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

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

JiHo Lee
University of Virginia
jiholee@virginia.edu

Basanta Chaulagain
University of Georgia
basanta.chaulagain@uga.edu

Yonghwi Kwon
University of Virginia
yongkwon@virginia.edu

Kyu Hyung Lee
University of Georgia
kyuhlee@uga.edu

***Abstract*—Testing database-backed web applications is challenging because their behaviors (e.g., control flow) are highly dependent on data returned from SQL queries. Without a database containing sufficient and realistic data, it is challenging to reach potentially vulnerable code snippets, limiting various existing dynamic-based security testing approaches. However, obtaining such a database for testing is difficult in practice as it often contains sensitive information. Sharing it can lead to data leaks and privacy issues.**

In this paper, we present SYNTHDB, a program analysis-based database generation technique for database-backed PHP applications. SYNTHDB leverages a concolic execution engine to identify interactions between PHP codebase and the SQL queries. It then collects and solves various constraints to reconstruct a database that can enable exploring uncovered program paths without violating database integrity. Our evaluation results show that the database generated by SYNTHDB outperforms state-of-the-arts database generation techniques in terms of code and query coverage in 17 real-world PHP applications. Specifically, SYNTHDB generated databases achieve 62.9% code and 77.1% query coverages, which are 14.0% and 24.2% more in code and query coverages than the state-of-the-art techniques. Furthermore, our security analysis results show that SYNTHDB effectively aids existing security testing tools: Burp Suite, Wfuzz, and webFuzz. Burp Suite aided by SYNTHDB detects 76.8% of vulnerabilities while other existing techniques cover 55.7% or fewer. Impressively, with SYNTHDB, Burp Suite discovers 33 previously unknown vulnerabilities from 5 real-world applications.

I. INTRODUCTION

Web servers deliver web pages to clients in order to provide web services to businesses and customers. Under the hood, upon a client's request, a web server runs a program on the server-side to process the client's request and generate the requested data to be displayed on the client-side (i.e., web browsers). As those server programs serve a large number of clients every day, they become a major target of cybercriminals [1]–[4]. Specifically, vulnerable server programs impose significant security concerns in practice because once exploited, a cyber attacker may compromise all the future client users of the server, causing catastrophic consequences. As a result, finding and fixing vulnerabilities before cyber attackers exploit them is of utmost importance.

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24632
www.ndss-symposium.org

There has been a line of research in analyzing web server programs statically and dynamically to find security issues (e.g., vulnerabilities) and harden them [5]–[8]. However, the execution of web server programs largely depends on the database content, which is highly dynamic. Moreover, database content are often used in dynamic language primitives (e.g., `eval()` for dynamic code generation), imposing significant challenges. Depending on the database content (i.e., data points), many parts of the programs can only be exercised, or they exhibit different behaviors. Static analysis techniques [9]–[11] have difficulty handling dynamically generated code and insufficient execution context due to the lack of a concrete database. While dynamic analysis techniques [12], [13] (i.e., analyzing concrete program execution) do not suffer from the dynamically generated code and execution context, they also *require a database with diverse content (i.e., data points) to be provided* for a successful analysis. Otherwise, they fail to cover and analyze program code dependent on the database. Unfortunately, despite the importance of the database in the security analysis, obtaining a realistic database for testing in practice is challenging. From our conversation with the industry collaborators, sharing a database of a real-world website is extremely difficult because of the privacy concerns raised by sensitive content in the database.

Multiple database synthesizing approaches are proposed to aid various program analysis and testing techniques. Typically, they generate a synthesized database by analyzing database schema and database queries in an application (i.e., query traces). In particular, the majority of them [14]–[17] focus on database schema (i.e., the definition of the database tables and entries) , which contains relational constraints of database entries (e.g., a foreign key). However, they fail to capture implicit relationships between database entries established by the program code (e.g., database entries that are dependent on others or always processed together). For example, recently EvoSQL [18] leverages SQL query traces and the database schema, capturing relational constraints exhibited in the queries. However, it only focuses on database queries, without taking the program code that handles the query results into account. Moreover, its analysis depends on the quality of the query traces provided by a user, while obtaining a comprehensive query trace is also typically dependent on the quality of the database: programs execute many queries based on previous query results (e.g., retrieving detailed data after narrowing down a specific data entry).

In this paper, 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 [19], 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 [20] and webFuzz [21], 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-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 [21]–[23].

```
1  $q1 = mysqli_query($db,
           "SELECT courseid FROM registrations
            WHERE studentid = '$_POST['student']'");                    A
2  while($registrations = mysqli_fetch_array($q1)) {
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]");
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]");
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);
```

Fig. 1. Simplified Code Snippet from SchoolMate [24].

II. MOTIVATING EXAMPLE

In this section, we use a real-world web solution called SchoolMate [24] to illustrate how SYNTHDB synthesizes a database for better security testing. SchoolMate [24] 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 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

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.

Fig. 2. Generated Synthetic Databases by Existing Techniques and SYNTHDB.

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 shows examples of the synthesized database by two state-of-the-art techniques [18], [25]. 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 [18].

1) Database Schema-based Synthesizing: Figure 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-(a) would not pass line 2, since there is no database entries with '`studentid=12`'.

2) Query-based Synthesizing: Figure 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-(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 1-❶~❹ points out key queries and program statements analyzed by SYNTHDB to satisfy all the conditions in the motivating example.

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

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

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

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 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-(c). However, in Figure 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-(c), contains all the desirable database entries, allowing to cover all the program statements shown in Figure 1. This test DB will provide a better coverage for further dynamic analysis, such as security scanning [13], [26]–[32] or fuzzing [20], [21] (Details in Section IV).

