Enhancing Web Application Security Through Program Analysis-Based Database Synthesis and Race Vulnerability Detection

by

An Chen

(Under the Direction of Kyu Hyung Lee)

Abstract

Conventional security analysis methods for web applications typically concentrate on either the application codebase or the backend database, often overlooking the critical interactions between them. To address this limitation, this dissertation presents an innovative program analysis-based approach that extracts dependencies between the codebase, queries, and schema. This method not only enhances dynamic security testing by synthesizing comprehensive databases but also automatically identifies and exploits complex request race conditions within the web application's database, thereby improving the overall security of the web application.

In this dissertation, we propose two closely related techniques to enhance the security of database-backed web applications. First, we introduce SynthDB, a program analysis-based database generation technique for web applications. SynthDB leverages a concolic execution engine to identify interactions between the codebase and queries. It collects and solves various database constraints to reconstruct a database that enables the exploration of uncovered program paths. Our evaluation shows that SynthDB outperforms state-of-the-art database generation techniques in code and query coverage across 17 real-world PHP applications, achieving 14.0% higher code and 24.2% higher query coverage.

Furthermore, we introduce RaceDB, a technique designed to address request race vulnerabilities based on SynthDB's concolic engine. RaceDB features Application-aware Request Race Detection (ARD), which provides comprehensive data dependency analysis of both database queries and application code. This allows RaceDB to identify subtle race conditions missed by other existing approaches. RaceDB also employs automated verification using replay-based execution, efficiently isolating true races from false positives and generating definitive exploits for verified vulnerabilities. RaceDB demonstrated superior detection rates, identifying 21 known request race cases and discovering 18 new ones in 14 real-world PHP web applications, exceeding the performance of existing techniques, which only identified 13 known cases and 6 new ones at best.

In summary, this dissertation proposes a system that employs a program analysis-based approach to extract dependencies between the codebase, queries, and schema. This system introduces multiple innovative techniques that collectively enhance the security testing landscape for database-backed web applications.

Index words: Web Application, Database-Backend Application, Program Analysis, Database Generation, Query Analysis, Security Testing, Request Race

Enhancing Web Application Security Through Program Analysis-Based Database Synthesis and Race Vulnerability Detection

by

An Chen

B.S., Macau University of Science and Technology, China, 2011

M.S., New Jersey Institute of Technology, 2015

A Dissertation Submitted to the Graduate Faculty of the

University of Georgia in Partial Fulfillment of the Requirements for the Degree.

Doctor of Philosophy

Athens, Georgia

2024

©2024

An Chen

All Rights Reserved

Enhancing Web Application Security Through Program Analysis-Based Database Synthesis and Race Vulnerability Detection

by

An Chen

Major Professor: Kyu Hyung Lee

Committee: Le Guan
Wenwen Wang
Yonghwi Kwon

Electronic Version Approved:

Ron Walcott
Dean of the Graduate School
The University of Georgia
August 2024

Acknowledgments

Firstly, I would like to extend my heartfelt gratitude to my advisor, Dr. Kyu Hyung Lee, for his personal and professional guidance throughout my Ph.D. studies. He has been instrumental in my growth as both a researcher and a person. I could not have imagined having a better advisor and mentor for my Ph.D. journey. I am also deeply grateful to Dr. Yonghwi Kwon for his patience and time. His invaluable assistance enabled me to complete two top-quality security papers. Next, I would like to thank my committee members, Dr. Le Guan and Dr. Wenwen Wang, for their time, insightful comments, and encouragement, which were crucial in helping me achieve my research goals during my Ph.D. years. I would also like to thank the University of Georgia and the School of Computing for providing laboratories, equipment, and other essential resources.

In addition, I want to express my appreciation to my wonderful friends at UGA for their support and assistance during my studies. Special thanks to Ruichao for always maintaining a positive attitude and keeping my spirits high.

Finally, I am profoundly grateful to my family and Xinran. I cannot express how thankful and fortunate I am to have your unwavering support and love. Thank you, always.

Contents

Acknowledgments **iv**

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

1.1 Research Challenges 3

1.2 Thesis Statement 5

1.3 Contributions 5

1.4 Outlines 6

2 SynthDB: Synthesizing Database via Program Analysis for Security Testing of Web Applications **7**

2.1 Motivating Example 9

2.2 Design and Implementation 13

2.3 Evaluation 28

3 RaceDB: Detecting Request Race Vulnerabilities in Database-Backed Web Applications **40**

3.1 Motivating Example 42

3.2 System Design 45

3.3 Evaluation 52

4 Limitations **67**

4.1 Synthesizing Database via Program Analysis for Web Applications 67

4.2 Detecting Request Race Vulnerabilities in Web Applications 69

5 Related Work **71**

5.1 Test Data Generation 71

5.2 Web Application Testing Techniques 72

5.3 Concurrency Bugs in Web Applications 73

5.4 Traditional Race Detection 73

6 Conclusion **74**

Bibliography **76**

List of Figures

2.1 Simplified Code Snippet from SchoolMate [90]. 9
2.2 Generated Synthetic Databases by Existing Techniques and SynthDB. 12
2.3 Overview of SynthDB (*Z3 solver [115]). 14
2.4 Extracting Path Constraints 15
2.5 Extracting Schema Constraints from Database Schema. 19
2.6 Extracting Query-condition Constraints. 21
2.7 Extracting Pre-query Constraints. 22
2.8 Extracting Post-query Constraints. 23
2.9 Extracting Synchronized-query Constraints. 24
2.10 Overall Procedure of Synthesizing Database (Highlighted columns in (b)~(e) present the source lines where we extracted constraints from). 25
2.11 Program code requiring two constraints that are conflicting. 28
2.12 Code and Query Coverage of SynthDB ('Default DB' presents an execution with a DB right after the installation. s1~s17 are the ids from Table 2.1). 33
2.13 Code Coverage Results by Existing Fuzzing Tools. 38

3.1 Motivation Example 42
3.2 System Overview of RaceDB. 45
3.3 ARD graph generated the motivation example. 49
3.4 Webchess - Black Player has extra queen. 58

List of Tables

2.1 List of PHP Applications. 29

2.2 Top 75 frequent SQL keywords. 30

2.3 Comparison with Baseline Approaches. 31

2.4 Burp Suite results with databases 36

2.5 Reachability Test against PHP Vulnerabilities. 37

2.6 Time Taken to Generate a Test Database (in minute) and the number of records generated. 39

3.1 List of PHP Applications. 53

3.2 List of Detected Vulnerabilities 65

3.3 False Positive Comparison 66

Chapter 1

Introduction

Web servers deliver web pages to clients, providing essential web services to businesses and customers. When web servers receive requests from remote users, they execute server-side programs, known as web applications, to process these requests and generate the requested data for display on the client-side, typically in web browsers.

As these web applications serve many clients daily, they become prime targets for cybercriminals [103, 98, 15, 108]. Firstly, taint-type vulnerabilities (e.g., SQL injection or XSS vulnerabilities) are frequently exploited by attackers, leading to severe consequences. For instance, in March 2023, a critical vulnerability was discovered in the WordPress plugin "Advanced Custom Fields," which allowed attackers to inject malicious scripts through improperly sanitized user inputs, resulting in unauthorized data access and manipulation that affected millions of websites [1]. Such vulnerabilities pose significant security concerns because, once exploited, a cyber attacker may compromise all future client users of the server, potentially causing catastrophic consequences through injected malicious scripts.

Secondly, beyond taint-type vulnerabilities, concurrent remote requests can lead to unpredictable and inconsistent behavior if not managed properly. Race conditions, deadlocks, and database corruption are potential outcomes of inadequate concurrency control. Several high-profile incidents underscore the importance of addressing race conditions. For example, vulnerabilities in Instacart, Starbucks, Flexcoin, and GitLab resulted in issues such as double coupon redemption [45], duplicate balance transfers [95],

wallet overdraws leading to Flexcoin's bankruptcy [33], and account hijacking [22]. The GitLab account hijacking incident in 2023 exploited a request race condition to forge verified emails and potentially take over third-party accounts when GitLab was used as an OAuth provider [22]. The attack exploited the lack of proper synchronization during the email verification process, allowing malicious actors to intercept and manipulate the verification flow, leading to unauthorized access to user accounts.

These incidents highlight the critical need to address vulnerabilities in modern web applications to prevent similar high-impact failures. Therefore, testing web server applications to identify and fix vulnerabilities before cyber attackers can exploit them is crucial to building secure web services.

Research has focused on either statically or dynamically analyzing web applications to find and mitigate security vulnerabilities. Static analysis techniques [91, 49, 23] struggle with the dynamic features of PHP, the predominant programming language for web applications. Conversely, dynamic security testing [38, 113, 4, 7, 11, 46] shows more promise when targeting PHP. However, dynamically testing database-backed web applications is challenging due to their reliance on external databases. The increasing prevalence of request races further complicates the interaction between application logic and database operations. Without a comprehensive database, reaching potentially vulnerable code is difficult, limiting the effectiveness of dynamic security testing methods. Despite the importance of databases in security analysis, obtaining a realistic database for testing is challenging. Conversations with industry collaborators revealed that sharing a real-world website's database is extremely difficult due to privacy concerns raised by the sensitive content within the database.

Moreover, request race vulnerabilities are challenging to detect, trigger, and verify. Existing dynamic request race detection techniques for web applications primarily focus on collected database query traces [106, 54, 84], often overlooking the interdependencies between application logic and database operations. This oversight leads to a high rate of false negatives. Additionally, these techniques validate request race vulnerabilities using predefined, static database content, which does not account for the varied behaviors that can arise from different database states. As a result, they lack robustness in reliably triggering and verifying potential request race vulnerabilities.

Traditional security analysis techniques for web applications typically concentrate on either the application codebase or the backend database, often neglecting the crucial interactions between them. This dissertation addresses this gap by developing methods to uncover the interdependencies between the application codebase, database queries, and the database schema. By thoroughly analyzing the interactions between application logic and database operations, we provide effective methods for database generation and request race vulnerability detection, these techniques significantly improve security testing coverage and broaden the scope of vulnerability detection.

1.1 Research Challenges

In this section, we discuss the unique challenges involved in identifying the interdependencies between the application codebase and database queries, effective methods for testing database generation, and detecting and verifying request race vulnerabilities.

1.1.1 Database synthesizing for web application security testing

Generating a database based on program analysis involves significant challenges due to the intricate dependencies between the codebase, the database queries, and the database schema. Multiple database synthesizing approaches have been proposed to support various program analysis and testing techniques. These approaches typically generate a synthesized database by analyzing the database schema and query traces. While most methods focus on the database schema, which includes the definition of database tables and entries along with relational constraints such as foreign keys [29, 52, 97, 118], they often fail to capture implicit relationships established by the program code. These relationships may include dependencies between database entries that are processed together or depend on each other.

For instance, the recent EvoSQL [17] approach leverages SQL query traces and the database schema to capture relational constraints exhibited in the queries. However, it only focuses on database queries, without accounting for the program code that processes these query results. This limitation means that implicit dependencies and the context in which queries are executed are often overlooked. For example, if

the return value of a `SELECT` has subsequently been checked by the predicate of an `IF` statement, which is common in modern web applications, such interdependencies between database content and application control flow cannot be identified by analyzing the concrete query traces alone. Since the return value can be arbitrary, existing techniques lack the ability to isolate it from other concrete values.

Moreover, the effectiveness of these analyses heavily depends on the quality of the query traces provided by the user. Obtaining comprehensive query traces is typically dependent on the quality of the database since many programs execute queries based on previous query results. For example, a program may retrieve detailed data only after narrowing down a specific data entry through initial queries. Thus, ensuring a robust and representative database for testing involves navigating these complex interdependencies, which presents a substantial challenge in program analysis.

1.1.2 Database analyzing for detecting request race in web applications

Detecting request race vulnerabilities in web server applications presents significant challenges due to the intricate dependencies between application code and database query traces. In these applications, interleavings occur between the execution of request handlers. If resources such as variables or database contents accessed during request handling are not properly protected from concurrent access, their values can be corrupted. Unlike local program variables, database content can be accessed and modified by any execution, resulting in a vastly larger number of potential content states to consider.

This complexity is further exacerbated by sophisticated database queries and the wide range of user requests. Existing automated concurrency bug-finding techniques, which often focus on exploring different interleavings of thread executions, struggle with this complexity. They typically test various permutations of thread schedules via customized schedulers, but these methods are less effective for web applications where database content and query traces are central to identifying race conditions. Consequently, there is a pressing need for more advanced techniques that can accurately account for the intricate interactions between application code and database states to detect and mitigate request race vulnerabilities.

1.2 Thesis Statement

Traditional security analysis techniques for web applications mainly focus on either the application codebase or the backend database, often neglecting the interactions between them. This dissertation introduces a novel program analysis-based method that synthesizes databases by extracting dependencies between the codebase, queries, and schema, thereby enhancing dynamic security testing and automatically identifying and exploiting complex request race vulnerabilities in web applications.

1.3 Contributions

This dissertation addresses the challenges discussed in the previous section. The main contributions of this dissertation can be summarized as follow.

- We propose SynthDB, an automated approach to synthesize test databases for web applications from scratch. By defining five types of database constraints and identifying them from the interactions of application codebase, database queries, and the database schema, SynthDB outperforms existing database generation techniques. It achieves 62.9% code and 77.1% query coverages, while existing techniques cover only 48.9% code or less. Security evaluations with Burp Suite revealed SynthDB detects more vulnerabilities and uncovers new ones in 17 real-world applications.
- We propose RaceDB, an automated system to detect and verify request race vulnerabilities in database-backed web applications. RaceDB uses Application-aware Request Race Detection (ARD) to identify data dependencies and an automated verification technique to detect execution divergences, reducing false positives. In evaluating 14 real-world PHP web applications, RaceDB outperformed existing techniques, detecting 21 known and 18 new request race cases, while existing tools detected only 13 known and 6 new cases. We reported all 18 new vulnerabilities, and already received CVE numbers for 7.

1.4 Outlines

This dissertation presents program analysis and database synthesizing for enhancing security testing of web applications. The rest of this dissertation is organized as follows.

- Chapter 2 presents SynthDB, a database synthesizing technique for web application security testing. We present details of our design and evaluation results of SynthDB in this chapter.
- Chapter 3 presents RaceDB, an automatic technique for detecting request race in web applications. We present details of our design and evaluation results of RaceDB in this chapter.
- Chapter 4 discusses the limitations of our proposed techniques.
- Chapter 5 reviews and discusses related literature.
- Chapter 6 concludes this dissertation.

Chapter 2

SynthDB: Synthesizing Database via Program Analysis for Security Testing of Web Applications

In this chapter, we present SynthDB, a system that *synthesizes a database from scratch (without any initial databases)* for a web server-side application written in PHP. Specifically, we analyze both a target program and its database schema to derive five types of constraints: 1) schema constraints, 2) query-condition constraints, 3) pre-query constraints, 4) post-query constraints and 5) synchronized-query constraints. The five constraints essentially describe the requirements of a desirable test database that can steer the execution toward the desired path while keeping the database integrity. For example, the schema constraints (obtained from the database schema) describe the requirements for database integrity. Query-condition, pre-query, and post-query constraints are collected by analyzing data- and control-dependence between PHP codebase and SQL queries. Synchronized-query constraints define integrity and consistency rules between multiple database records, and they are obtained by observing multiple `INSERT` and `UPDATE` queries executed synchronously. By solving collected constraints, SynthDB generates a test database containing desirable records that can help cover more program paths with realistic execution context (i.e., complying with the identified integrity requirements). The synthesized database is generic

and can be used by existing dynamic security testing techniques to improve the effectiveness of the testing. Our evaluation with 17 real-world PHP applications shows that SynthDB can generate high-quality test databases that aid dynamic testing techniques to improve the code coverage significantly. Our contributions are summarized as follows:

- We propose SynthDB, an automated approach that synthesizes a test database for database-backed web applications from scratch, without any input and initial database.
- We define five types of constraints for generating a desirable test database. Then, we develop an automated technique that identifies the constraints from interactions between the PHP codebase, the SQL queries, and the database schema.
- Our evaluation with 17 real-world PHP web applications shows that SynthDB outperforms existing state-of-the-art techniques. Databases generated by SynthDB helps achieve 62.9% code and 77.1% query coverages while existing techniques cover 48.9% or less of code and 52.9% or fewer queries.
- We conduct two security analyses using a state-of-the-art vulnerability scanner, Burp Suite [14], to evaluate how SynthDB-generated test databases help the security testing. (1) Running Burp Suite against 189 real-world vulnerabilities. SynthDB detects 76.8% of vulnerabilities while other existing techniques cover 55.7% or fewer. (2) Running Burp Suite to discover new vulnerabilities. SynthDB aid Burp Suite discovers 33 previously unknown vulnerabilities from 5 real-world applications.
- Two additional security tests further show the effectiveness of SynthDB. (1) Conducting the reachability test against the vulnerabilities. SynthDB reaches 80.9% of vulnerabilities while the existing techniques cover 55.3% or less. (2) Running two fuzzers, Wfuzz [110] and webFuzz [86], to evaluate the effectiveness of testing databases. SynthDB-generated databases help achieve the best coverage for the two fuzzers against all 17 programs.
- We plan to publicly release SynthDB to facilitate future research.

