SQLPsdem: A Proxy-Based Mechanism Towards Detecting, Locating and Preventing Second-Order SQL Injections

Bing Zhang, Rong Ren, Jia Liu, Mingcai Jiang, Jiadong Ren, and Jingyue Li, *Senior Member, IEEE*

***Abstract*—Due to well-hidden and stage-triggered properties of second-order SQL injections in web applications, current approaches are ineffective in addressing them and still report high false negatives and false positives. To reduce false results, we propose a Proxy-based static analysis and dynamic execution mechanism towards detecting, locating and preventing second-order SQL injections (SQLPsdem). The static analysis first locates SQL statements in web applications and identifies all data sources and injection points (e.g., Post, Sessions, Database, File names) that injection attacks can exploit. After that, we reconstruct the SQL statements and use attack engines to jointly generate attacks to cover all the state-of-the-art attack patterns so as to exploit these applications. We then use proxy-based dynamic execution to capture the data transmitted between web applications and their databases. The data are the reconstructed SQL statements with variable values from the attack payloads. If a web application is vulnerable, the data will contain malicious attacks on the database. We match the data with rules formulated by attack patterns to detect first and second-order SQL injection vulnerabilities in web applications, particularly the second-order ones. We use a representative and complete coverage of attack patterns and precise matching rules to reduce false results. By escaping and truncating malicious payloads in the data transmitted from the web application to the database, we can eliminate the possible negative impact of the data on the database. In the evaluation, by generating 52,771 SQL injection attacks using four attack generators, SQLPsdem successfully detects 26 second-order (including 13 newly discovered ones) and 375 first-order SQL injection vulnerabilities in 12 open-source web applications. SQLPsdem can also 100% eliminate the malicious impact of the data with negligible overhead.**

***Index Terms*—Second-order SQL injection, static analysis, dynamic execution, proxy, detection and prevention.**

Manuscript received 19 September 2023; revised 23 March 2024; accepted 9 May 2024. Date of publication 14 May 2024; date of current version 18 July 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62376240, in part by the S&T Program of Hebei under Grant 226Z0701G, Grant 236Z0702G, and Grant 236Z0304G, in part by the Natural Science Foundation of Hebei Province under Grant F2022203026, in part by the Science and Technology Project of Hebei Education Department under Grant BJK2022029, and in part by the Innovation Capability Improvement Plan Project of Hebei Province under Grant 22567637H. Recommended for acceptance by W. G. J. Halfond. *(Corresponding author: Jiadong Ren.)*

Bing Zhang, Rong Ren, Jia Liu, Mingcai Jiang, and Jiadong Ren are with the School of Information Science and Engineering, Yanshan University, and the Key Laboratory for Software Engineering of Hebei Province, Qinhuangdao, Hebei 066000, P. R. China (e-mail: bingzhang@ysu.edu.cn; raniy@sina.cn; liujia_ysu@163.com; m991027961@126.com; jdren@ysu.edu.cn).

Jingyue Li is with the Department of Computer Science, Norwegian University of Science and Technology, Trondheim 7030, Norway (e-mail: jingyue.li@ntnu.no).

Digital Object Identifier 10.1109/TSE.2024.3400404

I. Introduction

ACCORDING to the Common Weakness Enumeration (CWE) and the Open Web Application Security Project 2021 (OWASP), structured query language (SQL) injection vulnerabilities remain a serious threat to web application security [1], [2], [3]. Attackers can steal data from the database, delete information and gain access to the server via SQL injection vulnerabilities.

Based on attack injection points and steps, SQL injections can be classified into first and second-order injections. The concept of second-order SQL injection [4] was introduced in 2004. In second-order SQL injections, the attacker first stores the attack vector in persistent data stores (PDS), such as a database, file system, or session, and then triggers the stored attack vector in a follow-up attack. Since there are many variations of stage-triggered and well-hidden second-order SQL injection, its detection is far more challenging than that of first-order SQL injection. It is, therefore, critical to effectively protect against second-order SQL injections. There is much research on first-order injections, e.g., [5], [6], [7], [8], [9], [10], [11]. However, fewer detection and prevention methods are proposed to address second-order SQL injections, and existing ones suffer from high rates of false negatives and false positives. The approaches proposed in [12], [13], [14], [15] fail to cover critical injection points (SIPs), such as sessions, cookies, filenames, and variables generated by the database or server program. Consequently, they may overlook second-order SQL injections related to these SIPs. In addition, these studies lack sufficient details in explaining how their test cases are generated, making the representativeness and completeness of selected attack cases questionable. These approaches also require storing malicious inputs in the database to detect second-order SQL injection vulnerabilities, which will cause security risks to the data in the existing database. Other approaches proposed in [16], [17], [18], [19] miss detecting certain types of injection instances. Limitations arise from their sole reliance on structure and syntax matching [16], [17], [18] or taint analysis-based sanitization