III. 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 [33] of the program and database. SYNTHDB achieves the properties through the three components in Figure 3: (1) the concolic execution engine exploring execution paths of a target program (Section III-A), (2) the constraint identifier collecting database constraints related to the comprehensiveness and integrity of the database (Section III-B), and (3) the database generator synthesizing a database by solving the constraints (Section III-C).

A. Path Exploration via Concolic Execution

We first leverage concolic execution to obtain a set of inputs that can cover diverse program paths. Our concolic

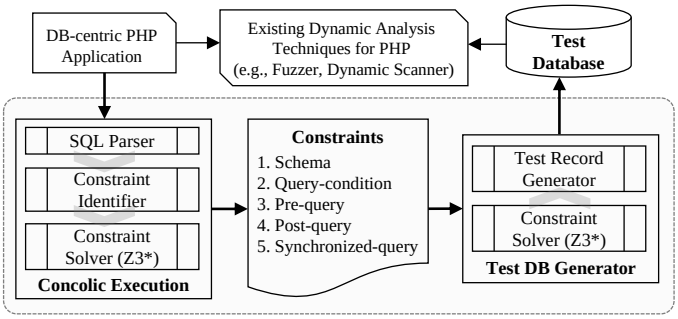


Fig. 3. Overview of SYNTHDB (*Z3 solver [34]).

execution engine is based on Vulcan Logic Dumper [35], which is an extension of Zend Engine [36]. 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 [34] 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* 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.

Algorithm. Due to the space, we provide an example of how SYNTHDB handles non-trivial path constraints and a complete algorithm of concolic execution in Appendix VII-A and VII-D.

B. 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

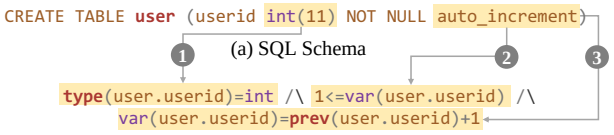
queries and program code by leveraging our concolic execution engine. Next, we explain each constraint with examples.

1) 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 JSQLParser [37] 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).



(a) SQL Schema

(b) Extracted Constraints

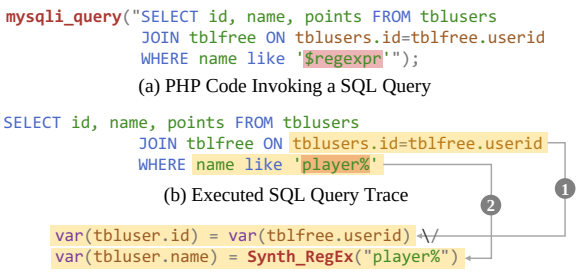
Fig. 4. Extracting Schema Constraints from Database Schema.

Example. Figure 4-(a)'s schema provides three constraints in Figure 4-(b). (1) From the `INT` type, we obtain the constraint that the field's type is an integer. (2) '`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.

2) 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 5-(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 5-(b) implies it is a regular expression. We model the value patterns and contexts to extract query-condition constraints.



(a) PHP Code Invoking a SQL Query

(b) Executed SQL Query Trace

(c) Extracted Constraints

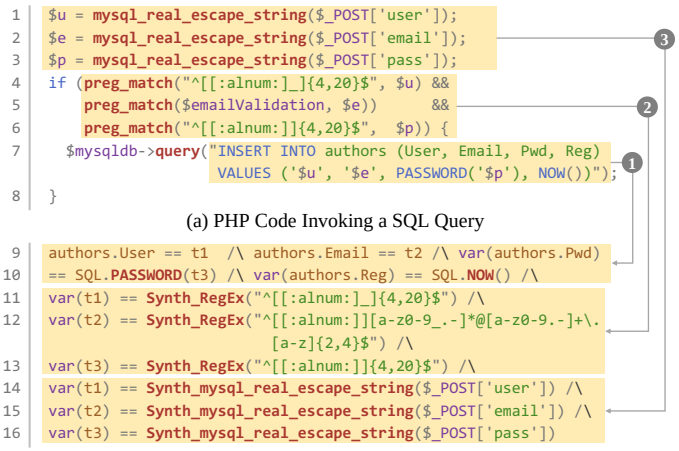
Fig. 5. Extracting Query-condition Constraints.

Example. Figure 5 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 5-(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 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).

3) 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 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 6-(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 6-(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



(a) PHP Code Invoking a SQL Query

(b) Pre-query Constraints

Fig. 6. Extracting Pre-query Constraints.

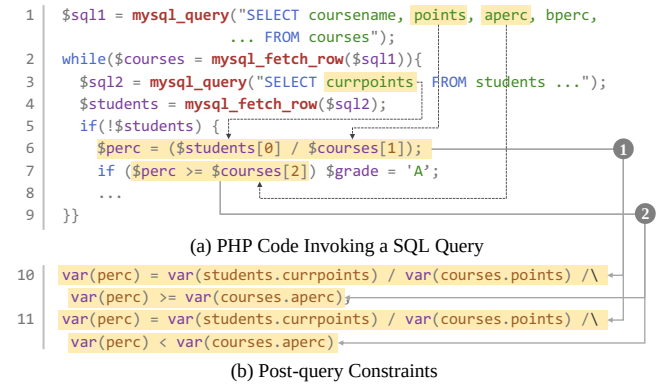
`t1~3` for program variables used in the query, `$u`, `$e`, and `$p`, respectively. SQL built-in functions are handled by defining our own functions that emulate the original functions (e.g., `SQL.PASSWORD()` and `SQL.NOW()` to generate a hashed password and return the current time, respectively). Observe 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.