Assumptions. We assume that a user who wants to analyze or test a web application depends on a database without providing a database and input. This is a typical scenario in practice, according to our conversations with industry collaborators. Specifically, a real-world database contains various privacy-

```
1   $q1 = mysqli_query($db,
            "SELECT courseid FROM registrations
             WHERE studentid = '$_POST['student']'");        A
2   while($registrations = mysqli_fetch_array($q1)) {         1
3    $q2 = mysqli_query($db,
            "SELECT coursename, teacherid, sectionnum,
                    roomnum, dotw FROM courses
             WHERE courseid = '$registrations[0]' AND
                   semesterid = '$_POST['semester']'");      A
4    while( $courses = mysqli_fetch_array($q2) ) {
5     $days = preg_split('//', $courses[4], -1, ...);
6     for( $j=0;  $j<count($days);  $j++ ) {
7      switch( $days[$j] ) {
8      case 'M':
9        $q3 = mysqli_query($db,
               "SELECT fname, lname FROM teachers
                WHERE teacherid = $courses[1]");             4
10       $teachers = mysql_fetch_row($q3);
11       $mon .= "... $courses[0] ... $teachers[0] ...";     B
12       ...
13       break;
14     case 'T':
15       $q3 = mysqli_query($db,
                "SELECT fname, lname FROM teachers
                 WHERE teacherid = $courses[1]");            5
16       $teachers = mysql_fetch_row($q3);
17       $tue .= "... $courses[0] ... $teachers[0] ...";     B
18       ...
19       break;
20     case 'W':
21       ...
22     }
23  }}}}
24  $tablerow = $mon."</td>".$tue."</td>".$wed."</td>";      B
25  print($tablerow);
```

Figure 2.1: Simplified Code Snippet from SchoolMate [90].

sensitive data, making it difficult to be shared for analysis and testing purposes. Moreover, inputs that can exercise various program paths are also difficult to obtain [86, 59, 58].

2.1 Motivating Example

In this section, we use a real-world web solution called SchoolMate [90] to illustrate how SynthDB synthesizes a database for better security testing. SchoolMate [90] is designed to manage classes, teachers, and students for schools.

Goal. We aim to synthesize a database with desirable content so that when we use a dynamic analysis tool that can identify security vulnerabilities, it can reach (potentially vulnerable) program statements that require certain database records. In particular, we aim to do it without requiring (1) concrete input, (2) an initial database, and (3) any SQL query traces from the users, as those are typically not available in practice.

Vulnerable Code under Testing. Figure 2.1 shows a simplified code snippet from VisualizeRegistration.php which displays a student's weekly schedule. There are three vulnerabilities in this code snippet. First, there are two SQL injection vulnerabilities via '`$_POST`' variables at lines 1 and 3 (A). Second, there is an XSS (Cross-Site Scripting) vulnerability at lines 11, 17 ,and 24~25 (B). Specifically, an attacker can inject a malicious code snippet (i.e., JavaScript code) as '`fname`' and '`lname`' in the `teachers` table (representing the first and last name of a teacher respectively) through `manageTeachers.php` and `AddTeacher.php` (we omit the two PHP files' source code due to the space limit). They are fetched (at lines 10 and 16), injected (at lines 11, 17, and 24), and eventually delivered to the client via `print()` at line 25.

Challenges. In this example, multiple conditions in loops (at lines 2, 4, and 6) and a `switch` statement (at line 7) depend on a database. If the database does not contain records that can satisfy the conditions, parts of the programs guarded by the conditions will not be executed and analyzed. For example, running this program without a database would not be able to exercise the loop body between lines 2~23, failing to test the vulnerable statements (lines 3, 11, and 17).

Existing Database Synthesizing Techniques. Figure 2.2 shows examples of the synthesized database by two state-of-the-art techniques [5, 17]. Note that existing techniques require concrete input to run the program for analysis, e.g., to gather SQL query traces. Hence, we provide concrete values '`12`' and '`202101`' for '`$_POST["student"]`' and '`$_POST["semester"]`' to obtain SQL query traces for [17].

1) Database Schema-based Synthesizing: Figure 2.2-(a) shows an example database generated by techniques focusing on database schema. Note that they do not leverage the provided input and program execution, ignoring the '`12`' and '`202101`' for '`studentid`' and '`semesterid`'. The numbers and strings in the synthesized database are randomly generated. For some values (e.g., '`fname`' and '`lname`' in the

`teachers` table), they randomly choose a value from a predefined list templates (e.g., a list of fake names). Unfortunately, running the program with Figure 2.2-(a) would not pass line 2, since there is no database entries with '`studentid=12`'.

2) Query-based Synthesizing: Figure 2.2-(b) shows an example database reconstructed by techniques leveraging both SQL query traces from concrete executions and the database schema. Observe that '`semesterid`' in the `courses` table and '`studentid`' in the `registrations` table have the values of the provided concrete input (i.e., '`202101`' and '`12`'). This is because the technique's analysis is based on the SQL query traces generated from the execution with the concrete input. Moreover, the synthesized database has the same set of values for '`courseid`' in the `registrations` and `courses` tables, to satisfy the `WHERE` clause's condition at line 3. Specifically, the technique obtains a query trace at line 3 where the value of '`$registrations[0]`' is a randomly generated number inserted in the `regsitrations` table. To satisfy condition '`courseid = $registrations[0]`' in the `WHERE` clause at line 3, it inserts another database entry with the value of '`$registrations[0]`' as '`courseid`', resulting in the two tables have entries with the same '`courseid`' values.

Running the program with the same input and the synthesized database can pass the first and second `while` loops (lines 2 and 4). For example, if the first query (at line 1) returns the first row of the `registration` table (i.e., `regid=0`, `studentid=12`, and `courseid=0`), it satisfies the `WHERE` clause at line 3. Then, the second query at line 3 returns the first row of the `courses` table.

However, it does not satisfy the `switch`'s conditions (at lines 8, 14, and 20) which require the values of `dotw`[1] to have one of the '`M`', '`T`', and '`W`' characters[2]. As shown in Figure 2.2-(b)'s `courses` table, `dotw`'s values are random strings, as they do not analyze how the program uses the values of `dotw`.

SynthDB: Program Analysis based Database Synthesizing. In addition to the database schema and queries, SynthDB takes program semantics into account, to synthesize a database that can satisfy the various program and query conditions so that it can help exercising more code and behaviors of the program under testing. Figure 2.1-❶~❹ points out key queries and program statements analyzed by SynthDB to satisfy all the conditions in the motivating example.

[1] '`dotw`' means 'day of the week'

[2] '`M`', '`T`', and '`W`' represent 'Monday', 'Tuesday', and 'Wednesdays'

Table "courses"

courseid	coursename	teacherid	semesterid	sectionnum	dotw
0	nulla	37	45	ndnn	pvcr
1	maxtime	27	41	wwdx	epox
2	aut	61	62	Tisq	lbnd

Table "registrations"

regid	studentid	courseid
0	10	72
1	93	8
2	59	50

Table "teachers"

teacherid	fname	lname
0	Hailie	Senger
1	Baby	Larson
2	Stanley	Schowalter

(a) Synthesized Database leveraging the Database Schema

Table "courses"

courseid	coursename	teacherid	semesterid	sectionnum	dotw
0	Waited	1589	202101	Paren	r2=xe
12	Student	1589	202101	While	H+!lw
78	televison	-428	202101	Parent	kzUt8

Table "registrations"

regid	studentid	courseid
0	12	0
1	12	12
2	12	78

Table "teachers"

teacherid	fname	lname
0	Room	was
1	Shepherd	Crash
2	student	Absent

(b) Synthesized Database leveraging the Database Schema and Query Traces

Table "courses"

courseid	coursename	teacherid	semesterid	sectionnum	dotw
0	Althea	0	202101	Sherman	M
1	Maryam	1	202101	Mckinney	T
2	Leonard	2	202101	Roberts	W

Table "registrations"

regid	studentid	courseid
0	12	0
1	12	1
2	12	2

Table "teachers"

teacherid	fname	lname
0	Vaughan	Gilmore
1	Hedley	Weeks
2	Victor	Wiley

(c) Synthesized Database by SynthDB

Note: Orange-colored cells indicate the keys of the tables. Red colored values are undesirable values while green colored values are desirable.

Figure 2.2: Generated Synthetic Databases by Existing Techniques and SynthDB.

We use a concolic execution engine to run the program and track values returned from a database. During the execution, we conduct a few different analyses. First, SynthDB identifies and analyzes conditions and relations between database fields in the query to infer desirable values for the fields. Second, if a variable is used in creating queries, SynthDB explore program paths that define the variable through concolic analysis, to identify possible values of the variable in the query. By analyzing program conditions related to the variable, SynthDB can infer constraints of desirable database records. Third, SynthDB tracks values returned from a database and analyzes how they are used in the program. Specifically, predicates and loop conditions depending on values returned from databases are analyzed to infer desirable database records.

SynthDB on the Motivating Example. Figure 2.1 shows how our technique reconstructs the database. First, SynthDB identify that the first query's return (`$q1`) is used in the second query's `WHERE` clause (❶) by tracking `$q1`. It reveals the relationship between the two tables `registrations` and `courses`. Specifically, it indicates that there should exist *database entries with the same '`courseid`' in the two tables*. SynthDB leverages this to correctly generate the '`courseid`' values in the `registrations` table.

Second, the record returned from the second query (at line 3, '`$q2`' and '`$courses`') are also tracked. The value of '`dotw`' is propagated to '`$days`' through `preg_split()` (at line 5, ❷), and used in the `switch` (at line 7, ❸). SYNTHDB identifies desirable values for '`dotw`' ('`M`', '`T`', and '`W`') from the `case` statements' conditions (lines 8, 14, and 20).

Third, SYNTHDB identifies that '`teacherid`' from the `courses` table is used in the third and fourth queries (at lines 9 and 15, ❹ and ❺), suggesting that *there should exist database entries with the same '`teacherid`' value in the two tables (`courses` and `teachers`)*. Observe that values of '`teacherid`' in the `courses` and `teachers` tables are overlapping. They both have '`0`', '`1`', and '`2`' as shown in Figure 2.2-(c). However, in Figure 2.2-(b), values of '`teacherid`' in the `courses` table ('`1589`' and '`-428`') do not overlap with the values in the `teachers` table ('`0`', '`1`', and '`2`').

Summary. The synthesized database by SYNTHDB, presented in Figure 2.2-(c), contains all the desirable database entries, allowing to cover all the program statements shown in Figure 2.1. This test DB will provide a better coverage for further dynamic analysis, such as security scanning [13, 77, 43, 62, 121, 39, 88, 113] or fuzzing [110, 86] (Details in section 2.3).

2.2 Design and Implementation

SYNTHDB aims to synthesize a *comprehensive* database with *integrity*, that can help exercise program paths dependent on the database. In our context, (1) a *comprehensive database* means a database containing sufficient entities satisfying the path conditions of the program under test. (2) A *database of integrity* means that records in the database are feasible and do not conflict with the integrity rules [72] of the program and database. SYNTHDB achieves the properties through the three components in Figure 2.3: (1) the concolic execution engine exploring execution paths of a target program (subsection 2.2.1), (2) the constraint identifier collecting database constraints related to the comprehensiveness and integrity of the database (subsection 2.2.3), and (3) the database generator synthesizing a database by solving the constraints (subsection 2.2.4).

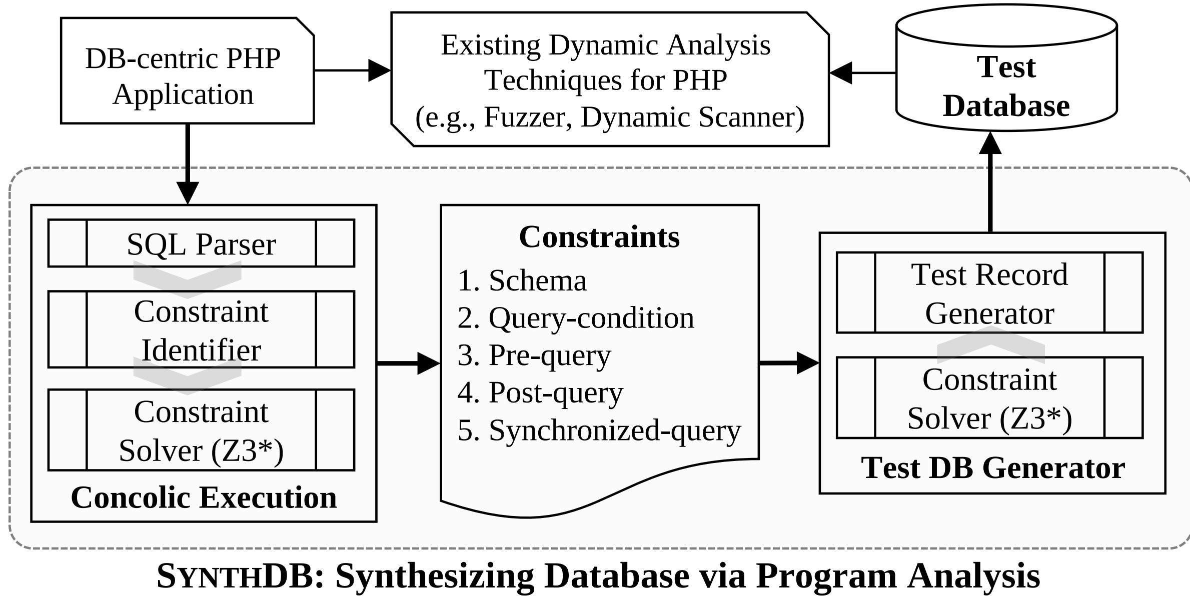


Figure 2.3: Overview of SYNTHDB (*Z3 solver [115]).

2.2.1 Path Exploration via Concolic Execution

We first leverage concolic execution to obtain a set of inputs that can cover diverse program paths. Our concolic execution engine is based on Vulcan Logic Dumper [104], which is an extension of Zend Engine [117]. Similar to other state-of-the-art concolic execution techniques, we concretely execute a PHP program and collect the path constraints during the execution. We then use the Z3 solver [115] to obtain additional inputs that can satisfy the uncovered path conditions.

Variables of Interest. SYNTHDB's concolic execution engine tracks the propagation of (1) inputs from remote users (e.g., `$_POST` or `$_GET`) and (2) variables holding data returned from database (e.g., returns of `mysqli_query()`).