[19]. For instance, Tian et al. [16] cannot detect attacks that do not modify the SQL query structure, Chen [17] cannot detect malicious user inputs lacking SQL keywords, Medeiros et al. [18] cannot detect attacks resulting in the identical number of nodes and their order in Query Model and Query Structure, Dahse et al. [19] cannot detect the path-sensitive sanitization of data written to the database. The latest approach [20] detected second-order vulnerabilities based on potential paths through fuzz testing. It has a low efficiency and coverage to identify and trigger potential vulnerability paths, and the details of the fuzz testing are not provided as well.

To address challenges and limitations of state-of-the-art approaches, we design a proxy-based static analysis and dynamic execution mechanism (SQLPsdem). To identify more comprehensive *SIPs* than existing studies, e.g., [12], [13], [14], [15], we apply holistic static analysis on the source codes of web applications to cover all *SIPs*. To identify all potential SQL injection attacks, including first and second-order ones, we apply a proxy-based mechanism, which intercepts and analyzes all SQL queries or commands transmitted from the business logic to the database without needing to store the possible malicious payloads to SIPs in the database. Different from existing studies, such as [16], [17], [18], [19], [20], which rely on fixed and unsystematic attack generation approaches, we generate representative and complete malicious payloads to these SIPs by joining the force of popular attack engines, to cover seven well-established attack patterns (tautologies, union query, piggy-backed queries, inference, alternate encodings, illegal/logically incorrect queries and stored procedures) [21], [22], [23], [24], [25], [26], [27], [28]. SOLPsdem is designed to be flexible so that we can independently plug in and out attack generators to update their latest versions or add new ones. Our detection mechanism compares SQL queries or commands transmitted between web applications and their database, which are their responses to generated attacks, against rules formulated by attack patterns. To minimize potential false detection results arising from SQL tokens being indistinguishable in the source code and those from other resources (such as user inputs) [5], SQLPsdem annotates SQL tokens within the source code using tags. This guarantees consistent differentiation between SQL tokens originating from the codebase and other sources. Besides detection, SOLPsdem can also locate vulnerabilities in web applications by combining the information from static analysis and defend against vulnerability exploitation through character escaping and truncation.

The evaluation results of SQLPsdem show that it can accurately detect, locate, and prevent second-order SQL injections. SQLPsdem can also detect first-order SQL injections because many SIPs identified in the static analysis are relevant to first-order injections. Overheads introduced in detecting and preventing the attacks are negligible.

The main contributions of this paper are as follows.

(1) A novel proxy-based dynamic execution mechanism is designed to detect SQL injection attacks right before attacks are sent to different databases to be executed, which can catch all known patterns of malicious first and second-order injection payloads to the database.

(2) A novel SQL injection vulnerability detection, localization, and prevention approach is proposed to take advantage of comprehensive static analysis and attack generation to minimize false negatives and rule-based attack matching to reduce false negatives and positives. The approach has a modular architecture that each of its modules, i.e., the static analysis method, attack engine, and attack matching rules, can be independently replaced and updated if new attack patterns emerge.

(3) Thirteen new second-order SQL injection vulnerabilities are identified, which can help improve the security of selected open-source web applications.

The rest of this paper is organized as follows. **Section II** describes the background and preliminaries related to second-order SQL injections. **Section III** introduces the design of SQLPsdem. **Section IV** presents the evaluation results of SQLPsdem. **Section V** compares SQLPsdem with related studies. **Section VI** discusses the work of this paper. **Section VII** concludes this paper and sheds light on future work.

II. Background and Preliminaries

Ray and Ligatti [29], [30] defined SQL injections as "code-injection attacks on outputs" (CIAO) and "inputs that 'broken' the NIE (Noncode Insertion or Expansion) property" (BroNIE). Injections that cause CIAOs must also cause BroNIEs.

Based on the attack intention, researchers have summarized SQL injection attack patterns into seven categories, namely, tautologies, union query, piggy-backed queries, inference, alternate encodings, illegal/logically incorrect queries and stored procedures [21], [22], [23], [24], [25], [26], [27], [28]. These seven attack patterns can lead to first and second-order SQL injections via injection points. This section first analyzes the principle of second-order SQL injection vulnerability, and then expounds on seven attack patterns and the mechanism of second-order SQL injections caused by them.