4) 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[3]. 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).

Example. Figure 7-(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 7-(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

[3]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.

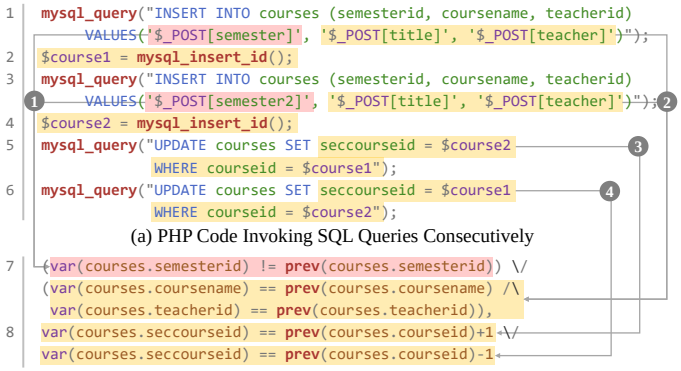


(a) PHP Code Invoking a SQL Query

(b) Post-query Constraints

Fig. 7. Extracting Post-query Constraints.

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.



(a) PHP Code Invoking SQL Queries Consecutively

(b) Synchronized-Query Constraints

Fig. 8. Extracting Synchronized-query Constraints.

5) 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 ($id, ...);`
2. `INSERT into tableB (y, ...) VALUES ($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 [38] 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.

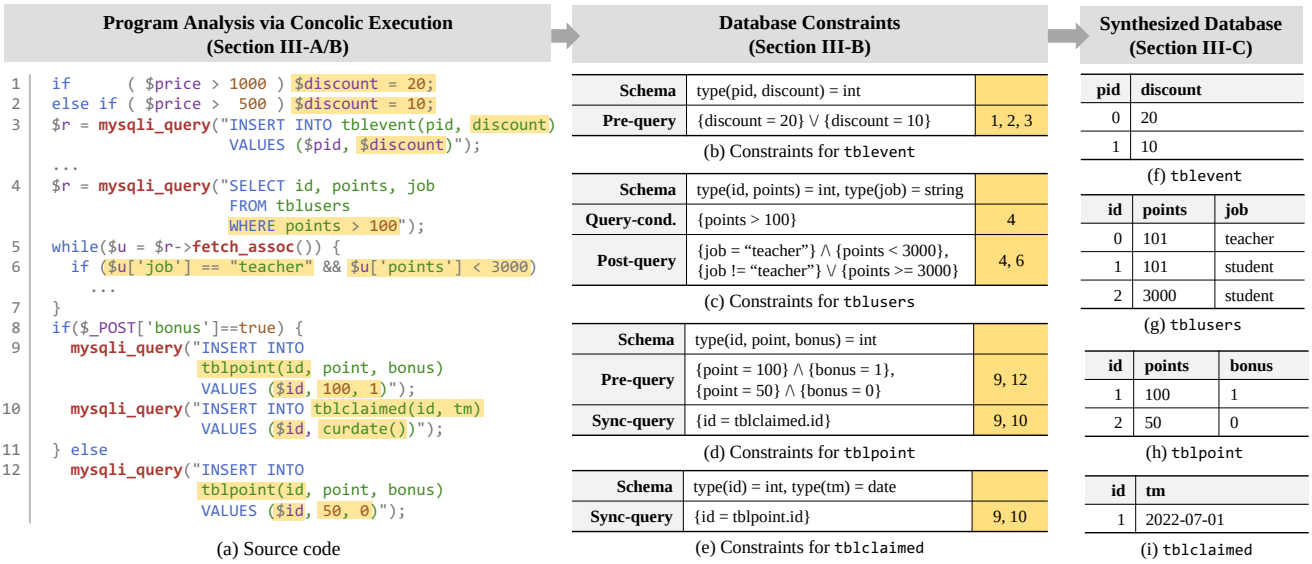


Fig. 9. Overall Procedure of Synthesizing Database (Highlighted columns in (b)~(e) present the source lines where we extracted constraints from).

Example. Figure 8-(a) shows a program executes four SQL queries consecutively. While the two consecutive queries have different values for `courses.semesterid` (1), they share variables for the next two fields: `courses.coursename` and `courses.teacherid` (2). 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).

C. Synthesizing Database

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

1) Overall Procedure: Figure 9 shows an end-to-end example from the source code to the synthesized database. Specifically, Figure 9-(a) shows the target PHP program, and our concolic execution engine identifies database constraints from the highlighted parts of the code and queries. Figure 9-(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 9-(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 9-(b). Finally, SYNTHDB generates database records that satisfy the constraints as shown in Figure 9-(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 9-(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 9-(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 9-(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 9-(e)). Lastly, we generate records satisfy the constraints in Figure 9-(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 [18], [25].

2) 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 10, 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 Appendix VII-F due to the space limit.

```
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 }
```

Fig. 10. 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 paper, 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.

IV. 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 [18], Domino [25], and Datafaker [39]. We then execute a dynamic analysis technique for PHP on top of each generated database and compare the observed code and query coverage (Section IV-A). We also conduct three types of security analysis to measure how the test databases affect security testing, including the vulnerability detection testing with an active vulnerability scanner, Burp Suite [19] (Section IV-B1), the reachability test against reported vulnerabilities (Section IV-B2), and integrating SynthDB with two fuzz testing tools, WFuzz [20] and webFuzz [21] (Section IV-B3).

PHP Applications for Evaluation. As presented in Table I, 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 (Details in Section VII-C).