Incremental Path Constrain Solving. We obtain a path condition that can exercise an unexplored path by negating the last branch condition of a previously explored path. Unfortunately, we encounter an excessive number of constraints due to a large number of program paths. Solving them all requires significant time. To address this problem, we leverage our observation that *many program paths overlap* with each other as well as their constraints. To this end, we identify and break down the overlapping constraints and cache resolved constraints' results. In particular, we leverage the cache to *incrementally*

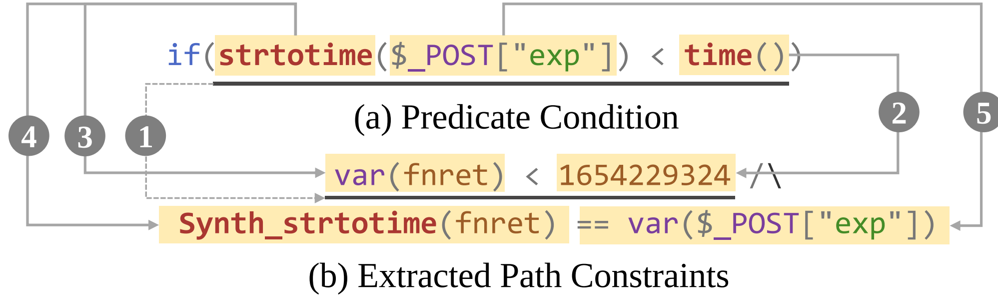


Figure 2.4: Extracting Path Constraints

solve the constraints. When we encounter a set of constraints including already resolved constraints, we solve unresolved (or not cached) constraints and then concatenate the new solution to the cached solutions, updating the cache. This incremental approach essentially mitigates the *path explosion* problem during the path exploration.

Terminating Condition. Since we aim to explore all possible execution paths, it often creates a number of executions, taking a long time to finish the analysis. Hence, our analysis's terminating condition is either it explores all execution paths or reached the time limit of 10 hours.

Handling Path Constraints. During the concolic execution, SynthDB solves various path constraints to cover more program paths. In particular, we obtain path constraints from predicate conditions. Figure 2.4-(a) shows an example predicate where SynthDB extracts the constraints shown in Figure 2.4-(b). Specifically, we first translate the structure of the predicate condition to the constraints (1). Then, we *concretize* all the functions that are not operating the tracked variables. In this example, we obtain the concrete return value of `time()` which is '1654229324' (2). Then, we *create symbolic variables* (e.g., `fnret`) to represent the remaining functions and expressions (3). If a symbolic variable represents a function, we define our own function handler in SynthDB that emulates the target function (e.g., `Synth_strtotime()` emulates `strtotime()`) (4). Finally, we define an additional constraint to relate the symbolic variables (e.g., `fnret`) to the tracked program variable (e.g., `$_POST["exp"]`) (5).

2.2.2 Algorithm: Concolic Execution

Algorithm. algorithm 1 runs as a part of our concolic execution engine. Specifically, after the concolic execution engine processes each instruction, we conduct our analysis to extract the constraints.

It takes an input UNTRUSTEDSRCS that contains untrusted sources. At line 2, our concolic execution engine taints before its analysis, so that any data originated from the untrusted sources will be tracked. Lines 3~46 form a large loop that is executed on every instruction. Specifically, at line 3, we run the concolic execution engine on each instruction (SINGLESTEPCONCOLICEXEC()), and obtain the executed instruction as *ins*. At lines 4~5, we implement a basic taint propagation logic: if *ins.operand* (i.e., an operand or argument of an instruction) is tainted, we taint its outcome (i.e., *result*).

Pre-query constraints (Line 6~13). When a query that inserts or updates the database is executed, we check whether the query is constructed by using program variables. If so, we collect the constraints related to those variables for pre-query constraints. Specifically, if the current instruction calls a DB function (e.g., `mysqli_query()`) to execute an `INSERT` or `UPDATE` query and the query string passed to the function is tainted (lines 9~10), we identify all the tainted variables used to compose the query (line 11) and collect them as a pre-query constraint (lines 12~13).

Post-query constraints (Line 14~18). Post-query constraints are collected from the program code, particularly branches, using the data returned from a database. Specifically, if the instruction is a branch[3], we check whether the branch condition is tainted, and originated from a `SELECT` query (lines 15~16). In other words, we try to identify cases that a `SELECT` query's return is used as a predicate condition such as '`if ($database['field'] == ...)`'. If so, we collect the current query and the instruction as a post-query constraint (lines 17~18). Note that the query represents the source of the constraint and the instruction for the predicate condition associated with the constraint.

Lines 19~22 show that SYNTHDB taints a return value of a DB function (e.g., `mysqli_query()`) if either `SELECT` or `UPDATE` query is executed (lines 21~22), as it retrieves data from a database.

[3] We consider the following opcodes as branch instructions: IS_IDENTICAL, IS_NOT_IDENTICAL, IS_EQUAL, IS_NOT_EQUAL, IS_SMALLER, IS_SMALLER_OR_EQUAL, SWITCH_LONG, SWITCH_STRING, ISSET_ISEMPTY_VAR, ISSET_ISEMPTY_DIM_OBJ, ISSET_ISEMPTY_PROP_OBJ, ISSET_ISEMPTY_CV, ISSET_ISEMPTY_THIS, and ISSET_ISEMPTY_STATIC_PROP.

Query-condition constraints (Line 23~29). Query-condition constraints are obtained from conditional clauses of a query. During the concolic execution, if the current instruction calls a DB function (e.g., `mysqli_query()`) with a `SELECT` or `UPDATE` query (line 26), we check the query passed to the function to see whether it has conditional clauses (e.g., `WHERE`, `JOIN`, and `HAVING`) at line 27. If it has, we collect the query as a query-condition constraint (lines 28~29).

Synchronized-query constraints (Line 30~46). SynthDB identifies synchronized-query constraints when queries that are always executed together (i.e., queries within the same basic block) are affected by the same program variables. Specifically, we first maintain a list of queries that are executed within the same basic block (lines 31~35). We identify consecutive queries by adding any `INSERT` or `UPDATE` queries to the set which we reset on a branch instruction, which indicates the beginning of a new basic block (lines 31~32).

With the consecutive query list, when a database function is invoked with an `INSERT` or `UPDATE` query (line 39), we check whether the query is constructed by tainted program variables (line 40). If so, we iterate all the consecutive queries within the basic block and their tainted variables (lines 41~42). If the current query and one of the consecutive queries have common tainted variables (lines 43 and 44), it means that those queries are constructed from a same program variable, resulting in synchronized-query constraints (line 45~46).

2.2.3 Identifying Database Constraints via Concolic Execution

Database Constraints. To synthesize a comprehensive database with integrity, we define and collect five types of database constraints: (1) Schema Constraints, (2) Query-condition Constraints, (3) Pre-query Constraints, (4) Post-query Constraints and (5) Synchronized-query Constraints. Note that except for the *schema constraints* which are directly derived from the database schema, the other four database constraints are inferred from interactions between the SQL schema, queries, and program code. Specifically, we focus on analyzing data- and control-dependencies in and between SQL queries and program code by leveraging our concolic execution engine. Next, we explain each constraint with examples.

Algorithm 1: Obtaining Database Constraints

Input : UNTRUSTEDSRC: a list of five untrusted sources ($_GET, $_POST, $_REQUEST, $_SESSION, and $_COOKIE).
Output : Collected four types (pre-query, query-condition, post-query, and synchronized-query) of constraints.

1 **procedure** *ObtainDatabaseContraints*
2 Taint(untrustedSrc)
3 **for** *ins* ← *SingleStepConcolicExec()* **do**
4 **if** *isTainted(ins.operand)* **then**
5 Taint(ins.result)
6 **// Extract the pre-query constraint**
7 **if** *isDBFunctionCall(ins)* **then**
8 queryGetFuncArgs(ins)
9 **if** *isTainted(query) and*
10 *queryType(query) = (insert or update)* **then**
11 taintedgetTaintedVars(query)
12 $\text{Constraints}_{pre-query}$ $\text{Constraints}_{pre-query}$ ∪
13 <query, tainted>
14 **// Extract the post-query constraint**
15 **if** *isBranch(ins.opcode) and*
16 *queryType(TaintSource(ins.operand)) = select* **then**
17 $\text{Constraints}_{post-query}$ $\text{Constraints}_{post-query}$ ∪
18 <TaintSource(ins.operand), ins>
19 **if** *isDBFunctionCall(ins) and*
20 *queryType(GetFuncArgs(ins)) =*
21 *(select or update)* **then**
22 Taint(GetFunctionReturn(ins))
23 **// Extract the query-condition constraint**
24 **if** *isDBFunctionCall(ins)* **then**
25 queryGetFuncArgs(ins)
26 **if** *queryType(query) = (select or update) and*
27 *getConditionClause(query) ≠ ∅* **then**
28 $\text{Constraints}_{query-cond}$ $\text{Constraints}_{query-cond}$ ∪
29 <query>
30 **// Extract the synchronized-query constraint**
31 **if** *isBranch(ins.opcode)* **then**
32 Consecutive-queries ∅
33 **else if** *isDBFunctionCall(ins) and queryType(GetFuncArgs(ins)) = (insert or update)* **then**
34 queryGetFuncArgs(ins)
35 Consecutive-queriesConsecutive-queries ∪ <query>
36 **if** *isDBFunctionCall(ins)* **then**
37 queryGetFuncArgs(ins)
38 **if** *isTainted(query) and*
39 *queryType(query) = (insert or update)* **then**
40 taintedgetTaintedVars(query)
41 **for** ∀ $query_{consec}$ ∈ *Consecutive-queries* **do**
42 tainted_{consec} getTaintedVars(query_{consec})
43 sharedtainted ∩ tainted_{consec}
44 **if** *shared* ≠ ∅ **then**
45 $\text{Constraints}_{sync-query}$ $\text{Constraints}_{sync-query}$
46 ∪ <query, query_{consec}, shared>
47 **return** <$\text{Constraints}_{pre-query}$, $\text{Constraints}_{query-cond}$,
48 $\text{Constraints}_{post-query}$, $\text{Constraints}_{sync-query}$>

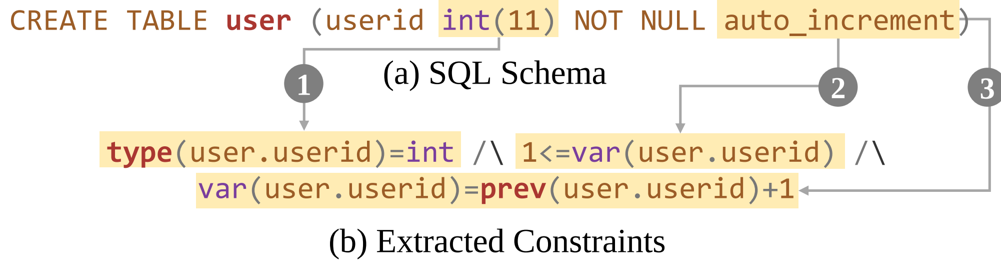


Figure 2.5: Extracting Schema Constraints from Database Schema.

Schema Constraints

Database schema defines the structure of a database to ensure database operations (e.g., data insertion and updating) are performed in a consistent way without violating the integrity of database records. The database integrity requires the records to satisfy three properties:

1. Structural properties between database fields (inferred from `KEY`, `PRIMARY KEY`, and `UNIQUE KEY` keywords).

2. Value range properties (from data types, e.g., `INT` and `DATETIME`, value specifications, e.g., `AUTO_INCREMENT`, and value filtering keywords, e.g., `CHECK`).

3. Table relationships via foreign keys (i.e., '`FOREIGN KEY`').

Database schema files are written in Data Definition Language (DDL). SynthDB uses JSQL-Parser [50] to extract databases' structures and specifications.

Challenges. A single definition may lead to multiple (implicit) constraints. For instance, `PRIMARY KEY` implies the value is (1) not null and (2) unique within the table. The `DATETIME` type indicates that the value is a string with a specific format. Hence, we model each definition and corresponding constraints. In addition, during the constraint extraction analysis, we consider multiple tables' definitions together (e.g., `FOREIGN KEY` specifies properties of another table).

Example. Figure 2.5-(a)'s schema provides three constraints in Figure 2.5-(b). ❶ From the `INT` type, we obtain the constraint that the field's type is an integer. ❷ '`auto_increment`' suggests that its default

initial value is `1` and it will always have a positive value. ③ '`auto_increment`' also indicates that the value will be always incremented by 1.

Query-condition Constraints

We analyze how the outcome of a query is handled (or processed) *before* it is returned to the PHP program. We focus on SQL clauses that operate on the query results such as `WHERE` for filtering and `JOIN` for combining. The query-condition constraints provide information on (1) possible values of a field (or column) and (2) relationships between database records within/across tables.

Challenges. When SQL queries are composed, program variables can be used to specify conditional clauses in the query as shown in Figure 2.6-(a) (see `$regexpr`). It leads to two challenges:

1. *The conditionals depend on the variables' values that are dynamically determined at runtime:* We handle this by leveraging our concolic execution engine to identify possible values to the variable used in the query. In particular, we conduct additional analysis on the constraints for variables used in queries, regardless of the path exploration (i.e., we analyze them even if it does not help explore new program paths). We then collect all the traces of the executed queries and use JSQLParser to parse them.

2. *The semantics of the query depends on the variables' concrete values:* We solve this by analyzing the concretized values in the traces with the context. For example, a string ends with '`%`' under `like` as shown in Figure 2.6-(b) implies it is a regular expression. We model the value patterns and contexts to extract query-condition constraints.

Example. Figure 2.6 shows how SynthDB extracts query-condition constraints from a PHP statement invoking a `SELECT` query. Note that SynthDB works on an executed SQL query trace, meaning that all the PHP program variables and functions are concretized as shown in Figure 2.6-(b): `$regexpr` is concretized to '`player%`' as highlighted in red. We first extract a relationship between `tbluser.id` and `tblfree.userid` from the `JOIN` clause (①). In addition, from the `WHERE` clause, we obtain a constraint that provides the value range of `tbluser.name`. In particular, the `like` keyword is used to

```
mysqli_query("SELECT id, name, points FROM tblusers
              JOIN tblfree ON tblusers.id=tblfree.userid
              WHERE name like '$regexpr'");
```

(a) PHP Code Invoking a SQL Query

```
SELECT id, name, points FROM tblusers
              JOIN tblfree ON tblusers.id=tblfree.userid
              WHERE name like 'player%'
```

(b) Executed SQL Query Trace

```
var(tbluser.id) = var(tblfree.userid)
var(tbluser.name) = Synth_RegEx("player%")
```

(c) Extracted Constraints

Figure 2.6: Extracting Query-condition Constraints.

filter records that match the given regular expression. SYNTHDB converts it to a user-defined function `Synth_RegEx()` that handles regular expressions for the `like` keyword (2).

Pre-query Constraints

PHP programs typically compose SQL queries by concatenating *program variables* (e.g., holding values or field/table names) and constant SQL keywords (e.g., `INSERT` and `SELECT`). The composed queries are passed to SQL functions such as `mysqli_query()`. Note that those variables are defined *before* a query is constructed and often go through various computations and predicates, which essentially *confine* the data values in the query. Pre-query constraints are essentially inferred by analyzing the *computations and predicate conditions*, implying possible values (e.g., ranges or patterns) of database fields.