A. Second-Order SQL Injection Principle

A complete second-order SQL injection attack requires the browser or the client to send queries twice to the server. The first query stores attack strings without triggering any vulnerabilities in the web application. In the second query, the attacker uses the attack vector stored in the first query to change the syntax or semantics of the stored SQL statement to trigger vulnerabilities. **Fig. 1** shows the PHP code snippet for user registration and password change. This code snippet uses the function *mysqli_real_escape_string()* to escape special characters (e.g.,\x00, \n, \r, \, ', ", \x1a) in user inputs to ensure that they are secure and reliable. However, the attack string *admin'--* crafted by the attacker is escaped as *admin\'--* and can be successfully stored in the database. During the password change, the attacker can use the malicious string *admin'--* stored in the database to perform the attack. In the MySQL database, since the character "--" indicates a comment, the SQL statement containing such a malicious string is passed to the server for parsing and execution, which will eventually change the password of the administrator with the account "admin", instead of that of the normal user with the account *admin'--*.

```
Register:
1....
2.$username = $_GET["username"]; //User input --> $_GET["username"]: admin'--
3.$password = md5($_GET["password "]); //User input --> $_GET["password"]: 000
4.$username = mysqli_real_escape_string($conn,$username);
// $username: admin\'--   $password:c6f057b86584942e415435ffb1fa93d4
5.$sql = "insert into user (username, password) values('$username', '$password')";
6.mysqli_query($sql); //Insert the username and password into the database
7....
Update password:
1....
2.$sql = "select username from user where id = '$id' ";
3.$username = mysqli_fetch_assoc($sql); //Database values are read without any checks
4.$update = "UPDATE user SET password='newpassword' WHERE username='$username'";
5.          →UPDATE user SET password= '123456' WHERE username='admin'-- '; //SQL
executed by the database
6....
```

Fig. 1. Code fragments for user registration and update password.

TABLE I
TYPES OF FIRST-ORDER SQL INJECTIONS

TypeID	Injection Type
τ_1	Tautologies
τ_2	Union Queries
τ_3	Piggy-Backed Queries
τ_4	Inference
τ_5	Alternate Encodings
τ_6	Illegal/Logically Incorrect Queries
τ_7	Stored Procedures

B. Second-Order SQL Injection Attack Patterns

A total of seven SQL injections attack patterns are listed in **Table I**.

We use the initial SQL statement SQL_0 as shown below to explain seven attack patterns and how they lead to second-order injections.

SQL_0	SELECT * FROM student WHERE user = 'SQLi' AND pass = 'SQLi'

(1) **Tautologies**: the Boolean value of the conditional clause in the SQL statement is always true, which makes it possible to bypass the validation of the statement for the associated condition. In SQL_1, "*or 1=1#*" is added to make *where* clause always true.

SQL_1	SELECT * FROM student WHERE user = 'SQLi' **or 1=1#**' AND pass = 'SQLi'

(2) **Union Queries**: the syntax of original SQL statements is changed, or additional SQL statements via keyword *Union* are inserted, so that the results returned by these SQL statements contain other query results. The example is shown in SQL_2.

SQL_2	SELECT * FROM student WHERE user = 'SQLi' **Union Select * FROM tableName#**' AND pass = 'SQLi'

(3) **Piggy-Backed Queries**: similar to Union Queries, other commands (such as *Drop*, *ShutDown*, etc.) are used to make current SQL statements execute extra SQL statements during execution. This type of injection is achieved by requiring the back-end database to initiate the command to execute multiple SQL statements, as shown in SQL_3.

SQL_3	SELECT * FROM student WHERE user = 'SQLi' **; system shutdown#**' AND pass = 'SQLi'

(4) **Inference**: the attacker designs multiple attack strings and judges whether there are injection points on the web page according to web page response results. This type of injection includes time inference and blind injection on different conditions. Examples of SQL statements under such attacks are SQL_4 and SQL_5 statements, respectively.

Time inference	
SQL_4	SELECT * FROM student WHERE user = 'SQLi' and **if(conditions, SLEEP(5), SLEEP(0))#**' AND pass = 'SQLi'
Blind injection on different conditions	
SQL_5	A. SELECT * FROM student WHERE user = 'SQLi' **or 1=1#**' AND pass = 'SQLi' B. SELECT * FROM student WHERE user = 'SQLi' **or 1=0#**' AND pass = 'SQLi'

Regarding time inference, by entering a time inference command containing a conditional formula, such as the *SLEEP(5)* function in SQL_4 and by observing whether the web page has a response delay for 5 seconds, we can know whether the injection is successful. For blind injections on different conditions, by entering a series of different conditional equations (A and B sentences in SQL_5) with true or false states, the web page reacts differently to true and false conditions to allow attackers obtain sensitive database information. As the syntax of this type of attack is more complex and far from fixed, it is not easy to identify its specific form. However, both attacks introduce complex attack strings which usually contain sensitive characters like *'*, *"*, *--*, and *#*.