1. We choose twelve applications (s1~s8, s12, and s15~s17) out of 28 applications that are frequently evaluated by previous work [5], [56]–[58]. 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 [59] PHP project for each category. We select WordPress [50] and OpenCart [48] for the *CMS* and *eCommerce platform* categories respectively. For the *online forum* category, we select two applications, phpBB [46] and SMF [51], 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 [25], Datafaker [39], and EvoSQL [18]. Table II summarizes the advantages and limitations of them, focusing on which database constraints are supported. First, Domino [25] and Datafaker [39] 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 [18] 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 (Section III-C1).

– *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 [39], [60] 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

TABLE I. 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 [24]	63	1,587	15	95	17	32	214	263	School management system
s2	PHP7-Webchess [40]	29	1,505	7	48	14	20	60	94	Web game
s3	Timeclock-1.04 [41]	63	10,820	8	35	18	19	262	299	Employment management system
s4	Mybloggie-2.1.4 [42]	59	3,053	4	24	5	5	74	84	Content management system
s5	Faqforge-1.3.2 [43]	15	302	2	11	3	5	22	30	Online forum
s6	Wackopicko-1.0 [44]	49	720	13	60	13	3	24	40	Photo management system
s7	phpBB-2.0.23 [45]	74	10,798	30	277	44	89	244	377	Online forum
s8	phpBB-3.3.8 [46]	1,091	40,612	69	601	64	341	938	1,343	
s9	OpenCart-3.0.3.8 [47]	1,932	60,515	136	834	246	111	586	943	Ecommerce platform
s10	OpenCart-4.0.0 [48]	2,866	49,018	142	871	258	118	623	999	
s11	WordPress-5.1.2 [49]	901	84,891	12	94	12	32	271	315	Content management system
s12	WordPress-6.0.1 [50]	1,332	110,227	12	94	12	31	264	307	
s13	SMF-2.1.2 [51]	316	45,641	73	525	7	270	929	1,206	Online forum
s14	OsCommerce-2.4.0 [52]	422	15,809	49	343	529	10	377	916	Ecommerce platform
s15	CEPhoenix-1.0.7 [53]	1361	23,938	55	369	149	101	436	686	Ecommerce platform
s16	ZenCart-1.5.7 [54]	1,829	74,960	103	848	394	215	1,311	1,920	Ecommerce platform
s17	Drupal-9.0.0 [55]	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	

TABLE II. 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.

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 (Section III-A) for all the existing techniques. In other words, all the experiments in Section IV-A, Section IV-C, and Section IV-B2 are conducted on top of our concolic execution engine with test databases generated by each technique.

A. 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 [61] to measure the code coverage. Figure 11(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 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 11(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

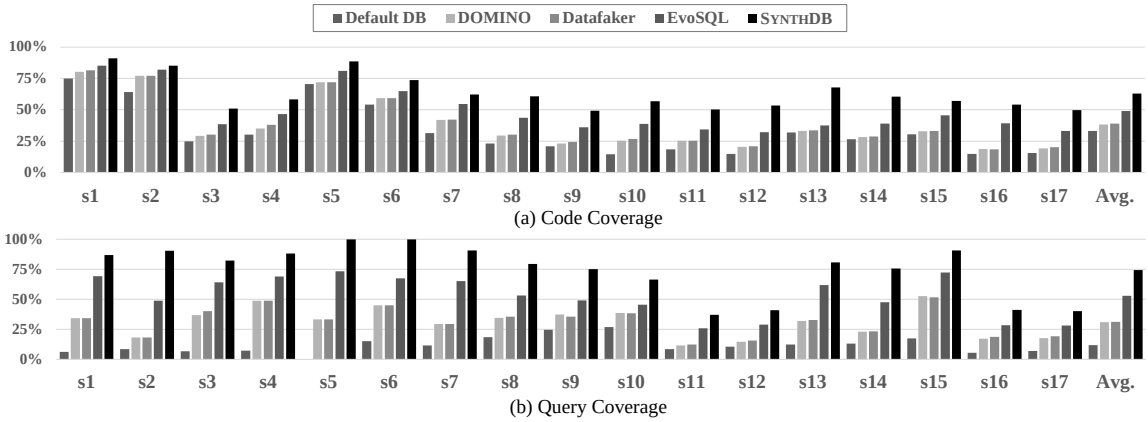


Fig. 11. Code and Query Coverage of SYNTHDB ('Default DB' presents an execution with a DB right after the installation. s1~s17 are the ids from Table I).

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. Due to page limit, we present the number of each constraints SYNTHDB derived in Appendix VII-B.

B. 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.

1) 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 I with test databases generated by SYNTHDB, EvoSQL, DOMINO, and Datafaker.

Vulnerability Report Collection. First, we collect vulnerability reports for PHP applications listed in Table I, from the CVE database [62], Exploit Database [63], previous research [64]–[66], and security reports by application developers[4]. We manually verify each reported vulnerability to prune out false positives and uncertain reports that do not provide sufficient details for the vulnerable code's location. 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 [67] 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 III 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 III does not include results for 5 applications (i.e., s8, s10, s13, s15, and s16) because they do not have any known vulnerabilities, and 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 [68] (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.

2) 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

[4]From public repositories such as GitHub.

TABLE III. 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.