Challenges. There are two prominent challenges. First, there are multiple sources of constraints from program code and queries: (1) predicates on variables restrict them to not have certain values along the path, (2) there are PHP functions that mutate variables' values (e.g., sanitizing), constraining the values, and (3) SQL functions such as '`PASSWORD()`' also process values before they are stored to the database. We handle them by modeling each source of constraints. Second, constraints from different sources, i.e., program code and SQL query, are combined and accumulated along the paths. Hence, errors in tracking

```
1  $u = mysql_real_escape_string($_POST['user']);
2  $e = mysql_real_escape_string($_POST['email']);      ——— 3
3  $p = mysql_real_escape_string($_POST['pass']);
4  if (preg_match("^[[:alnum:]_]{4,20}$", $u) &&
5      preg_match($emailValidation, $e))      &&         ——— 2
6      preg_match("^[[:alnum:]]{4,20}$",  $p)) {
7    $mysqldb->query("INSERT INTO authors (User, Email, Pwd, Reg)
                      VALUES ('$u', '$e', PASSWORD('$p'), NOW())");  ——— 1
8  }
```

(a) PHP Code Invoking a SQL Query

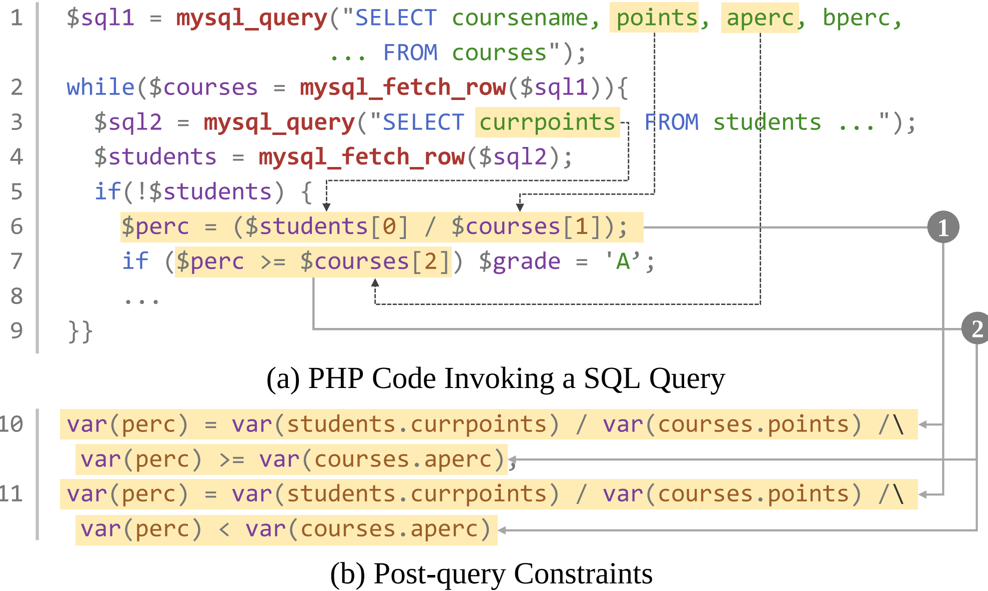
```
9   authors.User == t1  /\ authors.Email == t2 /\ var(authors.Pwd)
10  == SQL.PASSWORD(t3) /\ var(authors.Reg) == SQL.NOW() /\
11  var(t1) == Synth_RegEx("^[[:alnum:]_]{4,20}$") /\
12  var(t2) == Synth_RegEx("^[[:alnum:]][a-z0-9_.-]*@[a-z0-9.-]+\.
                            [a-z]{2,4}$") /\
13  var(t3) == Synth_RegEx("^[[:alnum:]]{4,20}$") /\
14  var(t1) == Synth_mysql_real_escape_string($_POST['user']) /\
15  var(t2) == Synth_mysql_real_escape_string($_POST['email']) /\
16  var(t3) == Synth_mysql_real_escape_string($_POST['pass'])
```

(b) Pre-query Constraints

Figure 2.7: Extracting Pre-query Constraints.

and integrating constraints may lead to substantial analysis failure down the road. To handle this, we make constraints from different sources to be compatible.

Example. Figure 2.7-(a) shows a program that sanitizes (lines 1~3) and validates input values (lines 4~6) before it inserts the values into the database at line 7. Figure 2.7-(b) shows the extracted constraints from the SQL query at line 7 (1), the predicates at lines 4~6 (2), and input sanitization functions at lines 1~3 (3). Observe that we create symbolic variables `t1~3` for program variables used in the query, `$u`, `$e`, and `$p`, respectively. SQL built-in functions are handled by defining our own functions that emulate the original functions (e.g., `SQL.PASSWORD()` and `SQL.NOW()` to generate a hashed password and return the current time, respectively). Observe that the predicates at lines 4~6 have regular expressions which are directly translated into the constraints at lines 11~13, using `Synth_RegEx()` that generates a string value that follows the given regular expression input.



(a) PHP Code Invoking a SQL Query

(b) Post-query Constraints

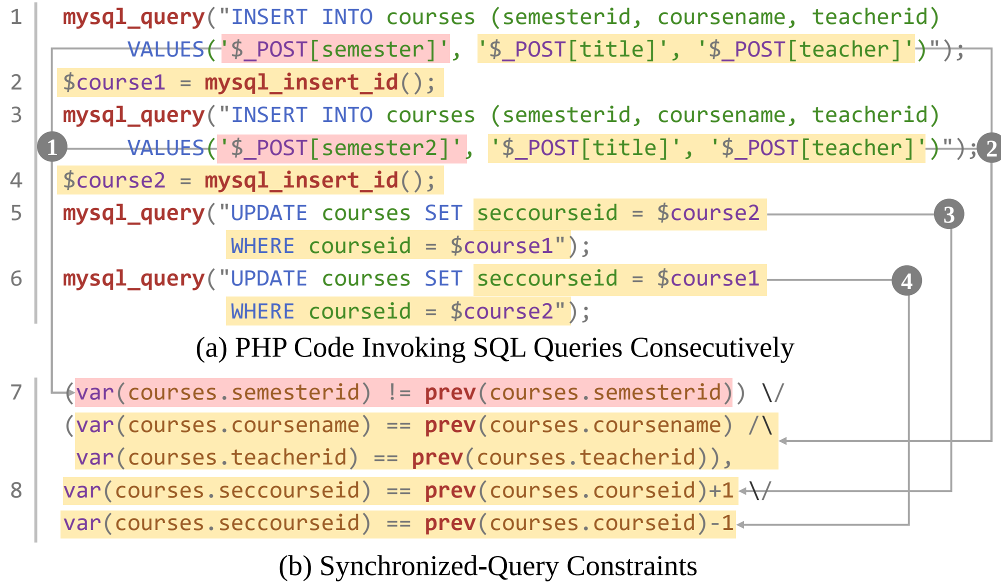
Figure 2.8: Extracting Post-query Constraints.

Post-query Constraints

Typically, results of SQL queries (e.g., return of `mysqli_query()`) are processed by program code (e.g., predicates and functions). For example, programs validate and filter invalid returned data with respect to the database field's semantics (e.g., negative values for an age field). As such, program statements operating on data returned from queries can provide potential values (or value ranges) in the database. To this end, we infer post-query constraints by analyzing program code dependent on the results of queries.

Challenges. To identify post-query constraints, SYNTHDB conducts the taint analysis from the return values of SQL query functions (e.g., `mysql_query()`). Since SYNTHDB analyzes every statement with tainted variables to obtain post-query constraints, over-tainting causes significant false positive cases for post-query constraints[4]. While overall, we conduct conservative taint analysis, for post-query constraint analysis, we configure our taint analysis particularly more conservatively (e.g., do not taint a variable if it is only partially affected by an already tainted variable, such as through bitwise, logical, and comparison operators).

[4]Over-tainting in the pre-query constraint analysis also causes false positives, while its impact is less critical than in the post-query constraint analysis.



(a) PHP Code Invoking SQL Queries Consecutively

(b) Synchronized-Query Constraints

Figure 2.9: Extracting Synchronized-query Constraints.

Example. Figure 2.8-(a) shows a code snippet calculating letter grades from students' score (`students.currpoints`) with respect to the pre-configured percentage value stored in the database (`courses.aperc`) for each letter grade.

To extract the constraints in Figure 2.8-(b), SYNTHDB tracks all the variables holding values returned from queries such as `$courses` and `$students`, via taint analysis. On a predicate condition that uses tainted variables (line 7), SYNTHDB creates constraints from the tainted variables (2) along the data dependencies of the variables (line 6, 1). We also obtain the constraints from the *negation* of the predicate condition to cover the *else condition* of the predicate such as the constraints at line 11.

Synchronized-query Constraints

A program may execute a set of SQL queries *always* together, meaning that the values between the queries will appear consistently on the database. Moreover, if a program variable is used in such queries on multiple tables, it suggests an implicit relationship between the tables (e.g., multiple tables have correlated fields). For example, assume the two consecutive queries:

1. `INSERT into tableA (x, ...) VALUES ($id, ...);`

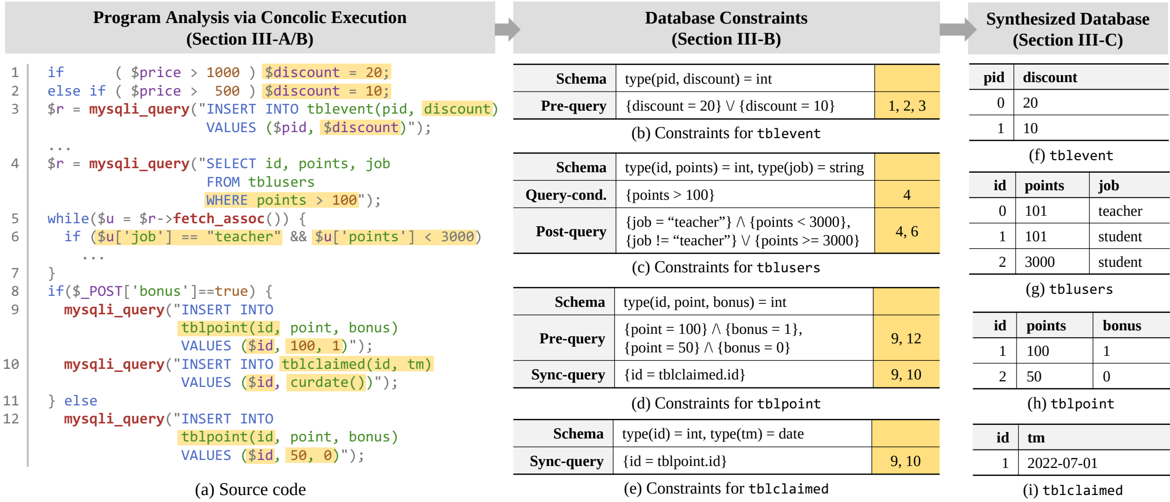


Figure 2.10: Overall Procedure of Synthesizing Database (Highlighted columns in (b)~(e) present the source lines where we extracted constraints from).

2. `INSERT into tableB (y, ...) VALUES ($id, ...);`

By observing that `$id` is used in both queries, we infer the correlation between `tableA.x` and `tableB.y` (i.e., they are identical). To this end, we obtain synchronized-query constraints by identifying queries in the same or subsequent basic blocks, which will be *always executed together*.

Challenges. There are two major challenges. First, beyond the queries executed within the same basic block, queries in multiple basic blocks may always execute together if the basic blocks are always executed along every path. To solve this, we compute dominators [55] from the control flow graphs of the target program. Given queries of a basic block, all the *dominator* basic blocks' queries are executed together. Second, values of database fields between the queries executed together should be analyzed to identify how the queries are synchronized. For instance, two related values can be stored in two different tables. We solve this by comparing all the dependencies between the values used in the queries.

Example. Figure 2.9-(a) shows a program executes four SQL queries consecutively. While the two consecutive queries have different values for `courses.semesterid` (❶), they share variables for the next two fields: `courses.coursename` and `courses.teacherid` (❷). The constraints at line 7 summarize the relationship. The order of the two queries are represented by `prev()`. In addition, at lines 5 and 6, it updates the two inserted rows at lines 1 and 3, so that each of the record will have the other record's id in

`seccourseid`. The constraints at line 8 captures this relationship between the two consecutive queries (3 and 4).

2.2.4 Synthesizing Database

We synthesize a database by solving all the database constraints collected in subsection 2.2.3. Specifically, we iteratively solve the collected constraints to generate concrete data for corresponding fields in tables.

Overall Procedure

Figure 2.10 shows an end-to-end example from the source code to the synthesized database. Specifically, Figure 2.10-(a) shows the target PHP program, and our concolic execution engine identifies database constraints from the highlighted parts of the code and queries. Figure 2.10-(b)~(e) show identified database constraints from the example. Its third column shows which source code line numbers are analyzed to obtain the corresponding constraints. Lastly, Figure 2.10-(f)~(i) present the synthesized database. In the next paragraphs, we illustrate how SynthDB synthesizes each table of a database.

Synthesizing `tblevent`. Observe that `$discount` is defined at lines 1~2, and used in the `INSERT` query at line 3, suggesting the relationship between `tblevent.discount` and `$discount`. Then, from the lines 1~2, the value of `$discount` (and `tblevent.disount`) must be one of the two values: `20` or `10`. This is translated to the pre-query constraints in Figure 2.10-(b). Finally, SynthDB generates database records that satisfy the constraints as shown in Figure 2.10-(f). Note that we do not have constraints for `tblevent.pid`. By default, we use any value from a given data type if a field have no value constraints. In this case, we use 0 and 1.

Synthesizing `tblusers`. A query at line 4 leads to a query-condition constraint in Figure 2.10-(c). In addition, the predicate at line 6 uses the data returned from the query at line 4 as it compares values of the `job` and `points` fields with a string `teacher` and 3,000. Specifically, the first constraint is directly obtained from the predicate condition, while the second constraint is obtained by negating the predicate

conditions which essentially indicates its `else` branch. To this end, SynthDB generates records that satisfy the all the constraints as shown in Figure 2.10-(g).

Synthesizing `tblpoint` and `tblclaimed`. Observe two queries at lines 9 and 12 insert two records to the database with constant values for `point` and `bonus`. They lead to the pre-query constraints in Figure 2.10-(d). In addition, lines 9~10 have two queries that are always executed together, resulting in the synchronized-query constraints: `tblpoint.id=tblclaimed.id`. Note that this also leads to another synchronized-query constraints in `tblclaimed` (Figure 2.10-(e)). Lastly, we generate records satisfy the constraints in Figure 2.10-(h) and (i). First, the two records in `tblpoint` is to satisfy the first pre-query constraint of `tblpoint`. The first records in `tblpoint` and `tblclaimed` have the same `id` value, satisfying the synchronized-query constraint. Note that to satisfy synchronized-query constraints describing inter-table relationships, multiple records across tables are needed.

Domain-Specific Value Generation. We use randomly generated values for database fields that satisfy collected constraints. Purely random values may work fine with automated analysis tools, but they often decrease the readability of the user, especially for certain fields such as "name", "email", and "phone number". To generate a more realistic and readable database, we apply simple heuristic techniques for those fields, similar to existing techniques [17, 5].

Implications of Constraints

The five database constraints are extracted from different sources and have different implications for generating database records. Specifically, schema and pre-query constraints are used to define strict rules that confine the database. They are used to restrict the value range of each field in a table, meaning that all items in a generated database *must* satisfy the constraints. Other constraints, however, such as query-condition, post-query, or synchronized-query constraints, are not as strict as the schema and pre-query constraints. They essentially indicate that there exist *some database records satisfying the constraints*, but not all records must satisfy. While they are less strict, since there are predicates that depend on those constraints, they are crucial in covering more program paths. Query-condition constraints are similar to post-query constraints, as we need at least one record to get a valid return from a `SELECT` query.

Conflicting Constraints. Multiple constraints may have conflicting definitions that cannot be satisfied within a single database. For example, as shown in Figure 2.11, a PHP program that has a `if` and `else` blocks where the first block (❶, line 3) is executed when the `SELECT` query returns less than 100 records while the other block (❷, line 5) requires the query to return 100 or more than 100 records. In other words, with a single database, only one of the two blocks can be covered, meaning that the constraints for the two blocks are conflicting. We discuss other sources of conflicting constraints in section 4.1.

```
1  $r = mysqli_query("SELECT ... FROM ... WHERE ...");
2  if(mysqli_num_rows($r) < 100) {
3      ...  ❶ requiring a database with less than 100 rows
4  } else {
5      ...  ❷ requiring a database with more than or equal to 100 rows
6  }
```

Figure 2.11: Program code requiring two constraints that are conflicting.