(5) **Alternate Encodings**: this type of attack is commonly used to bypass the filter set by the developer, and it converts characters which are the targets of the filter into an encoded form, such as *hexadecimal*, *ASCII* or *Unicode*, as shown in SQL_6 and SQL_7 statements, respectively.

SQL_6	SELECT * FROM student WHERE user = 'SQLi' **;exec(char(0X73687574646f776e))- -** ' AND pass = 'SQLi'
SQL_7	INSERT INTO student VALUES(1, CONCAT(CAHR(0X3C), **"script src= 'Malicious link'"**, CHAR(0X3E)))

As indicated by the SQL_6 statement, the attacker encodes the *SHUTDOWN* command into Hexadecimal *0X73687574646f776e* to enable the attack to bypass relevant filters. There are different characters in different types of encoding, and the specific form of attacks is not easy to be identified. If this type of attacks also achieves injection by changing the semantics of the conditional clause of SQL statements, the reason may be the use of alternate encoding for the special characters like *'*, *"*, *--*, and *#*.

In the SQL_7 statement, the attacker substitutes '<' and '>' with codes *0X3C* and *0X3E* to ensure that the statement can bypass the developer's censorship mechanism, insert some script information into the database, display the script on the web page

via the SQL query information mechanism, and finally achieve a stored cross-site scripting (XSS) attack. As the syntax and semantics of the SQL statement are not changed by its storage and query process, this type of attacks is difficult to be detected and prevented.

(6) **Illegal/Logically Incorrect Queries**: an attack string is constructed to make the syntax of the SQL statement incorrect. Based on the error information returned from the page, the attacker can obtain sensitive information (such as backend database type, data table structure, table fields, etc.), or provide support for other forms of attacks. An example of such an attack is shown in SQL_8.

SQL_8	SELECT * FROM student WHERE user = 'SQLi' **and covert (int, 'test')#**' AND pass = 'SQLi'

This type of attacks aims to make the SQL statement execute abnormally in diverse forms of attacks, and it is difficult to construct matching rules to detect it. If the constructed attack string contains special characters like *', '', --, and #* that can change the semantics of the SQL statement, it will cause Illegal/logically incorrect queries-based SQL injections.

(7) **Stored Procedures**: by encapsulating the SQL statement and providing reserved interfaces for calling, the developer has implemented measures to obfuscate the SQL statement used within the storage procedure. This makes it difficult to directly obtain the SQL statement, its corresponding interface names, and variables during runtime. For example, the SQL_9 is the statement for the storage procedure SQL, and the statement SQL_{10} is used for calling.

SQL_9	SELECT * FROM student WHERE user = '**param1**' AND pass = '**param2**'
SQL_{10}	CALL storeProcedureName(param1,param2)

The attacker can change the semantics or syntax of the SQL statement by constructing Call function parameters *param1* and *param2*. This type of attacks may cover the aforemethioned six types of attacks.

In the SQL statements explained above, if the attack string originates from a PDS, all the attacks can be categorized as second-order SQL injection vulnerabilities, according to the definition of second-order SQL injections in [19].

III. SQLPsdem Design and Implementation

As explained in Section II, there are many variations of injection attacks. A complex web application usually has many SIPs that attackers can exploit. A high number of attack variations and SIPs make it challenging to detect SQL injection vulnerabilities accurately.

- To address the challenge of missing SIPs, we design a systematic and holistic static analysis approach. The static analysis aimed to identify SIPs is explained in Sections III-A1, III-A2 and III-A3.
- To address the attack variation challenge, based on identified SIPs, we inject malicious payloads using multiple penetration testing tools, which complement each other to cover all the attack patterns explained in Section II. To guarantee differentiation between SQL tokens in the generated attacks and those in the source code, thereby reducing the risk of false detection in subsequent stages, we annotate SQL tokens in the source code using tags as part of the preparation for attack generation. Steps for attack generation preparation and execution are explained in Sections III-A4 and III-B.
- SQL injection attacks aim to read or write data in web applications' databases, regardless of whether the attacks are first or second-order. To accurately identify injection payloads through SIPs that lead to injection exploitation, we design a proxy between the web application and the database to catch SQL queries and commands from the application to the database. We utilize annotation tags to distinguish SQL tokens within the source code from those originating from other resources. Subsequently, we compare the SQL queries and commands against well-established rules to detect, locate, and prevent SQL injections before they are executed in the database. The methods are described in Section III-B.