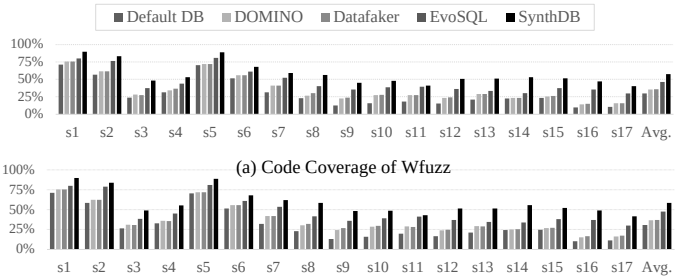
TABLE IV. REACHABILITY TEST AGAINST PHP VULNERABILITIES.

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.

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 IV shows a result. SYNTHDB can successfully reach 80.9% (171 out of 189) vulnerable statements. EvoSQL can cover 55.3% (128 out of 189), 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. We further investigate the vulnerabilities that SYNTHDB failed to reach and discuss our findings in Appendix VII-E.

3) Integrating with Fuzzing Methods: We use two popular fuzzing tools, Wfuzz [20] and webFuzz [21], that are designed for testing web applications. Figure 14 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



(a) Code Coverage of Wfuzz

(b) Code Coverage of webFuzz

Fig. 12. Code Coverage Results by Existing Fuzzing Tools.

TABLE V. 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.

the executions with a default DB.

C. Runtime Performance Measurement

We measure the time taken for generating a test database by each technique Table V 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 [18] 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.[5] Datafaker [39] and DOMINO [25] 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 V 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

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

script, 6.5% for parsing the trace files, 25.8% for database generation and 12.2% for other components.

V. Related Work

Test Data Generation. Emmi et al. [14] 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 [18] 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 [15], [16], [69] have been proposed to generate test data to cover various SQL queries. DOMINO [25] is an automated test data generation technique that aims to systematically exercise the integrity constraints in database schemas. There exist prior works [17], [70], [71] studying test data generation techniques for exercising and evaluating database integrity constraints.

Recently, JaSoN [72] 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 [73] is a database test suite reduction technique by merging similar test cases.

Static Analysis for Web Applications. Static-based security analysis [9], [11], [74], [75] and vulnerability scanner [10], [76]–[82] are popular approaches for identifying security issues of web applications. However, web applications are typically written in dynamic languages such as PHP, and most of them are frequently interact with external resources, such as a database, to store and retrieve data effectively. Static-based techniques have difficulties analyzing dynamic code and interaction with databases.

Dynamic Analysis for Web Applications. There exist approaches to provide effective dynamic analysis frameworks or testing environments for web applications [13], [26]–[32]. Dynamic vulnerability testing techniques for web applications [27], [83]–[87] execute the target application on top of dynamic analysis frameworks to identify vulnerable code or malicious logic. Hybrid approaches [88]–[90] combine static and dynamic techniques. Existing dynamic and hybrid approaches do not consider database-backed applications, or assume that the user provides a proper database.

VI. Discussion

Other Languages and DBMS. The current version of SYNTHDB only supports PHP and MySQL database. SYNTHDB uses JSQLParser [37] 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 Section IV-A, 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.

VII. Conclusion

We present SYNTHDB, a system that synthesizes a database for dynamic security analysis of database-backed PHP web applications. It leverages a concolic execution to identify interactions between PHP codebase and the SQL queries, 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 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 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 vulnerabilities from 5 real-world applications.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive feedback. The authors gratefully acknowledge the support of NSF 1916499, 1916500, 1908021, 1909856, 1850392, and 2145616. This research was also partially supported by a gift from Cisco Systems. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

REFERENCES

[1] Verizon, "Data breach investigations report," https://enterprise.verizon.com/resources/reports/2021-data-breach-investigations-report.pdfx.

[2] Sucuri, "Website threat research report," https://sucuri.net/wp-content/uploads/2020/01/20-sucuri-2019-hacked-report-1.pdf, 2019.

[3] D. Canali and D. 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.

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

[5] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "Navex: Precise and scalable exploit generation for dynamic web applications," in *27th USENIX Security Symposium*, 2018, pp. 377–392.

[6] B. Anderson and D. 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.

[7] K. Borgolte, C. Kruegel, and G. Vigna, "Delta: automatic identification of unknown web-based infection campaigns," in *Proceedings of the ACM CCS'13*, pp. 109–120.

[8] L. Invernizzi, P. M. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna, "Evilseed: A guided approach to finding malicious web pages," in *2012 IEEE symposium on Security and Privacy*, 2012.

[9] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," in *European Symposium on Research in Computer Security*. Springer, 2008.

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

[11] J. Dahse and T. Holz, "Simulation of built-in php features for precise static code analysis." in *NDSS*, vol. 14. Citeseer, 2014, pp. 23–26.

[12] T. P. Group, "Dphp runkit book," http://php.net/manual/en/book.runkit.php, 2016.

[13] P. M. Wrench and B. V. Irwin, "Towards a sandbox for the deobfuscation and dissection of php malware," in *2014 Information Security for South Africa*. IEEE, 2014, pp. 1–8.

[14] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *ISSTA '07*, 2007.

[15] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid, "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.

[16] M. J. Suárez-Cabal, C. de la Riva, J. Tuya, and R. Blanco, "Incremental test data generation for database queries," *Automated Software Engineering*, vol. 24, no. 4, pp. 719–755, 2017.

[17] J. Zhang, C. 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.

[18] J. Castelein, M. Aniche, M. Soltani, A. Panichella, and A. van Deursen, "Search-based test data generation for sql queries," in *Proceedings of the 40th international conference on software engineering*, 2018.

[19] "Burp suite," 2020, https://portswigger.net/burp.

[20] "Wfuzz – The Web Fuzzer," 2020, https://github.com/xmendez/wfuzz.

[21] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "Webfuzz: Grey-box fuzzing for web applications," in *Computer Security – ESORICS 2021*. Springer-Verlag, 2021.

