. Smith, L. N. Q. Do, and E. Murphy-Hill, "Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security," in *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, 2020, pp. 221–238.

[67] S. Son, K. S. McKinley, and V. Shmatikov, "Diglossia: detecting code injection attacks with precision and efficiency," *Proceedings of the ACM conference on Computer & communications security*, 2013.

[68] "Egor homakov. [n.d.]. hacking starbucks for unlimited coffee." 2015, https://sakurity.com/blog/2015/05/21/starbucks.html.

[69] B. A. Stoica, S. Lu, M. Musuvathi, and S. Nath, "Waffle: Exposing memory ordering bugs efficiently with active delay injection," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 111–126. [Online]. Available: https://doi.org/10.1145/3552326.3567507

[70] F. Sun, L. Xu, and Z. Su, "Detecting logic vulnerabilities in e-commerce applications," in *NDSS*, 2014.

[71] T. Warszawski and P. Bailis, "Acidrain: Concurrency-related attacks on database-backed web applications," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 5–20.

[72] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *PLDI '07*, 2007.

[73] "Webchess," https://github.com/halojoy/PHP7-Webchess.

[74] "WordPress," https://wordpress.com/.

[75] Y.Zheng and X.Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," *35th International Conference on Software Engineering*, pp. 652–661, 2013.

[76] Y. Zheng and X. Zhang, "Static detection of resource contention problems in server-side scripts," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 584–594.

[77] Y. Zhou and D. Evans, "Ssoscan: Automated testing of web applications for single sign-on vulnerabilities," in *USENIX Security'14*.

Appendix A.
Meta-Review

A.1. Summary of Paper

The paper presents RaceDB, a tool for finding database races in web applications. RaceDB builds on and extends SynthDB and the 2AD algorithm to detect races between parts of the database related via the program code. The detected potential races are then subjected to an automatic verification technique based on ProxySQL to replay the race to single out the actual races that cause differences in the database state, application state, or database access patterns. Multiple CVEs in popular PHP web applications demonstrate the tool's practical impact.

A.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability

A.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. It contributes to improving the area of race condition vulnerability detection by introducing an approach that combines concolic execution, code and database data dependency analysis, and dynamic testing by replaying execution traces and recording discrepancies.
2) The paper creates a new tool to enable future science. The dual-context analysis that underlines the tool provides a more thorough detection mechanism compared to traditional methods.
3) The paper identifies multiple vulnerabilities in popular open-source PHP web applications. Some of these confirm known vulnerabilities while others report newly discovered vulnerabilities leading to freshly assigned CVEs.
4) Authors plan to release the tool for reproducibility and future science.

A.4. Noteworthy Concerns

1) One reviewer has raised concerns about the limited scale of evaluation, as the paper includes only a small selection of applications assessing false negatives.