The strategies are integrated and implemented in SQLPsdem, as shown in **Fig. 2**.

A. Static Analysis Module

The static analysis module is further divided into two steps. One is SIP identification from Section III-A1 to Section III-A3, and the other is preparation for generating attacks in Section III-A4.

1) Identifying and Locating SQL Statements: There are multiple types of SQL statements and subtle differences in their syntax among different types of databases. In order to find all the SQL statements from the complicated web application source code, we develop a strategy to identify the location of SQL statements based on the SQL execution function. The idea of this paper is to limit its focus to popular PHP and MYSQL-based web application. The functions of SQL statements [31] can usually be divided into Data Definition Language (DDL), Data Query Language (DQL), Data Manipulation Language (DML), and Data Control Language (DCL). DQL and DML act on the table content in the database. Their first reserved words of SQL statements are usually *Select, Update, Delete, Insert,* and *Call* to call the stored procedure statements. DDL and DCL act on the structure of the database. Their first reserved words of the SQL statements are usually *Create, Alter, Drop, Truncate,* and so on.

We locate SQL statements based on functions, such as *mysql_db_query*, which execute SQL statements. More examples of functions executing SQL statements in the PHP code are shown in Table II. We first analyze the delivery modes (direct delivery or variable transfer) of execution functions. Direct delivery means that SQL statements are the parameters of execution functions, such as $mysqli\_query$ ("*select * from ...*"). Variable transfer means that SQL statements are delivered to execution functions, such as $mysqli\_query(\$sql)$, as variables. For direct delivery, we can directly locate SQL statements. For variable transfer, php-parser [32] is used to scan the file