[22] G. A. D. Lucca and A. R. Fasolino, "Testing web-based applications: The state of the art and future trends," *Inf. Softw. Technol.*, 2006.

[23] Y.-F. Li, P. K. Das, and D. L. Dowe, "Two decades of web application testing - a survey of recent advances," *Inf. Syst.*, vol. 43, pp. 20–54.

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

[25] A. Alsharif, G. M. Kapfhammer, and P. McMinn, "Domino: Fast and effective test data generation for relational database schemas," *2018 IEEE ICST*, 2018.

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

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

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

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

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

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

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

[33] Oracle, "Data integrity," https://docs.oracle.com/cd/B19306_01/server.102/b14220/data_int.htm.

[34] "Z3," 2022, https://github.com/Z3Prover/z3.

[35] "Vulcan logic dumper," https://derickrethans.nl/projects.html, 2016.

[36] "The PHP Interpreter," 2021, https://github.com/php/php-src.

[37] "JSqlParser," 2021, https://github.com/JSQLParser/JSqlParser.

[38] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, p. 121–141, jan 1979.

[39] "Datafaker-Tool for faking data," https://github.com/gangly/datafaker.

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

[41] "Timeclock," https://sourceforge.net/projects/timeclock/files/PHP%20Timeclock/.

[42] "myBloggie," https://sourceforge.net/projects/mybloggie/files/mybloggie/.

[43] "FaqForge," https://sourceforge.net/projects/faqforge/files/faqforge/.

[44] "WackoPicko Vulnerable Website," 2018, https://github.com/adamdoupe/WackoPicko.

[45] "phpBB 2.0.23," http://www.oldversion.com/windows/phpbb-2-0-23/.

[46] "phpBB 3.3.8," https://www.phpbb.com/.

[47] "OpenCart 3.0.3.8," https://github.com/opencart/opencart/releases/tag/3.0.3.8/.

[48] "OpenCart 4.0.0," https://github.com/opencart/opencart/releases/tag/4.0.0.0/.

[49] "WordPress 5.1.2," https://github.com/WordPress/WordPress/releases/tag/5.1.2/.

[50] "WordPress 6.0.1," https://github.com/WordPress/WordPress/releases/tag/6.0.1/.

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

[52] "OsCommerce240," https://github.com/osCommerce/oscommerce2.

[53] "Ce-phoenix," https://github.com/gburton/CE-Phoenix/tree/1.0.5.0.

[54] "Zencart 1.5.7," https://github.com/zencart/zencart/tree/v155.

[55] "Drupal 9.0.0," https://www.drupal.org/project/drupal/releases/9.0.0.

[56] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "webfuzz: Grey-box fuzzing for web applications," in *ESORICS*, 2021.

[57] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Chainsaw: Chained automated workflow-based exploit generation," in *Proceedings of the ACM CCS'16*, pp. 641–652.

[58] Y. Zou, Z. Chen, Y. Zheng, X. Zhang, and Z. Gao, "Virtual dom coverage for effective testing of dynamic web applications," in *Proceedings of ISSTA'14*, 2014, p. 60–70.

[59] "builtwith," 2021, https://builtwith.com/.

[60] "SchemaAnalyst," https://github.com/schemaanalyst/schemaanalyst.

[61] "Xdebug ," 2021, https://xdebug.org/.

[62] "Common Vulnerabilities and Exposures," 2021, https://cve.mitre.org/.

[63] "Exploit Database," 2021, https://www.exploit-db.com/.

[64] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," *IEEE 31st International Conference on Software Engineering*, pp. 199–209, 2009.

[65] "Security Testing Report," 2020, https://github.com/carloFanc/Security-Testing/blob/main/FinalReportCarloFanciulli.pdf.

[66] "Security Testing Project," 2017, https://github.com/davidepedranz/security_testing_project/blob/master/report/vulnerabilities.pdf.

[67] PortSwigger, "Burp intruder," https://portswigger.net/burp/documentation/desktop/tools/intruder.

[68] "Issue definitions - burp suite," https://portswigger.net/kb/issues.

[69] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "Qagen: Generating query-aware test databases," in *Proceedings of the 2007 ACM SIGMOD*, New York, NY, USA, 2007.

[70] P. McMinn, C. J. Wright, and G. M. Kapfhammer, "The effectiveness of test coverage criteria for relational database schema integrity constraints," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, pp. 1 – 49, 2015.

[71] P. McMinn, C. J. Wright, C. Kinneer, C. J. McCurdy, M. Camara, and G. M. Kapfhammer, "Schemaanalyst: Search-based test data generation for relational database schemas," *IEEE International Conference on Software Maintenance and Evolution*, pp. 586–590, 2016.

[72] H. Winkelmann and H. Kuchen, "Symbolic Execution of NoSQL Applications Using Versioned Schemas," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, ser. SAC '21, New York, NY, USA, 2021, p. 1778–1787.

[73] A. Alsharif, G. M. Kapfhammer, and P. McMinn, "Sticcer: Fast and effective database test suite reduction through merging of similar test cases," *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 220–230, 2020.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[88] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," *2008 IEEE Symposium on Security and Privacy (S&P'08)*, pp. 387–401, 2008.

[89] R. Jahanshahi, A. Doup'e, and M. Egele, "You shall not pass: Mitigating sql injection attacks on legacy web applications," *Proceedings of the ACM ASIACCS*, 2020.

[90] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *USENIX Security*, 2005.

[91] V. Garousi, R. Özkan, and A. Betin-Can, "Multi-objective regression test selection in practice: An empirical study in the defense software industry," *Information and Software Technology*, vol. 103, 2018.