Since conflicting constraints cannot be satisfied within a single database, multiple databases need to be used. However, in this project, we focus on a single database that can satisfy the most number of constraints. Hence, we choose a database with the least number of conflicting constraints as output. We manually investigate all the conflicting constraint cases and we miss 8.4% of code coverage and 8.7% of query coverage on average in our evaluation, meaning that our method of choosing the database satisfying the most constraints is effective in practice. We leave handling conflicting constraints as our future work by generating multiple versions of tables or databases.

2.3 Evaluation

We evaluate SynthDB with 17 real-world PHP applications and compare the quality of the synthesized database by SynthDB with three state-of-the-art techniques: EvoSQL [17], Domino [5], and Datafaker [25]. We then execute a dynamic analysis technique for PHP on top of each generated database and compare the observed code and query coverage (subsection 2.3.1). We also conduct three types of security analysis to measure how the test databases affect security testing, including the vulnerability detection

Table 2.1: 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 [90]	63	1,587	15	95	17	32	214	263	School management system
s2	PHP7-Webchess [109]	29	1,505	7	48	14	20	60	94	Web game
s3	Timeclock-1.04 [102]	63	10,820	8	35	18	19	262	299	Employment management system
s4	Mybloggie-2.1.4 [69]	59	3,053	4	24	5	5	74	84	Content management system
s5	Faqforge-1.3.2 [31]	15	302	2	11	3	5	22	30	Online forum
s6	Wackopicko-1.0 [105]	49	720	13	60	13	3	24	40	Photo management system
s7	phpBB-2.0.23 [78]	74	10,798	30	277	44	89	244	377	Online forum
s8	phpBB-3.3.8 [78]	1,091	40,612	69	601	64	341	938	1,343	
s9	OpenCart-3.0.3.8 [71]	1,932	60,515	136	834	246	111	586	943	Ecommerce platform
s10	OpenCart-4.0.0 [71]	2,866	49,018	142	871	258	118	623	999	
s11	WordPress-5.1.2 [112]	901	84,891	12	94	12	32	271	315	Content management system
s12	WordPress-6.0.1 [112]	1,332	110,227	12	94	12	31	264	307	
s13	SMF-2.1.2 [92]	316	45,641	73	525	7	270	929	1,206	Online forum
s14	OsCommerce-2.4.0 [73]	422	15,809	49	343	529	10	377	916	Ecommerce platform
s15	CEPhoenix-1.0.7 [18]	1361	23,938	55	369	149	101	436	686	Ecommerce platform
s16	ZenCart-1.5.7 [116]	1,829	74,960	103	848	394	215	1,311	1,920	Ecommerce platform
s17	Drupal-9.0.0 [27]	8,854	237,001	72	544	39	65	218	322	Content management system
Total		21,256	771,406	802	5,673	1,824	1,466	6,860	10,144	

testing with an active vulnerability scanner, Burp Suite [14] (subsubsection 2.3.2), the reachability test against reported vulnerabilities (subsubsection 2.3.1), and integrating SynthDB with two fuzz testing tools, WFuzz [110] and webFuzz [86] (subsubsection 2.3.1).

PHP Applications for Evaluation. As presented in Table 2.1, we use 17 real-world PHP applications. The first column shows ids (i.e., identifiers) that we will use to refer to applications throughout the section for brevity. The next column show the application name and version, followed by two columns presenting the number of PHP files and the logical lines of code (LLOC). The next two columns show the number of tables and columns, and the following three columns show the number of each `INSERT`, `UPDATE`, and `SELECT` query, respectively. The tenth column shows the total number of those three types SQL queries and the last column presents a brief description of each application. In total, the selected applications include 21,256 PHP files, 771k PHP LLOC, and 10,144 SQL queries.

– *Selection Criteria:* In choosing the target database-backed PHP applications, we consider categories of web applications where the PHP and database are popularly used, including management systems, online forums, eCommerce platforms, web games, and Content Management System (CMS). Moreover, we also consider the frequency and diversity of SQL queries used in the programs. We run statistical analysis on

Table 2.2: Top 75 frequent SQL keywords.

Supported	WHERE, FROM, SELECT, NOT, AND, SET, TABLE, DEFAULT, NULL, AS, DELETE, UPDATE, BY, ON, JOIN, INT, KEY, INSERT, INTO, IN, LEFT, CREATE, IF, DROP, UNSIGNED, PRIMARY KEY, OR, INNER, TINYINT, DISTINCT, MEDIUMINT, VALUES, TEXT, DATETIME, ALTER, SMALLINT, IS, PRIMARY, LIKE, CASE, BIGINT, UNIQUE KEY, BETWEEN, DATE, MEDIUMTEXT, HAVING, ELSE
Not supported	EXISTS, GROUP, INTERVAL
No Effect	ORDER, LIMIT, FOR, COLLATE, DESC, ASC, COALESCE, LOCK, UNLOCK, ENABLE, DISABLE, DATA, THEN, TO, WHEN, END, YEAR, MONTH, LONGTEXT, TRUE, ADMIN USER, REPLACE, FIRST, IGNORE

all 17 applications we evaluated to show that (1) our selected applications include a wide spectrum of SQL queries and (2) SynthDB supports most of those frequently used SQL keywords and functionalities. Specifically, we search all the SQL keywords in PHP source files and schema files. Then, we rank the SQL keywords by the number of appearances.

SQL keyword Statistics. First, there are 213 SQL keywords according to the MySQL version 8.0.24's specification. Among them, 170 keywords are appeared in our selected 17 applications (Table 2.1), meaning that the selected applications include diverse SQL queries (hence those applications are of high quality for the evaluation). Second, a total of 51,510 SQL keywords are used in our selected 17 applications, and the top 25 most frequent keywords are the dominant majority as they are 83.3% of the total extracted keywords. SynthDB supports all those top 25 frequent keywords. For the top 50 frequent SQL keywords, which are 97.5% of the total extracted keywords, SynthDB failed to handle only two of them, `EXISTS` and `GROUP BY`, which appear 691 times out of 51,510. Note that the `GROUP BY` statement is typically used with aggregate functions (`COUNT()`, `MAX()`, `MIN()`, `SUM()`, `AVG()`), which we do not support due to the conflicting constraints. Lastly, we present the top 75 frequent SQL keywords in Table 2.2, which are 99.7% of the total extracted keywords. Among 75 keywords, 25 of them are generic SQL keywords that do not contribute to database reconstruction. SynthDB supports 47 out of the remaining 50 keywords but does not support 3 keywords.

Back to the Selection Criteria:

Table 2.3: Comparison with Baseline Approaches.

	DOMINO	Datafaker	EvoSQL	SYNTHDB
Schema Constraints	✓	✓	✓	✓
Query-condition Constraints	☐	☐	⊟	✓
Pre-query Constraint	☐	☐	☐	✓
Post-query Constraint	☐	☐	☐	✓
Synchronized-query Constraint	☐	☐	☐	✓
Domain-Specific Value Generation	☐	✓	☐	✓

⊟: Supporting `SELECT` queries only.

1. We choose twelve applications (s1~s8, s12, and s15~s17) out of 28 applications that are frequently evaluated by previous work [4, 87, 3, 122]. Specifically, among 28 programs, we exclude 7 applications that have limited database interactions (less than 30 queries, and 9 applications use database engines or PHP versions that SYNTHDB does not support (e.g., MariaDB or PHP version<7).
2. We additionally include five popular real-world applications (s9~s11, s13, and s14) that have large codebase. They are chosen as follows. First, we search for the most popular projects from three categories where the DB-backed PHP is dominant: *CMS*, *eCommerce platform*, and *online forum*. Then, we select the most installed [12] PHP project for each category. We select WordPress [112] and OpenCart [71] for the *CMS* and *eCommerce platform* categories respectively. For the *online forum* category, we select two applications, phpBB [78] and SMF [92], as they have almost the same number of installations (47,631 for phpBB and 47,716 for SMF) as of July 2022.

– *Summary of Existing Techniques:* We compare our technique with three state-of-the-art test database generation techniques, DOMINO [5], Datafaker [25], and EvoSQL [17]. Table 2.3 summarizes the advantages and limitations of them, focusing on which database constraints are supported. First, DOMINO [5] and Datafaker [25] focus on analyzing database schema to synthesize test data that follow integrity rules. While Datafaker uses domain-specific value generation for creating realistic looking test data, both DOMINO and Datafaker do not support four database constraints (i.e., query-condition, pre-query, post-query, and synchronized-query constraints). Second, EvoSQL [17] is a query-aware technique that leverages the genetic algorithm to generate test data. However, it has limited support for the query-condition constraints,

handling the SELECT query only. As shown in the last column, SynthDB supports all five database constraints, as well as domain-specific value generation (subsubsection 2.2.4).

– *Configurations of Existing Techniques:* During our evaluation, we try our best to fairly treat existing techniques. Specifically, EvoSQL takes a list of concrete queries and a schema. We collect all concrete queries from our concolic execution runs for each application and feed them to EvoSQL to generate test databases. We acquire the implementation of Domino and Datafaker from their official sites [89, 25] and feed the database schema for each PHP application to generate test databases. Note that SynthDB improves the effectiveness of testing techniques because (1) our concolic execution engine solves path constraints, allowing many program paths to be tested, and (2) synthesized database enables testing tools to cover more program paths. Unfortunately, existing techniques that we compare with do not have concolic execution engine, making it difficult to measure the effectiveness coming from the synthesized database. To focus on the effectiveness of the synthesized database, we use our concolic execution engine (subsection 2.2.1) for all the existing techniques. In other words, all the experiments in subsection 2.3.1, subsection 2.3.3, and subsubsection 2.3.1 are conducted on top of our concolic execution engine with test databases generated by each technique.

2.3.1 Coverage Evaluation with Test Databases

To evaluate the quality of test databases generated by different techniques, we measure code and SQL query coverages while we execute PHP applications with the concolic execution engine with 10 hours of timeout. As a baseline, we execute each application with a default database that is shipped with the application or generated during the installation.

Code Coverage. We use Xdebug [114] to measure the code coverage. Figure 2.12(a) shows the code coverage result. The concolic executions with SynthDB-generated test databases achieve the best code coverage (63.9% on average). On average, databases generated by EvoSQL, Datafaker, and Domino achieve 48.9%, 38.9%, and 38.3%, respectively. The executions with a default DB achieves 33.0%. Among them, we observe four programs (s3, s9, s11, s17) have significantly lower than (e.g., more than 10%) the

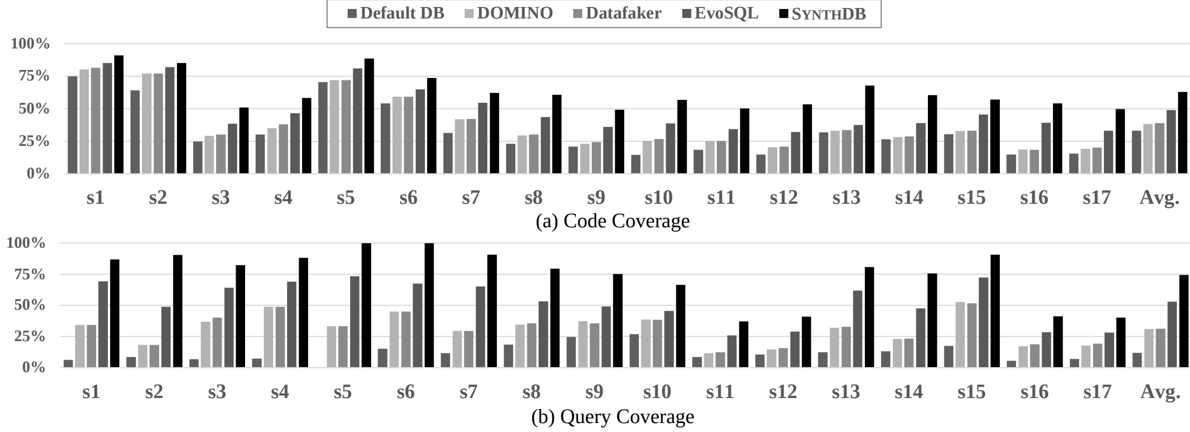


Figure 2.12: Code and Query Coverage of SynthDB ('Default DB' presents an execution with a DB right after the installation. s1~s17 are the ids from Table 2.1).

average code coverage. However, observe that the code coverage with SynthDB synthesized database are consistently higher than the coverage with databases generated by other techniques.

Our manual inspection reveals that there are two major reasons for those low code coverage cases: (1) code requires specific configurations and (2) code requires complex input formats which is challenging for the SMT solver. First, a program may have modules that are unreachable when using the default configuration. To cover those code, one needs to install and configure additional extensions, while in our evaluation, we run all the programs with the default configuration and extensions/plug-ins. For instance, 11.3% of the uncovered code of OpenCart belongs to multi-language support modules, whereas only the English module is activated by default. Also, we observe that 63.6% of uncovered code can be activated only with payment extensions (e.g., Ali-pay or Amazon-pay). Second, program paths may require complex inputs to be covered. For example, to cover OpenCart's email service code, we need to provide valid values for the Simple Mail Transfer Protocol (SMTP) service such as host address, username, password, and port, which are extremely challenging to handle for SMT solvers.

SQL Query Coverage. We statically scan the code to identify the SQL queries used in each PHP application, and we leverage Xdebug to count the executed SQL queries. Queries that return a valid result without an error are counted as covered. The Figure 2.12(b) show that the execution with SynthDB-generated DB can cover 77.1% of SQL queries in PHP applications while test databases by EvoSQL, Datafaker, and Domino can cover 52.9%, 31.3%, and 30.9%, respectively. We also test the query coverage with a default

database. We observe that most `SELECT` and `UPDATE` queries failed because the target items do not exist. However, `INSERT` queries are executed normally, and `SELECT` or `UPDATE` against items inserted by former `INSERT` can also be executed and counted as covered.

Out of 10,144 SQL queries in PHP applications, the current implementation of SynthDB failed to cover 2,832 queries. There are two major reasons for the uncovered queries: (1) 2,173 queries are located in PHP code that we failed to cover the code (due to the limitation of the default configuration or inactivated plug-ins), (2) 659 queries are sub-queries that return empty results and hence are not counted, even if the statements executing them are technically reached. We further analyze the uncovered queries regarding their impact on our analysis. 848 of them contain query constraints that we can also extract from other already covered queries, meaning that missing them does not impact our analysis.

Observations. From the coverage evaluation, the major contributing factor of SynthDB outperforms existing techniques is that SynthDB supports the post-query constraints and query-condition constraints. Specifically, the most common cases that SynthDB can cover while others failed to cover, are the predicates that evaluate values returned from database queries.

2.3.2 Enhancing Existing Security Testing using Test Databases

We evaluate how effectively test databases can aid existing security testing techniques, using a state-of-the-art vulnerability scanner, Burp Suite, and two fuzzers, Wfuzz and webFuzz.

Vulnerability Detection with Burp Suite

We use Burp Suite to demonstrate how SynthDB can help vulnerability detection for database-backed applications. We conduct Burp Suite's active scanning (i.e., automated mode) for applications in Table 2.1 with test databases generated by SynthDB, EvoSQL, Domino, and Datafaker.

Vulnerability Report Collection. First, we collect vulnerability reports for PHP applications listed in Table 2.1, from the CVE database [21], Exploit Database [30], previous research [53, 100, 101], and security reports by application developers[5]. We manually verify each reported vulnerability to prune out false

[5] From public repositories such as GitHub.

positives and uncertain reports that do not provide sufficient details for the vulnerable code's location. After removing 33 false positives (out of 222), we collected information of 189 known vulnerabilities from 11 PHP applications, including 126 cross-site scripting (XSS), 27 SQL injection, and 36 other vulnerabilities (e.g., directory traversal, forceful browsing, remote admin addition, and parameter manipulation). Note that we could not find publicly available vulnerability reports for six applications: phpBB-3.3.8, OpenCart-4.0.0, SMF-2.1.2, OsCommerce2.4.0, CE-Phoenix-1.0.7, and Zencart-1.5.7.