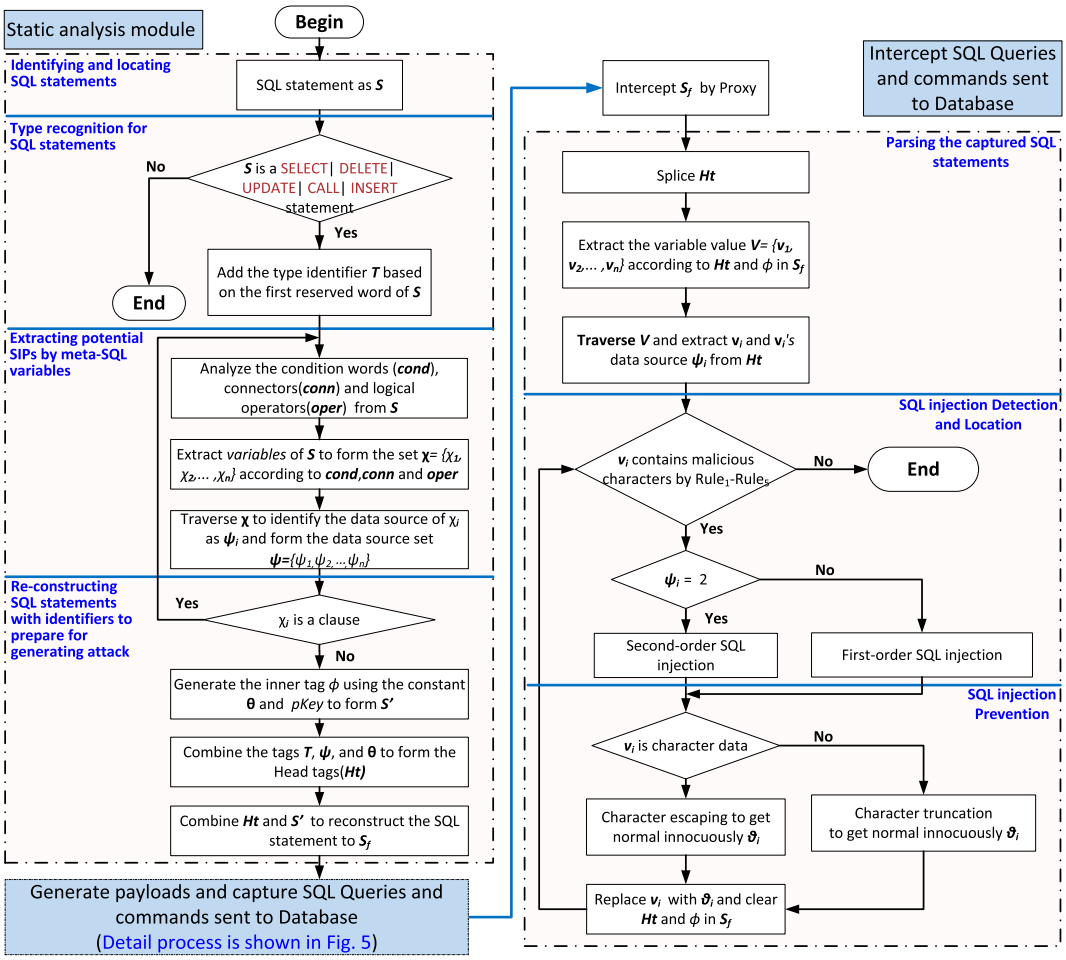


Fig. 2. The overview process of static analysis and dynamic execution modules of SQLPsdem.

TABLE II
THE RELATED MYSQL EXECUTION FUNCTIONS IN PHP

PHP Version	Execution Function
PHP4, PHP5	mysql_db_query() mysql_query()
PHP5, PHP7, PHP8	mysqli_multi_query() mysqli_query() mysqli_real_query()
	mysqli::multi_query() mysqli::query() mysqli::real_query()

```
1.<?php
2.....
3.$conn = DatabaseConnect();
4.$sql = "SELECT * FROM tblName where user='username' ";
5.$sql .= " and password='pw' ";
6.mysqli_query($conn, $sql);
7.mysqli_query($conn, "DELETE * FROM tblName where user='username' ");
8.....
9.?>
```

Fig. 3. SQL statement code snippets.

where the currently executing function is located and output the corresponding abstract syntax tree (AST). We perform intra-procedure analyses. The data flow of the AST is analyzed, and the SQL statement containing the target variable name is extracted to locate the SQL statement. As shown in the example in **Fig. 3**, the delivery mode of the execution function in Line 6 is a variable transfer. Here, the value of the parameter $\$sql$ is the SQL statement we seek. The execution function Line 7 uses a direct delivery mode, and the SQL statement is the value of the second parameter.

2) Type Recognization for SQL Statements: After locating the SQL statements, our next step is to identify the SIPs, which are the variables in static SQL statements. An SQL statement can contain several sub-statements and clauses connected by reserved words, such as *where*, *group by*, and *order by*. The SIPs can be found in SQL statements and their sub-statements or clauses. Thus, we first identify the type of SQL statements, then extract their sub-statements and clauses if there are, and finally search for the SIPs in different elements of the SQL statements.

In general, the first reserved word of a SQL statement determines the type of the SQL statement. For example, when the

reserved word is $Select$, it is a query statement. Therefore, We first covert a SQL statement to *S(T, K)* where *T* denotes its type and *K* is its first reserved word. Based on the value of *K*, we can classify *T* into the following five types.

- If $T = \zeta$, the SQL statement is a Select statement or a Delete statement, and the *SIPs* of this SQL statement mainly exist in *Where* and *Having* clauses.
- If *T* = *U*, the SQL statement is a statement updating the database table information, and the *SIPs* exist not only in *Where* and *Having* clauses, but also in the *Set* clause.
- If *T* = *C*, the SQL statement is a statement that calls a stored procedure, and this type of statement contains only the parameter values passed to the underlying SQL. Then, the parameter values in the current statement are *SIPs*.
- If *T* = *I*, then the SQL statement is a statement that inserts information into the database table, and the *SIPs* exist in the *Values* and the *Value* clause or *Set* clauses (a syntax supported by MySQL only).
- When the SQL statement does not belong to any of the types listed above, it is usually a SQL clause rather than a principal SQL statement. Then, we assign the value *null* to *T*.

3) Extracting Potential SIPs by Meta-SQL Variables: Apart from the first reserved word, the condition words (*Where*, *limit*, "()", etc.), connectors (*and*, *or*, etc.) and operators (=, >, etc.) are related to the structure of the statement. Here, we use *cond* to represent the condition words in S(T, K), *conn* to represent the connectors, and *oper* to represent the logical operators. We analyze and summarize them in **Equations (1)–(3)** below.

$$cond = \begin{cases} \{Where, Group\ By, Having, Order\ By, \\ Limit\}, if\ T = \zeta \\ \{Set, Where, Having, Order\ By, \\ Limit\}, if\ T = U \\ \{()\}, if\ T = C \\ \{Values, Value, Set\}, if\ T = I \end{cases} \quad (1)$$

$$conn = \begin{cases} \{And, Or, \&\&, ||\}, if\ T = \zeta\ or\ U \\ \{,, =\}, if\ T = C\ or\ I \end{cases} \quad (2)$$

$$oper = \begin{cases} \{>=, <=, <>, >, <, =, !=, like, rlike, in, \\ between\}, if\ T = \zeta\ or\ U \\ \oslash, if\ T = C\ or\ I \end{cases} \quad (3)$$

After the *cond*, *conn* and *oper* are extracted by **Equations (1)–(3)**, we can divide *S(T, K)* accordingly to extract corresponding variables $\chi = \{\chi_1, \chi_2, ..., \chi_n\}$, which are potential SIPs. The steps are as follows.