[92] A. Arrieta, P. Valle, J. A. Agirre, and G. Sagardui, "Some seeds are strong: Seeding strategies for search-based test case selection," *ACM Transactions on Software Engineering and Methodology*, 2022.

[93] "Advanced PHP 7 eCommerce Website," https://github.com/justinhartman/complete-php7-ecom-website.

[94] "Online shopping system advanced," https://github.com/PuneethReddyHC/online-shopping-system-advanced.

[95] "Doctor-Appointment," https://github.com/divScorp/Doctor-Appointment.

[96] "Hostel Management System," https://github.com/Bharat-Reddy/Hostel-Management-System.

[97] "Inventory management system," https://github.com/carloFanc/Security-Testing/tree/main/inventory-management-system-fixed.

[98] "Andy's PHP Knowledgebase," https://sourceforge.net/projects/aphpkb/files/.

[99] "MediaWiki," https://www.mediawiki.org/wiki/MediaWiki.

[100] "Better Search," https://wordpress.org/plugins/better-search/.

[101] "Contact Form 7 Database Addon – CFDB7," https://wordpress.org/plugins/contact-form-cfdb7/.

[102] "Student Result," https://wordpress.org/plugins/simple-student-result/.

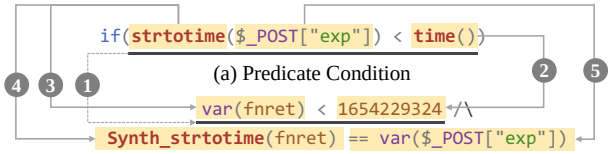
[103] "Contact Forms Lite," https://wordpress.org/plugins/wpforms-lite/.

APPENDIX

A. 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 13-(a) shows an example predicate where SYNTHDB extracts the constraints shown in Figure 13-(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).



(a) Predicate Condition

(b) Extracted Path Constraints

Fig. 13. Extracting Path Constraints

B. Observation from Coverage Evaluations

From the coverage evaluation, we observe that the most common cases that SYNTHDB can cover but others cannot are predicates that evaluate values from the query return, including logical comparisons between two or more values from different columns. Let's revisit the example code presented in Figure 7 (a). As we discussed, the query return values from two seperate queries at line 1 and 3 are stored in `$sql1` and `$sql2`, and the program's path changes based on the values from three columns, `points`, `aperc`, and `students`. To explore the true branch at line 7, we need to understand the relationships between those two columns and generate items accordingly. We observe that *query-condition* and *post-query* constraints play significant roles in improving the code coverage and Table VI shows the number of each constraint SYNTHDB derived from each PHP application we evaluate.

C. SQL keyword Statistics

We run statistical analysis on 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.

Results and Observations. First, there are 744 SQL keywords according to the MySQL version 8.0.24's specification. Among them, 214 keywords are appeared in our selected 17 applications (Table I), meaning that the selected applications include diverse SQL queries (hence those applications are of high quality for the evaluation). Second, a total of 49,585 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.

TABLE VI. QUERY-CONDITION (QC^1) AND POST-QUERY (PQ^2) CONSTRAINTS SYNTHDB DERIVED FROM PHP APPLICATIONS.

Id	QC^1	PQ^2	Total	Id	QC^1	PQ^2	Total
s1	59	126	185	s10	481	379	860
s2	30	27	57	s11	247	179	426
s3	142	54	196	s12	262	203	465
s4	34	37	71	s13	752	518	1,270
s5	12	11	23	s14	214	198	412
s6	21	12	33	s15	428	227	655
s7	284	191	475	s16	805	614	1,419
s8	628	418	1,046	s17	215	198	413
s9	557	317	874	**Total**	5,171	3,709	8,880

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 VII, 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.

TABLE VII. 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

D. Algorithm: Concolic Execution

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[6], 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.

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).

E. Reachability Test: Failed Cases of SYNTHDB

We further investigated the vulnerabilities that SYNTHDB failed to reach in the reachability evaluation (Section IV-B2). Among the 18 cases, six of them require other external resources. One requires different configurations (e.g., changing the application language or turning on the legacy features), five cases failed due to SMT solver's limited URL support,

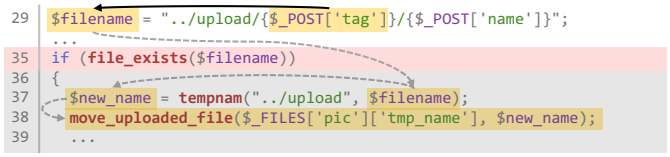
[6]We consider the following opcodes as branch instructions: IS_IDENTICAL, IS_NOT_IDENTICAL, IS_EQUAL, IS_NOT_EQUAL, IS_SMALLER, IS_SMALLER_OR_EQUAL, SWITCH_LONG, SWITCH_STRING, ISSET_ISEMPTY_VAR, ISSET_ISEMPTY_DIM_OBJ, ISSET_ISEMPTY_PROP_OBJ, ISSET_ISEMPTY_CV, ISSET_ISEMPTY_THIS, and ISSET_ISEMPTY_STATIC_PROP.