We count the number of vulnerabilities reported by Burp Suite and prune out false positives by manually checking reported vulnerabilities. Specifically, we leverage Burp Intruder [80] to generate a specific exploit for each vulnerability and ensure the detected vulnerability is exploitable. For instance, we forge a request with a generated payload and send it to the server for input-based vulnerabilities (e.g., XSS, SQL injection, and file path traversal). Table 2.4 shows the number of known vulnerabilities and the number of vulnerabilities reported by Burp Suite with each test database. Burp Suite detects the most number of vulnerabilities when it runs with the DB-generated by SynthDB.

33 New Vulnerabilities Discovered. Notably, with SynthDB, Burp Suite detected 33 *previously unreported vulnerabilities* from 5 real-world applications, including 21 XSS vulnerabilities and 12 SQL injection vulnerabilities. We have reported the discovered vulnerabilities to the developers with detailed instructions including how to create the test databases (as they are not reproducible without a proper database). Note that Table 2.4 does not include results for 5 applications (i.e., s8, s10, s13, s15, and s16) because they do not have any known vulnerabilities, and Burp Suite could not detect any new vulnerabilities. Out of 189 vulnerabilities we collected, Burp Suite with SynthDB failed to detect 25 of them. Our further analysis shows that 15 of them are vulnerability types that Burp Suite does not aim to detect [79] (e.g., session/object injection, logical fault, and authentication bypass). Burp Suite with SynthDB failed to detect the remaining 10 vulnerabilities for the following reasons: (1) seven of them require additional external resources, such as local files or network service, (2) two of them require inputs that the SMT solver could not handle (e.g., requires a specific HTTP referer format), (3) the last one requires a specific configuration change.

Table 2.4: Burp Suite results with databases

Id	# Known Vuln.	Default DB	DOMINO	Datafaker	EvoSQL	SYNTHDB
s1	80	7	31	31	62	80 (+13)[#]
s2	18	3	5	5	12	18 (+4)[†]
s3	8	2	2	2	3	7 (+7)[‡]
s4	13	3	4	4	8	9 (+8)[★]
s5	5	3	4	4	5	5
s6	15	5	6	6	8	11
s7	5	1	1	1	3	5
s9	3	0	0	0	0	0
s11	23	6	6	7	9	12
s12	15	3	3	3	4	7
s14	N/A	N/A	N/A	N/A	N/A	N/A (+1)$^{\rho}$
s17	4	0	0	0	1	1

– The number in parentheses indicates the number of new vulnerabilities discovered.
– Background color red, yellow, light-green, and green represent 0%~25%, 25%~50%, 50%~75%, and 75%~100% of known vulnerabilities detected, respectively.
#: Total 93 (13 new vulnerabilities). †: Total 22 (4 new vulnerabilities). ‡: Total 14 (7 new vulnerabilities). ★: Total 17 (8 new vulnerabilities). ρ: 1 new vulnerability.

Reachability of Security Vulnerabilities

Although our experiments with Burp Suite clearly show the effectiveness of SYNTHDB, the limitation of Burp Suite prevents us from identifying a number of known vulnerabilities. To further evaluate how the quality of test databases affects the security testing and analysis, we conduct more generic tests that *do not rely on* a specific tool. Specifically, we conduct a reachability test against reported vulnerabilities for each application.

We execute each PHP application on top of our concolic execution engine with different test databases to measure how many vulnerable statements have been covered. Table 2.5 shows a result. SYNTHDB can successfully reach 80.9% (171 out of 189) vulnerable statements. EvoSQL can cover 55.3% (128 out of 189),

Table 2.5: Reachability Test against PHP Vulnerabilities.

Id	# Vuln.	Default DB	Domino	Datafaker	EvoSQL	SynthDB
s1	80	8	33	33	62	80
s2	18	5	8	8	12	18
s3	8	2	2	2	3	7
s4	13	3	6	6	11	13
s5	5	3	4	4	5	5
s6	15	7	11	11	12	14
s7	5	1	2	2	3	5
s9	3	0	0	0	0	2
s11	23	6	8	9	12	16
s12	15	3	5	5	7	10
s17	4	0	0	0	1	1
Total (%)	189	38 (24.9%)	79 (37.7%)	80 (38.1%)	128 (55.3%)	171 (80.9%)

– Background color red, yellow, light-green, and green represent 0%~25%, 25%~50%, 50%~75%, and 75%~100% of vulnerabilities reached, respectively.

Datafaker reaches 38.1% (80 out of 189), and Domino covers 37.7% (79 out of 189). The execution can only reach 24.9% (38 out of 189) with a default database.

Integrating with Fuzzing Methods

We use two popular fuzzing tools, Wfuzz [110] and webFuzz [86], that are designed for testing web applications. Figure 2.13 shows the code coverage reported by Wfuzz and webFuzz. We use the default setup for each fuzzing test, and we use the timeout of 10 hours for each test. SynthDB-generated database helps to achieve the best coverage for both fuzzing tools in all the cases. On average, webFuzz reports 58.6% code coverage with SynthDB's database while EvoSQL's database achieves 47.4%, Datafaker and Domino get 37.0%, and the executions with an default DB achieve only 30.8%. Wfuzz shows 57.3% code coverage with SynthDB, 46.4% with EvoSQL, 36.0% for both Datafaker and Domino, and it covers only 29.9% in the executions with a default DB.

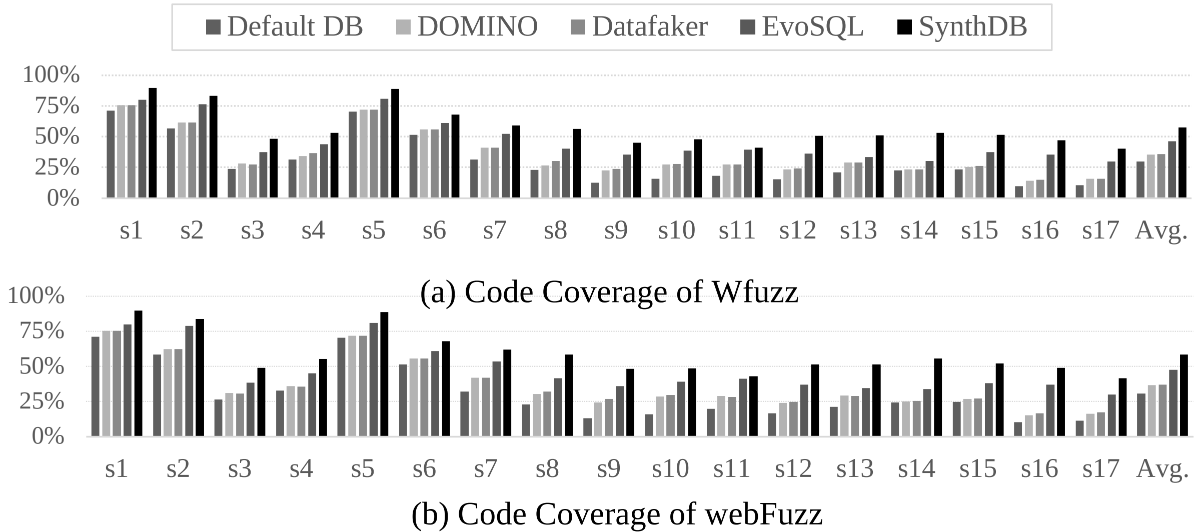


Figure 2.13: Code Coverage Results by Existing Fuzzing Tools.

2.3.3 Runtime Performance Measurement

We measure the time taken for generating a test database by each technique Table 2.6 shows the results. SynthDB takes the longest average time (2.5x compared to EvoSQL) because we comprehensively analyze the program and database, identifying five types of database constraints, requiring multiple runs of concolic execution. EvoSQL [17] is a query-aware technique and analyzes a list of queries. For this evaluation, we assume that the queries are prepared and provided by the user, and we only measure the time taken to generate test data.[6] Datafaker [25] and Domino [5] are much faster than SynthDB and EvoSQL because they only analyze the schema. While SynthDB takes a longer time than others, generating a test database is a *one-time effort* for each PHP application, and the generated database can be reused for different dynamic testing and analysis techniques. Table 2.6 also shows the number of records each technique generates for a test database. From our observation, 41.2% of the time is attributed to constraints solving, 14.3% for running the PHP script, 6.5% for parsing the trace files, 25.8% for database generation and 12.2% for other components.

[6] It would take a longer time than the presented result in practice.

Table 2.6: Time Taken to Generate a Test Database (in minute) and the number of records generated.

	Domino		**Datafaker**		**EvoSQL**		**SynthDB**	
	Time	Rec#	Time	Rec#	Time	Rec#	Time	Rec#
s1	< 1 m	416	< 1 m	500	18 m	458	8 m	421
s2	< 1 m	191	< 1 m	300	6 m	137	4 m	166
s3	< 1 m	139	< 1 m	500	29 m	556	51 m	475
s4	< 1 m	95	< 1 m	300	7 m	152	16 m	133
s5	< 1 m	43	< 1 m	300	1 m	24	2 m	21
s6	< 1 m	239	< 1 m	300	3 m	51	4 m	49
s7	< 1 m	1,108	< 1 m	1,000	41 m	653	82 m	517
s8	< 1 m	2,136	< 1 m	1,500	312 m	1,920	417 m	1,306
s9	< 1 m	3,336	< 1 m	1,500	193 m	1,412	600* m	1,552
s10	< 1 m	3,484	< 1 m	1,500	239 m	1,589	600* m	935
s11	< 1 m	376	< 1 m	500	45 m	601	600* m	514
s12	< 1 m	376	< 1 m	500	51 m	632	600* m	482
s13	< 1 m	2,100	< 1 m	1,500	384 m	2106	600* m	1,307
s14	< 1 m	1,372	< 1 m	1,000	82 m	909	113 m	895
s15	< 1 m	1,476	< 1 m	1,500	318 m	1,610	182 m	1,052
s16	< 1 m	3,392	< 1 m	1,500	271 m	1,573	600* m	1,253
s17	< 1 m	2,176	< 1 m	1,000	68 m	752	600* m	941
Average	< 1 m	1,321	< 1 m	894	122 m	890	299 m	707

#: The number of records generated. *: Reached 10 hours of timeout.

Chapter 3

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

In this chapter, we propose RaceDB, that systematically analyzes database-backed web server applications for race condition issues. RaceDB goes beyond existing approaches [54, 106, 84] 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 [106] 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 gen-

erate 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 [54, 84] 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 plan to publicly release RaceDB upon publication.

```
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 3.1: Motivation Example

3.1 Motivating Example

We use a request race vulnerability found by RaceDB on CE Phoenix Cart [116], 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 3.1-(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).

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 3.1-(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 [106, 54, 84, 75, 120, 35] 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 [54] and ReqRacer [84], 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

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

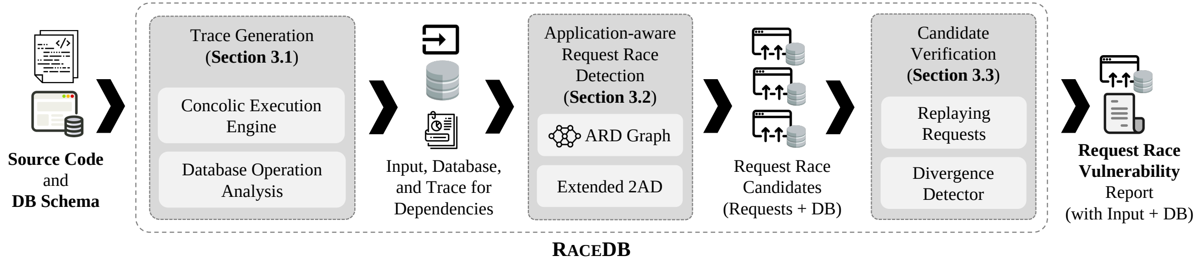


Figure 3.2: System Overview of RACEDB.

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.

3.2 System 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 [120, 93, 49, 65] are ineffective in identifying and detecting request races. As a result, there exist techniques leveraging traces collected from runtime executions [54, 84, 106, 75]. 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 [54, 106] 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 3.2 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 (subsection 3.2.1). Second, the application-aware request detection component constructs an Application-aware Request Race Detection (ARD) graph by analyzing execution traces (subsection 3.2.2). 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 (subsection 3.2.3).

3.2.1 Trace Generation via Concolic Execution

Concolic Execution Engine. Recently, SynthDB [19] 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 3.1-(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 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 3.1-(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 subsection 3.2.2.

3.2.2 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 [106, 54, 84]. 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 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 3.3 shows an ARD graph generated from the motivating example section 3.1. There are two requests (Request `A` and `B`) in this example, where each request containing four database operations

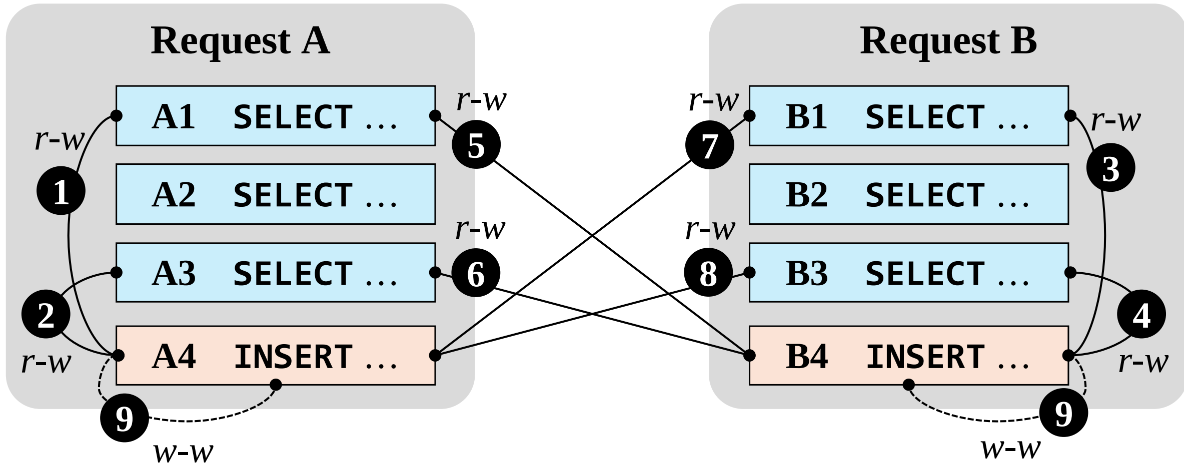


Figure 3.3: ARD graph generated the motivation example.

(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 section 3.1, 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 3.1. 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 [106] with a slight extension. Unlike traditional approaches relying on concrete concurrent traces [2], 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 [106].

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 3.3), 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 section 3.1. 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.

3.2.3 Candidate Verification with Replay Execution

Unlike Raccoon [54] and ReqRacer [84] that either focusing on database access patterns or utilizing error messages for their verification, RaceDB develops an automated approach based on execution divergence between 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` first and waits until it finishes. Then, it execute the request `r2` upon r1's completion (`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 Prox-

ySQL [81], 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.

3.3 Evaluation

This section details the evaluation of RaceDB's effectiveness in detecting request race vulnerabilities. In subsection 3.3.1, we describe the evaluation methodology, including the dataset of real-world web applications, collecting reported vulnerabilities, and the configuration of compared tools. subsection 3.3.2 evaluates RaceDB's ability to identify vulnerabilities compared to existing tools (e.g., Raccoon [54],

Table 3.1: 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 [90]	63	1,587	15	95	17	32	214	263	Content management system
s2	PHP7-Webchess [109]	29	1,505	7	48	14	20	60	94	Web game
s3	OsCommerce-2.4.0 [73]	422	15,809	49	343	529	10	377	916	Ecommerce platform
s4	CE Phoenix Cart-1.0.7 [18]	1,361	23,938	55	369	149	101	436	686	Ecommerce platform
s5	OpenCart-3.0.3.8 [71]	1,932	60,515	136	834	246	111	586	943	Ecommerce platform
s6	MyBB-1.8.15 [68]	312	49,390	75	824	133	379	2,330	2,842	Online forum
s7	OXID eShop-6.0.2 [74]	663	29,021	38	397	43	58	795	896	Ecommerce platform
s8	Moodle-3.11.8 [67]	11,695	741,387	444	4,077	2,138	1,849	12,219	16,206	Ecommerce platform
s9	Drupal-7.6.9 [27]	148	3,315	62	488	230	140	496	866	Content management system
s10	SMF-2.1.2 [92]	316	45,641	73	525	7	270	929	1,206	Online forum
s11	Zen Cart-1.5.7 [116]	1,829	74,960	103	848	394	215	1,311	1,920	Ecommerce platform
s12	phpBB-3.3.8 [78]	1,091	40,612	69	601	64	341	938	1,343	Online forum
s13	WordPress-5.1.2 [112]	901	84,891	12	94	12	32	271	315	Content management system
s14	InvoicePlane-1.5.11 [47]	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	

ReqRacer [84]). To illustrate RaceDB's capabilities in detail, subsection 3.3.3 discusses two specific vulnerabilities detected by RaceDB. subsection 3.3.4 analyzes false positives reported by RaceDB and compares them to the false positive rates of existing tools.

3.3.1 Experimental Setup

To demonstrate the feasibility of our methodology in a practical setting, we developed a prototype of RaceDB in Python. RaceDB is designed 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 [28, 70, 48]. We discuss future extension plans to broaden support for additional web application frameworks and database technologies in section 4.1.

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 [54, 84]. 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 [12], a website profiler that tracks backend technologies and analytics. For example, web applications such as WordPress [112], phpBB [78], InvoicePlane [47], and Zen Cart [116] have thousands of deployments on the Internet. We also included web applications previously tested by other studies [54, 84], such as Open Cart [71], MyBB [68], OXID eShop [74], Moodle [67], and Drupal [27]. 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 [36] 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 [84] and Moodle-2.0.10 [84] 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 chapter, we include 14 web applications, as shown in Table 3.1, 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 subsection 3.2.3. 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 [21], 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 (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 [83] 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 [85] 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 [37] 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 [54, 84]. Additionally, we evaluated the total 14 applications with RaceDB to identify any new request race vulnerabilities. The results are reported in the following sections.

3.3.2 Detection Results

Table 3.2 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 6 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 3.2, 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 subsection 3.2.2. 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 subsection 3.2.3. 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, "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 section 3.1). 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 primarily 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. subsubsection 3.3.3 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.

3.3.3 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 [54, 84].

Figure 3.4: Webchess - Black Player has extra queen.

Webchess Game-breaking (v7)

RaceDB identified a request race vulnerability in Webchess [109], 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 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 3.4. 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: Vulnerable code in Webchess game (v7).