- Partition the statement *S(T, K)* by ***cond***, and extract statement fragment S_0 which contains connector *conn*. For example, we divide a SQL statement into two parts by the condition word *where*, and extract the attribute variables from the statement fragment after *where*.
- There may be more than one "attribute name-attribute variable" pair in the statement fragment S_0 connected by the connector ***conn***, such as *username='$user' and password='$pwd'*. Therefore, by subdividing such pairs using the string *conn*, we can get an ordered list of "attribute name-attribute variable" pairs. Similarly, for the "Attribute name-attribute variable" pairs connected by the logical operator ***oper***, we can partition them to the ordered variable set $\chi = \{\chi_1, \chi_2, ..., \chi_n\}$.

4) Re-Constructing SQL Statements With Identifiers to Prepare for Generating Attacks: In this step, we reconstruct SQL statements of the web application to be tested. We add extra information, called identifiers, to the SQL statements. The identifiers consist of Head tags and Inner tag, which are employed to annotate SQL tokens within the source code. This ensures their consistent differentiation from SQL tokens originating from other sources. We implement this differentiation to minimize false results in the dynamic analysis stage. In the reconstruction, we need to keep the original systems of SQL statements intact.

(1) **Head tags**

Head tags (Ht) of SQL statement S(T, K) are formed according to **Equation (4)**.

$$Ht = [existT\_\psi_1\_\psi_2\_...\_\psi_n\_\theta], \quad (4)$$

where "exist" is a keyword we choose to use to label Head tags. Users can change this keyword to any other strings they like to use. *T* is the value of SQL statement types, i.e., ζ, *U*, *C*, *I*, *null*. The ψ_i ($1 \le i \le n$) represent the type of the extracted variables $\chi = \{\chi_1, \chi_2, ..., \chi_n\}$ of the SQL statement. The values of ψ_i are calculated as follows.

- 0 represents constant: the variable is composed of fixed characters or numbers in the current SQL statement, such as *admin* or *123*.
- 1 represents user input: a user-initiated value from $_GET and $_POST request, $_QUERY and $_SERVER function; or a preset value obtaining from external files, such as $_GET["username"] or $_param["connectString"] from the "webConfig.xml", which is closely related to first-order SQL injection.
- 2 represents non-user input: the variable takes assignment from Cookie ($_COOKIE) or Session ($_SESSION) or obtains the name of the uploaded or configuration file ($_FILES) or the value read from the back-end database via the SQL statement by execution functions, as shown in **Table II**. The value obtained in this way is usually closely related to second-order SQL injection.
- 3 represents that the variable is a SQL clause. For example, when $\$Sql = "select * from table" + \$d$ and $\$d = "where\ username =' admin'"$, $\$d$ variable is a SQL clause, and the value of this type is 3.

The parameter θ of the head tag is a constant value. In this paper, we use "*SQLi*" for θ. The use of constant θ is to generate a random and unique value, which is called the Inner tag.

(2) **Inner tag**

As mentioned, we use penetration testing tools to generate attack payloads for the web application to execute. One challenge of this step is that generated payloads can be identical to reserved SQL keywords, which may mess up the syntax of SQL statements and confuse the interpretation of SQL statements in later steps. To avoid mixing the reserved words of SQL statements and generated payloads, we create a random value, which is very unlikely to be a value of generated payloads, to mark the reserved words. We can use any random number

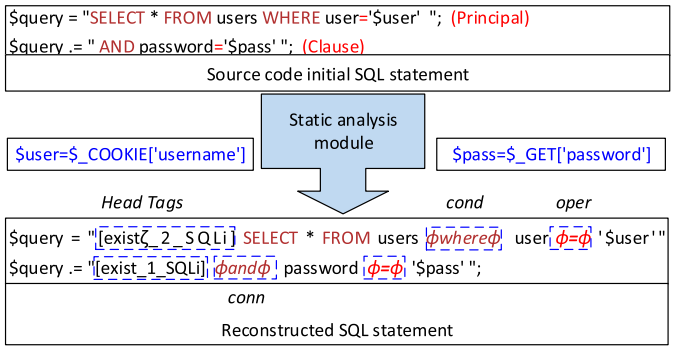


Fig. 4. Examples of SQL statements before and after the reconstruction.