Algorithm 1: Obtaining Database Constraints

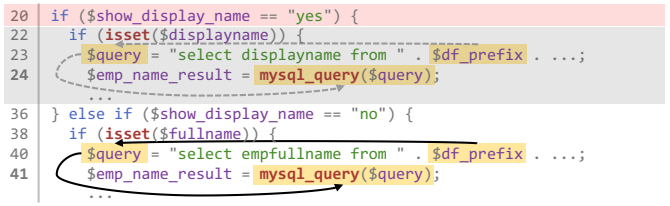
```
   Input : UNTRUSTEDSRC: a list of five untrusted sources ($_GET,
           $_POST, $_REQUEST, $_SESSION, and $_COOKIE).
   Output: Collected four types (pre-query, query-condition,
           post-query, and synchronized-query) of constraints.
 1 procedure OBTAINDATABASECONTRAINTS
 2   TAINT( UNTRUSTEDSRC )
 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       query ← GETFUNCARGS( ins )
 9       if ISTAINTED( query ) and
10         QUERYTYPE( query ) = (INSERT or UPDATE) then
11         tainted ← GETTAINTEDVARS( query )
12         Constraints_pre-query ← 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       Constraints_post-query ← 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       query ← GETFUNCARGS( ins )
26       if QUERYTYPE( query ) = (SELECT or UPDATE) and
27         GETCONDITIONCLAUSE( query ) ≠ ∅ then
28         Constraints_query-cond ← 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       query ← GETFUNCARGS( ins )
35       Consecutive-queries ← Consecutive-queries ∪ <query>
36     if ISDBFUNCTIONCALL( ins ) then
37       query ← GETFUNCARGS( ins )
38       if ISTAINTED( query ) and
39         QUERYTYPE( query ) = (INSERT or UPDATE) then
40         tainted ← GETTAINTEDVARS( query )
41         for ∀ query_consec ∈ Consecutive-queries do
42           tainted_consec ← GETTAINTEDVARS( query_consec )
43           shared ← tainted ∩ tainted_consec
44           if shared ≠ ∅ then
45             Constraints_sync-query ← Constraints_sync-query
46                                      ∪ <query, query_consec, shared>
47     return <Constraints_pre-query, Constraints_query-cond,
48             Constraints_post-query, Constraints_sync-query>
```

two cases require serialized object as input, and two case requires a specific plug-in installed. Furthermore, we analyze the remaining two cases that SYNTHDB failed to reach.

1. *Case 1:* Figure 14-(a) presents the PHP code from WackoPicko [44] that contains a directory traversal vulnerability. The if-statement at line 35 evaluates the existence of a specific file where the file name is provided by the user through `$_POST`. SYNTHDB failed to take the `true` branch because the system does not have the requested file in



(a) Case 1: **WackoPicko-upload.php** (Directory traversal vulnerability at 38)



(b) Case 2: **timeclock-1.04-leftmain.php** (SQL injection at 24 and 41)

Fig. 14. Analysis of Failed Cases of SYNTHDB (Gray shaded regions are not reached as SYNTHDB-synthesized database failed to provide data that satisfy the red-shaded predicates' conditions at lines 35 and 20).

it. SYNTHDB focuses on generating a test database for web application testing, but reconstructing other external resources, such as a file, is out of the scope of this work.

2. *Case 2:* Figure 14-(b) shows code snippets from timeclock-1.04 [41]. `$show_display_name` variable at lines 20 and 36 is defined by a dynamically generated configuration file. Our concolic execution only takes the `else` branch at line 36 (not the `true` branch at lines 22~24) because the default value of `$show_display_name` is "no". The current implementation of SYNTHDB does not control the values defined in the configuration file.

We leave handling the above cases as future work.

F. Additional Discussion

Complexity of Regular Expression Constraints. We observed the average length and processing time of regular expression is 11 (in # of characters) and 51 ms for each regular expression. Due to the limitation of the library exrex that SYNTHDB uses, 3.27% of regular expressions were not solved. For instance, "/^[^<]*+(?:<[^>]*+>[^<]*+)*+$/" raises a "multiple repeat error."

Clarification of SYNTHDB's Results and Other Tools' Results. SynthDB's result is almost a superset except for an average of 0.32% total code (0%,0.61%,0.65%,0% for default DB, Domino, Datafaker, and EvoSQL respectively). The main reason for the missing 0.32% is conflicting constraints, as we discussed in Section III-C "Conflicting Constraints" single database cannot satisfy multiple database constraints that have conflicting definitions. SynthDB chooses a database with the least number of conflicting constraints as output, and thus causes missing code and queries. Supporting multiple databases to handle conflicting constraints is our future work.

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 Section III-C, 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 [91], [92] 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.

Generality of Database Constraints. We try our best to be generic in deriving constraints from our observation. Specifically, we study 28 common and popular web applications' codebases and databases (we manually operate the applications to obtain the databases), to derive the constraints. The 28 applications are (1) 17 evaluated applications, (2) 7 applications (Ecom-site [93], Onlineshop [94], Doctor-Appointment [95], Hotel-Management-System [96], Inventory [97], Aphpkb [98], and mediawiki [99]), and (3) 4 wordpress plug-ins. (better-search [100], cfdb7 [101], student-result [102], and wp-forms [103]) For the concolic execution, we develop a standard concolic execution engine as described in Section III-A. We additionally implemented support for symbolizing queries, query return values, and database fields.

False Positives and Negatives. We identify three main sources for FNs by manually inspecting PHP code, SQL queries, and generated test databases: Conflicting constraints are the main source of FNs (Section III-C "Conflicting Constraints"). The second source is unsupported SQL keywords (Section VII-C "SQL keyword Statistics"). The third source is dynamic schema changes (Section VI "Dynamic Schema Changes"). From our observation, 13.5% of uncovered code and 15.2% of uncovered queries are caused by the first two main sources. We start to evaluate the web applications after all installations/upgrading is done, so the current evaluation is unaffected by the third source. We plan to support dynamic schema changing during the evaluation in the future.