```
1  <?
2  $history = mysqli_query($dbh, "SELECT * FROM history WHERE gameID = (...)");
3  if ($isMoving){
4      $tmpQuery = "INSERT INTO history (...) VALUES (...)";
5      doMove();
6      saveGame();
7  }
8  elseif($history[$numMoves]['curPiece'] == 'pawn' && $history[$numMoves]['promotedTo'] == null)
9  {
10     if($history[$numMoves]['toRow'] == 7 || $history[$numMoves]['toRow'] == 0)
11     {
12         mysqli_query($dbh, "UPDATE history SET promotedTo = '".getPieceName($_POST['promotion'])."' WHERE gameID =
            ↪  ".$_SESSION['gameID']." AND timeOfMove = '".$history[$numMoves]['timeOfMove']."'");
13         saveGame();
14     }
15 }
16
17 function saveGame(){
18     $values[] = collect_pieces_information();
19     // clear old data, then insert new data
20     mysqli_query($dbh, "DELETE FROM pieces WHERE gameID = ".$_SESSION['gameID']);
21     mysqli_query($dbh, "INSERT INTO pieces (gameID, color, piece, row, col) VALUES ($values));
22 }
```

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 information from the `history` table (line 2). Then updates the promotion information in the database, reflecting the white player's choice (line 14). Finally, the `saveGame()` function is called (line 15). 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 6), followed by calling `saveGame()` (line 8). Following these initial actions, the white player's promotion request executes queries at lines 22 and 23. 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 22 and 23) 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.

Zen Cart Double Gifts (v30)

Zen Cart, a popular e-commerce platform used by over 6,900 stores [96], 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

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

```
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 ...
```

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 4) and a subsequent `UPDATE` query (line 9) 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

Listing 3: False Positive Case #1 - Zen Cart

```
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 }
```

write queries between serialized and concurrent executions of the query trace. However, if the 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.

3.3.4 Analysis of False Positives

We discuss false positive cases reported by RaceDB and compares them to Raccoon and ReqRacer. As shown in Table 3.3, RaceDB identified 106 potential request race bugs. Through manual analysis, we confirmed 39 of these to be actual race conditions that could lead to data corruption, application errors, or database errors. However, 67 reports were classified as false positives because they did not affect data integrity or the application's state.

We further investigated the root causes of these false positives and identified two main factors. First, as discussed in subsection 3.2.2, RaceDB employs automated program analysis to identify data dependencies. These dependencies are cruicial 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 false positives. For example, in Listing 3, RaceDB analyzes application-level data dependency in Zen Cart application, appeared at 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

Listing 4: False Positive Case #2 - Webchess

```
1  <?
2  function saveGame(){
3      mysqli_query($dbh, "DELETE FROM pieces WHERE gameID = ".$_SESSION['gameID']);
4      mysqli_query($dbh, "INSERT INTO pieces '(...) VALUES (...);")
5  }
6  
7  function loadGame(){
8      $pieces = mysqli_query("SELECT * FROM pieces WHERE gameID = $_SESSION['gameID']");
9      isInCheck();
10 }
11 function isInCheck(){
12     if($findking){return true;}
13     else{
14         echo("CRITICAL ERROR: KING MISSING!");
15         return false;
16     }
17 }
```

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 false positive 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 3.3 also presents false positives reported by Raccoon and ReqRacer. Raccoon identified a total of 155 races, of which only 18 were confirmed to be actual races (resulting in 137 false positives). Similarly, ReqRacer reported 74 cases, with 17 confirmed races and 57 false positives.

Table 3.2: 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**		

Table 3.3: False Positive Comparison

	Raccoon		**ReqRacer**		**RaceDB**	
	Reported	FP (%)	Reported	FP (%)	Reported	FP (%)
s1	21	18 (85.7%)	7	4 (57.1%)	13	7 (53.9%)
s2	8	7 (87.5%)	3	2 (66.7%)	6	4 (66.7%)
s3	10	9 (90.0%)	9	9 (100%)	11	9 (81.8%)
s4	2	2 (100%)	4	4 (100%)	5	4 (80.0%)
s5	5	2 (40.0%)	2	1 (50.0%)	10	7 (70.0%)
s6	9	4 (44.4%)	6	4 (66.67%)	13	8 (61.5%)
s7	2	1 (50.0%)	5	5 (100%)	3	2 (66.7%)
s8	23	23 (100%)	7	3 (42.9%)	9	5 (55.6%)
s9	26	26 (100%)	2	1 (50.0%)	3	2 (66.7%)
s10	12	12 (100%)	3	2 (66.7%)	3	2 (66.7%)
s11	19	19 (100%)	5	5 (100%)	7	3 (42.9%)
s12	5	5 (100%)	9	8 (88.9%)	8	7 (87.5%)
s13	10	7 (70.0%)	8	5 (62.5%)	10	5 (50.0%)
s14	3	2 (66.7%)	4	4 (100%)	5	2 (40.0%)
Total	155	137 (81.0%)	74	57 (75.1%)	106	67 (63.6%)

Chapter 4

Limitations

4.1 Synthesizing Database via Program Analysis for Web Applications

Other Languages and DBMS. The current version of SynthDB only supports PHP and MySQL database. SynthDB uses JSQLParser [50] to disassemble the recorded query. While it claims to support various DBMS (e.g., MySQL, Oracle, and PostgreSQL), its query analyzer needs to be extended to handle syntax differences between DBMSs (e.g., dialects). For instance, PostgreSQL supports EXCEPT keyword while MySQL does not. To support languages other than PHP, an instruction-level trace, a trace reader, and a parser for the target language need to be developed. We leave this as future work.

Dynamic Schema Changes. There are applications (e.g., WordPress) that allow the installation of plugins or extensions at runtime, and they may change the schema dynamically. While the current implementation of SynthDB does not support dynamically changing database schema during its analysis, it can be used for such plugins and extensions. Specifically, the user can install the plugin first and then run SynthDB to generate a test database that can support plugins for further security analysis. Note that after the plugin is installed, the schema would not be changed further at runtime. To fully handle dynamic schema changes, the final output needs to include multiple database instances to support each of the possible schemas.

Improving Concolic Execution. As discussed in subsection 2.3.1, our concolic execution engine is less effective for applications globally accessing a large number of user inputs. Hence, we plan to develop a guided concolic execution to improve the performance of the concolic execution engine. Specifically, we will identify PHP code that affects or is affected by SQL queries and leverage guided concolic execution techniques to preferentially explore query-related code.

Object-Relational Mapping (ORM). Object-relational mapping (ORM) is a program layer between the language and the database that lets users access data from a database using an object-oriented paradigm. The current version of SynthDB does not support ORM. We observe that ORM implementations vary significantly between the APIs. Supporting them in a generic way is challenging while not impossible. We also observe that some PHP ORM have their own database abstraction layer (DAL), which can be leveraged to abstract the implementation differences. We leave this as future work.

Code Injection Attack. We do not consider the presence of a code injection attack at the time of generating the database. In other words, we assume that the target PHP application and the database schema are not compromised when the user launches SynthDB to generate a test database. We believe that the generated database can help existing security tools to identify code injection vulnerabilities.

Implicit Datatype Conversion. SynthDB infers the datatype by analyzing the definition of the variables and the query. If a datatype differs between the PHP code and the database schema, we implicitly convert the type by prioritizing a more concrete datatype than the other (e.g., mostly from the schema). SynthDB currently supports conversions between three datatypes of PHP (i.e., String, Integer, and Float) and all types of the database.

Conflicting Constraints. In addition to conflicting constraints discussed in subsection 2.2.4, another commonly observed pattern of a conflicting constraint is an error-handling/exception routine that is only executed when a query (e.g., `SELECT`) returns no record. In most cases, such an error-handling is followed by a data processing code that handles returned records from the query. In that case, if a database does not have records, it will only cover the error-handling routine, while if a database has records, it will only cover the data processing code. Other conflicting constraints include SQL queries using aggregation functions, such as `MIN`, `MAX`, and `AVG`. Specifically, suppose there are multiple queries specifying different values for

the same database and table. In that case, multiple databases are required (e.g., if two queries are expecting MIN values of 1 and 2, two databases having the smallest value of 1 and 2 are needed). We leave this for future research.

Overhead and Re-analysis SynthDB's overhead is not trivial as we conduct a more comprehensive analysis than existing techniques. However, we believe that it is acceptable in the context of security testing, where existing dynamic testing techniques (e.g., fuzzing) typically run 6~35 hours. If the target program's source code is updated (potentially also updating the database-related code), SynthDB requires a re-analysis of the updated program. This is a typical limitation of dynamic analysis techniques. SynthDB can be further improved to support incremental analysis. Specifically, we can leverage the existing regression testing techniques [34, 8] while there are additional challenges, such as how to integrate the incremental analysis result to the previous analysis result from the old version of the program. We leave this for future research.

4.2 Detecting Request Race Vulnerabilities in Web Applications

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 subsection 3.3.1, an example of this limitation is the inability to detect cache-related races. Existing tools like ReqRacer [84] 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 subsection 3.3.4, RaceDB currently generates some false positives even after the automated verification step. These false positives primarily come from non-harmful

request races. In these cases, while a race condition is detected and a divergence is observed in the database or application error messages, these inconsistencies are temporary and do not negatively impact user functionality. 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.

Chapter 5

Related Work

5.1 Test Data Generation

Emmi et al. [29] have proposed an automatic test input generation technique for database applications written in Java. Similar to SynthDB, their technique is based on concolic execution to derive input values and database records to explore uncovered application paths. However, their technique focuses on generate concrete SQL query string that can satisfy the symbolic constraints. SynthDB uses concolic executions to identify the five types of database constraints to generate a test database that ensures data integrity while providing valid query results to enable exploring uncovered PHP codebase. In addition, their approach handles a SQL query as string constraints (equality, inequality, and LIKE), and it only supports `WHERE` and `FROM` clauses. SynthDB parses SQL queries to utilize the semantics to recognize database constraints, and it can handle queries with `JOIN` operation.

There exist several approaches to generate test data or a test database to examine SQL queries or database integrity constraints. EvoSQL [17] is a query-aware test data generation technique. It models a test data generation problem as a search-based problem to effectively find an optimal solution that contains test data to cover realistic SQL queries. Other query-aware techniques [52, 97, 10] have been proposed to generate test data to cover various SQL queries. Domino [5] is an automated test data generation technique that aims to systematically exercise the integrity constraints in database schemas. There exist

prior works [63, 64, 118] studying test data generation techniques for exercising and evaluating database integrity constraints.

Recently, JaSoN [111] proposed a systematic test case generation technique for Java applications using MongoDB. It uses a symbolic execution approach to generate executable JUnit test cases. JaSoN applies a versioned schema-approach to generating valid test inputs without relying on an explicit schema. Orthogonal to SYNTHDB, STICCER [6] is a database test suite reduction technique by merging similar test cases.

5.2 Web Application Testing Techniques

Static code scanning is a widely used technique for identifying security vulnerabilities in web applications [44, 41, 56, 9, 24, 66, 99, 107, 119]. 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 [13, 77, 43, 62, 121, 39, 88, 77, 26, 51, 76, 40, 94]. 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 [19] 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 [57] proposed a record-and-replay technique for database-backed web applications, faithfully replaying concurrent bugs.

5.3 Concurrency Bugs in Web Applications

Throughout Chapter 3, we comprehensively discuss the most closely related works, Raccoon [54] and ReqRacer [84], 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 [75, 106] have been adopted in Raccoon. Zheng et al.[120] proposed a static approach to detect race problems in server-side scripts. Furthermore, recent studies[83, 82] have conducted thorough investigations into concurrency problems, including races [83] and deadlocks [82], and their effects on web applications.

5.4 Traditional Race Detection

Race conditions have been widely studied in multi-threaded applications [60, 32, 61] and distributed systems [20, 16, 42]. 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.

Chapter 6

Conclusion

While traditional security analysis techniques for web applications primarily focus on either the application codebase or the backend database, this dissertation aims to bridge the gap by developing techniques to identify the interdependencies between the application codebase, database queries, and the database schema. By analyzing the interplay between application logic and database interactions, we enhance security testing through the synthesis of a comprehensive database and strengthen web applications against request race vulnerabilities using advanced detection algorithms and replay-based automatic verification.

We first presented SynthDB, a technique that synthesizes a database for dynamic security analysis of database-backed web applications. It leverages a concolic execution to identify interactions between the application codebase, database queries, and the database schema, deriving five types of database constraints. SynthDB creates database records by solving the constraints, the generated database can be used to exercise program paths dependent on database queries. Our evaluation with 17 real-world PHP web applications demonstrates that SynthDB outperforms existing state-of-the-art database generation techniques, achieving 62.9% code and 77.1% query coverages while other techniques cover <48.9% code and <52.9% queries. Our security analysis results show that SynthDB could effectively assist existing dynamic security testing approaches, including Burp Suite, Wfuzz, and WebFuzz. Burp Suite aided by SynthDB detects 76.8% of vulnerabilities while other existing techniques cover 55.7% or fewer.

Notably, SynthDB helps to discover 33 previously unknown taint-style vulnerabilities from 5 real-world applications.

In addition, We proposed RaceDB, a novel technique that automatically detects and verifies request races in database-backed web applications. RaceDB analyzes diverse data dependencies within both the application and the database, enabling the identification of intricate race conditions that are often overlooked by existing request race detection techniques, which primarily focus only on database queries. Furthermore, RaceDB's 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. RaceDB successfully identified 21 known vulnerabilities and discovered 18 new vulnerabilities, significantly outperforming existing tools, which detected only 13 known vulnerabilities and 6 new cases.

In summary, SynthDB and RaceDB collectively advance the security testing of database-backed web applications by offering robust methods for database synthesizing and the detection of request race vulnerabilities. These techniques notably enhance the comprehensiveness of security testing and broaden the range of detectable vulnerabilities.

Bibliography

[1] *W. Wordfence, "Critical Vulnerability Patched in Advanced Custom Fields Plugin," Wordfence, March 2023. [Online].* `https://www.wordfence.com/blog/2023/03/critical-vulnerability-patched-in-advanced-custom-fields-plugin/`.

[2] Atul Adya. "Weak consistency: a generalized theory and optimistic implementations for distributed transactions". In: (1999).

[3] Abeer Alhuzali et al. "Chainsaw: Chained automated workflow-based exploit generation". In: *Proceedings of the ACM CCS'16*, pp. 641–652.

[4] Abeer Alhuzali et al. "NAVEX: Precise and scalable exploit generation for dynamic web applications". In: *27th USENIX Security Symposium*. 2018, pp. 377–392.

[5] Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. "DOMINO: Fast and Effective Test Data Generation for Relational Database Schemas". In: *2018 IEEE ICST* (2018).

[6] Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. "STICCER: Fast and Effective Database Test Suite Reduction Through Merging of Similar Test Cases". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)* (2020), pp. 220–230.

[7] Blake Anderson and David McGrew. "Identifying encrypted malware traffic with contextual flow data". In: *Proceedings of the 2016 ACM workshop on artificial intelligence and security*. 2016, pp. 35–46.

[8] Aitor Arrieta et al. "Some Seeds are Strong: Seeding Strategies for Search-based Test Case Selection". In: *ACM Transactions on Software Engineering and Methodology* (2022).

[9] Michael Backes et al. "Efficient and Flexible Discovery of PHP Application Vulnerabilities". In: *IEEE EuroS&P'17* (), pp. 334–349.

[10] Carsten Binnig et al. "QAGen: Generating Query-Aware Test Databases". In: *Proceedings of the 2007 ACM SIGMOD*. New York, NY, USA, 2007. ISBN: 9781595936868.

[11] Kevin Borgolte, Christopher Kruegel, and Giovanni Vigna. "Delta: automatic identification of unknown web-based infection campaigns". In: *Proceedings of the ACM CCS'13*, pp. 109–120.

[12] *BuiltWith*. `https://builtwith.com/`. 2024.

[13] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. "Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists". In: *30th USENIX Security Symposium (USENIX Security 21)*.

[14] *Burp Suite*. `https://portswigger.net/burp`. 2020.

[15] Davide Canali and Davide Balzarotti. "Behind the scenes of online attacks: an analysis of exploitation behaviors on the web". In: *20th Annual Network & Distributed System Security Symposium (NDSS 2013)*. 2013.

[16] Yuhao Cao et al. "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.

[17] Jeroen Castelein et al. "Search-based test data generation for SQL queries". In: *Proceedings of the 40th international conference on software engineering*. 2018.

[18] *CE Phoenix Cart*. `https://phoenixcart.org/`.

[19] An Chen et al. "SynthDB: Synthesizing Database via Program Analysis for Security Testing of Web Applications." In: *NDSS*. 2023.

[20] Guoliang Chen et al. "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.

[21] *Common Vulnerabilities and Exposures*. `https://cve.mitre.org/`. 2021.

[22] *CVE-2022-4037*. `https://nvd.nist.gov/vuln/detail/CVE-2022-4037`. 2023.

[23] Johannes Dahse and Thorsten Holz. "Simulation of Built-in PHP Features for Precise Static Code Analysis." In: *NDSS*. Vol. 14. Citeseer. 2014, pp. 23–26.

[24] Johannes Dahse and Thorsten Holz. "Static Detection of Second-Order Vulnerabilities in Web Applications". In: *USENIX Security Symposium*. 2014.

[25] *Datafaker-Tool for faking data*. `https://github.com/gangly/datafaker`.

[26] Adam Doupé et al. "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner". In: *21st USENIX Security Symposium*. Aug. 2012, pp. 523–538. ISBN: 978-931971-95-9.

[27] *Drupal*. `https://www.drupal.org/`.

[28] M. Elahi et al. "Performance Evaluation of Web Servers for LAMP Stack Web Applications". In: *International Journal of Computer Applications* 166.11 (2017), pp. 20–24.

[29] Michael Emmi, Rupak Majumdar, and Koushik Sen. "Dynamic test input generation for database applications". In: *ISSTA '07*. 2007.

[30] *Exploit Database*. `https://www.exploit-db.com/`. 2021.

[31] *FaqForge*. `https://sourceforge.net/projects/faqforge/files/faqforge/`.

[32] Cormac Flanagan and Stephen N. 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.

[33] *Lucian Constantin. [n.d.]. Withdrawal vulnerabilities enabled bitcoin theft from Flexcoin and Poloniex.* `https://www.pcworld.com/article/2104940/withdrawalvulnerabilities-enabled-bitcoin-theft-from-flexcoin-and-poloniex.html.`. 2014.

[34] Vahid Garousi, Ramazan Özkan, and Aysu Betin-Can. "Multi-objective regression test selection in practice: An empirical study in the defense software industry". In: *Information and Software Technology* 103 (2018).

[35] Milos Gligoric and Rupak 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.

[36] *Google Scholar*. `https://scholar.google.com/`.

[37] *Gor*. `https://github.com/adjust/gor`. 2024.

[38] The PHP Group. *DPHP runkit book*. `http://php.net/manual/en/book.runkit.php`. 2016.

[39] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation". In: *IEEE Transactions on Software Engineering* 34 (2008), pp. 65–81.

[40] Byron Hawkins and Brian Demsky. "ZenIDS: Introspective Intrusion Detection for PHP Applications". In: *IEEE/ACM 39th International Conference on Software Engineering* (2017), pp. 232–243.

[41] Mark Hills, Paul Klint, and Jurgen J. Vinju. "An empirical study of PHP feature usage: a static analysis perspective". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (2013).

[42] Xiang Huang et al. "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.

[43] Yao-Wen Huang et al. "A testing framework for Web application security assessment". In: *Comput. Networks* 48 (2005), pp. 739–761.

[44] Yao-Wen Huang et al. "Securing web application code by static analysis and runtime protection". In: *WWW '04*. 2004.

[45] *Jack Cable. [n.d.]. Race Condition in Redeeming Coupons.* .`https://hackerone.com/reports/157996`. 2016.

[46] Luca Invernizzi et al. "Evilseed: A guided approach to finding malicious web pages". In: *2012 IEEE symposium on Security and Privacy*. 2012.

[47] *InvoicePlane*. `https://www.invoiceplane.com/`.

[48] 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.

[49] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. "Pixy: A static analysis tool for detecting web application vulnerabilities". In: *IEEE Symposium on Security and Privacy (S&P'06)*. IEEE. 2006.

[50] *JSqlParser*. `https://github.com/JSQLParser/JSqlParser`. 2021.

[51] Stefan Kals et al. "SecuBat: a web vulnerability scanner". In: *WWW '06*. 2006.

[52] Shadi Abdul Khalek et al. "Query-aware test generation using a relational constraint solver". In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2008, pp. 238–247.

[53] Adam Kiezun et al. "Automatic creation of SQL Injection and cross-site scripting attacks". In: *IEEE 31st International Conference on Software Engineering* (2009), pp. 199–209.

[54] Simon Koch et al. "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.

[55] Thomas Lengauer and Robert Endre Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph". In: *ACM Trans. Program. Lang. Syst.* 1.1 (Jan. 1979), pp. 121–141. ISSN: 0164-0925. DOI: 10.1145/357062.357071.

[56] Penghui Li and W. Meng. "LChecker: Detecting Loose Comparison Bugs in PHP". In: *Proceedings of the Web Conference 2021* (2021).

[57] Qian Li et al. "R3: Record-Replay-Retroaction for Database-Backed Applications". In: *Proc. VLDB Endow.* 16.11 (July 2023), pp. 3085–3097. ISSN: 2150-8097. DOI: 10.14778/3611479.3611510. URL: https://doi.org/10.14778/3611479.3611510.

[58] Yuan-Fang Li, Paramjit K. Das, and David L. Dowe. "Two decades of Web application testing - A survey of recent advances". In: *Inf. Syst.* 43 (), pp. 20–54.

[59] Giuseppe A. Di Lucca and Anna Rita Fasolino. "Testing Web-based applications: The state of the art and future trends". In: *Inf. Softw. Technol.* (2006).

[60] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. "LiteRace: Effective sampling for lightweight data-race detection". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 134–143.

[61] Umang Mathur and Mahesh 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.

[62] Sean McAllister, Engin Kirda, and Christopher Krügel. "Leveraging User Interactions for In-Depth Testing of Web Applications". In: *RAID*. 2008.

[63] Phil McMinn, Chris J. Wright, and Gregory M. Kapfhammer. "The Effectiveness of Test Coverage Criteria for Relational Database Schema Integrity Constraints". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25 (2015), pp. 1–49.

[64] Phil McMinn et al. "SchemaAnalyst: Search-Based Test Data Generation for Relational Database Schemas". In: *IEEE International Conference on Software Maintenance and Evolution* (2016), pp. 586–590.

[65] Ibéria Medeiros, Nuno Neves, and Miguel 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.

[66] Maliheh Monshizadeh, Prasad Naldurg, and Venkat Venkatakrishnan. "MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications". In: *Proceedings of the ACM CCS'14* ().

[67] *Moodle*. https://moodle.org/.

[68] *MyBB*. https://mybb.com/.

[69] *myBloggie*. https://sourceforge.net/projects/mybloggie/files/mybloggie/.

[70] E. Naramore et al. *Beginning PHP5, Apache, MySQL Web Development*. John Wiley & Sons, 2005.

[71] *OpenCart*. https://www.opencart.com/.

[72] Oracle. *Data Integrity*. https://docs.oracle.com/cd/B19306_01/server.102/b14220/data_int.htm.

[73] *OsCommerce*. https://www.oscommerce.com/.

[74] *OXID eShop*. https://www.oxid-esales.com/en/.

[75] Roberto Paleari et al. "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.

[76] Giancarlo Pellegrino and Davide Balzarotti. "Toward Black-Box Detection of Logic Flaws in Web Applications". In: *NDSS*. 2014.

[77] Giancarlo Pellegrino et al. "Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Dallas, Texas, USA, 2017, pp. 1757–1771. ISBN: 9781450349468.

[78] *phpBB*. https://www.phpbb.com/.

[79] *Issue Definitions - Burp Suite*. https://portswigger.net/kb/issues.

[80] PortSwigger. *Burp Intruder*. https://portswigger.net/burp/documentation/desktop/tools/intruder.

[81] *proxySQL*. https://proxysql.com/.

[82] Zhengyi Qiu et al. "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. DOI: 10.1109/ISSRE52982.2021.00059.

[83] Zhengyi Qiu et al. "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. DOI: 10.1145/3524842.3528463.

[84] Zhengyi Qiu et al. "Understanding and detecting server-side request races in web applications". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021.

[85] *Reqracer artifact*. https://github.com/caseqiu213/reqracer_fse_artifact. 2024.

[86] Orpheas van Rooij et al. "WebFuzz: Grey-Box Fuzzing for Web Applications". In: *Computer Security – ESORICS 2021*. Darmstadt, Germany: Springer-Verlag, 2021. ISBN: 978-3-030-88417-8. DOI: 10.1007/978-3-030-88418-5_8.

[87] Orpheas van Rooij et al. "webFuzz: Grey-Box Fuzzing for Web Applications". In: *ESORICS*. 2021.

[88] P. Saxena, David A. Molnar, and Benjamin Livshits. "SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications". In: *CCS '11*. 2011.

[89] *SchemaAnalyst*. https://github.com/schemaanalyst/schemaanalyst.

[90] *SchoolMate*. https://sourceforge.net/projects/schoolmate/files/SchoolMate/.

[91] Monirul Sharif et al. "Eureka: A framework for enabling static malware analysis". In: *European Symposium on Research in Computer Security*. Springer. 2008.

[92] *Simple Machines Forum*. https://www.simplemachines.org/. 2022.

[93] Justin Smith, Lisa Nguyen Quang Do, and Emerson 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.

[94] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. "Diglossia: detecting code injection attacks with precision and efficiency". In: *Proceedings of the ACM conference on Computer & communications security* (2013).

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

[96] *Store Leads*. https://storeleads.app/reports/zencart.

[97] María José Suárez-Cabal et al. "Incremental test data generation for database queries". In: *Automated Software Engineering* 24.4 (2017), pp. 719–755.

[98] Sucuri. *Website Threat Research Report*. https://sucuri.net/wp-content/uploads/2020/01/20-sucuri-2019-hacked-report-1.pdf. 2019.

[99] Fangqi Sun, Liang Xu, and Zhendong Su. "Detecting Logic Vulnerabilities in E-commerce Applications". In: *NDSS*. 2014.

[100] *Security Testing Report*. https://github.com/carloFanc/Security-Testing/blob/main/FinalReportCarloFanciulli.pdf. 2020.

[101] *Security Testing Project*. https://github.com/davidepedranz/security_testing_project/blob/master/report/vulnerabilities.pdf. 2017.

[102] *Timeclock*. https://sourceforge.net/projects/timeclock/files/Timeclock/.

[103] Verizon. *Data Breach Investigations Report*. https://enterprise.verizon.com/resources/reports/2021-data-breach-investigations-report.pdfx.

[104] *Vulcan Logic Dumper*. https://derickrethans.nl/projects.html. 2016.

[105] *WackoPicko Vulnerable Website*. https://github.com/adamdoupe/WackoPicko. 2018.

[106] Todd Warszawski and Peter 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.

[107] Gary Wassermann and Zhendong Su. "Sound and precise analysis of web applications for injection vulnerabilities". In: *PLDI '07*. 2007.

[108] *Web Application Vulnerabilities: Attacks Statistics for 2018*. `https://www.ptsecurity.com/ww-en/analytics/web-application-vulnerabilities-statistics-2019/`. 2019.

[109] *Webchess*. `https://github.com/halojoy/PHP7-Webchess`.

[110] *Wfuzz – The Web Fuzzer*. `https://github.com/xmendez/wfuzz`. 2020.

[111] Hendrik Winkelmann and Herbert Kuchen. "Symbolic Execution of NoSQL Applications Using Versioned Schemas". In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC '21. Virtual Event, Republic of Korea, 2021, pp. 1778–1787. ISBN: 9781450381048. DOI: `10.1145/3412841.3442050`.

[112] *WordPress*. `https://wordpress.com/`.

[113] Peter M Wrench and Barry VW Irwin. "Towards a Sandbox for the Deobfuscation and Dissection of PHP Malware". In: *2014 Information Security for South Africa*. IEEE. 2014, pp. 1–8.

[114] *xdebug*. `https://xdebug.org/`. 2024.

[115] *Z3*. `https://github.com/Z3Prover/z3`. 2024.

[116] *Zen Cart*. `https://www.zen-cart.com/`.

[117] *The PHP Interpreter*. `https://github.com/php/php-src`. 2021.

[118] Jian Zhang, Chen Xu, and S-C Cheung. "Automatic generation of database instances for white-box testing". In: *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. IEEE. 2001, pp. 161–165.

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

[120] Yunhui Zheng and Xiangyu Zhang. "Static detection of resource contention problems in server-side scripts". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 584–594.

[121] Yuchen Zhou and David Evans. "SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities". In: *USENIX Security'14*.

[122] Yunxiao Zou et al. "Virtual DOM Coverage for Effective Testing of Dynamic Web Applications". In: *Proceedings of ISSTA'14*. San Jose, CA, USA, 2014, pp. 60–70. ISBN: 9781450326452. DOI: 10.1145/2610384.2610399.