generator to generate such a value. However, we do not want the random number to be too long to make it challenging to store and process. So, we use the MD5 algorithm [33] to generate the hash value of θ first and then perform the "XOR" operation on the harsh value with another constant value *pKey*, to strengthen the uniqueness of the generated value. The generated value is called Inner tag ϕ and is expressed in **Equation (5)**.

$$\phi = MD5(\theta) \oplus pKey \tag{5}$$

In this paper, we use the name of a web application, such as *"DVWA"*, as the *pKey*.

(3) **Using Head tags and Inner tag to reconstruct SQL statements**

There are three steps for the reconstruction of a SQL statement *S(T, K)* using Head and Inner tag. The first step is to calculate ϕ by using **Equation (5)**. The second step is convert all characters in *cond* and *conn* to lowercase characters, and add ϕ before and after each *cond*, *conn* and *oper* to get, for example, "ϕwhereϕ", "ϕandϕ", or "$\phi > \phi$". By performing this conversation to *S(T, K)*, we get *S'(T, K)*. The last step is to combine *Ht* with *S'(T, K)* and achieve the reconstructed SQL statement $S_f(T, K)$, as shown in **Equation (6)**.

$$S_f(T, K) = Ht + S'(T, K) \tag{6}$$

Fig. 4 shows an example of the reconstructing process of a SQL statement, which contains a principal SQL statement and a clause. The reserved word *K* for the principal SQL statement is "SELECT". The type *T* of the SQL statement is ζ, and the type *T* of the clause is *null*. According to **Equation (1)-(3)**, we can get *cond* = {*WHERE*}, *conn* = {*AND*} and *oper* = {=}, respectively, and extract the two variables '*$user*' and '*$pass*' by partitioning the SQL statement. The variables' data sources are $\psi_1 = 2$ and $\psi_2 = 1$, respectively. Assuming that we have θ = "*SQLi*" and *pKey*="*Dvwa*", we get the Inner tag ϕ='9d80700280fda0331ad9bd8d3984cfbf' using **Equation (5)**. Finally, we get Head tags $Ht_{PrincipalSQLstatement}$ = [existζ_2_SQLi] and Ht_{Clause} = [exist_1_SQLi], and the corresponding *cond*, *conn*, and *oper* are converted to "ϕwhereϕ", "ϕandϕ" and "ϕ=ϕ". The reconstructed SQL statements are finally formed using **Equation (6)**, as shown in **Fig. 4**.

B. Proxy-Based Dynamic Execute Module

As mentioned above, the SQL statement reconstruction explained in Section III-A4 is a preparation step for the proxy-based dynamic execution, which acts on the interaction between the web application and the database. The Proxy-based dynamic execute module is shown in **Fig. 5**, which includes three steps.

The first step is to use penetration testing tools, such as JSQL injector (JSQLi) and others listed in [34], to generate a large number of test cases to attack web applications. Penetration testing tools listed in [34] need syntactically correct HTTP request messages between a web application's client and server to generate valid test cases that the web application can execute. To achieve proper HTTP request messages, we deploy and execute the web application to interact with the SIPs we identify through the method explained in Section in III-A3. We designed a web crawler based on Fiddler (*https://www.telerik.com/fiddler*) and Selenium Webdriver (*https://www.selenium.dev/documentation/webdriver/*) to automatically capture the HTTP request messages sent from the web client to the server in application execution. We input the caught HTTP request to penetration testing tools and guide these tools to generate random test cases containing seven types of SQL injections (τ_1-τ_7). As this study serves as a proof-of-concept, and given the varying input requirements of penetration testing tools, the execution of the web application and the guidance for generating seven types of SQL injections were carried out manually. In the future, we can develop adapters for each penetration testing tool to automatically generate the seven types of SQL injections based on the specific information format needed by these tools.

After the execution of the generated test cases, web applications have the capability to send requests to the back-end databases in order to manipulate the data stored within the database. The second step of the Proxy-based dynamic execution module is to capture the requests, which are SQL statements with Head tags and inner tag, sent from the web application to the database before the requests execute. We use Mysql-Proxy 0.8.5 as a proxy server and its function *read_query()* to capture requests from the web application to the database.

The last step is to parse the captured SQL statement to extract the actual values from SIPs for matching with SQL injection vulnerability detection rules defined in Section III-B 2. The SQL statement parsing and matching with the rules are implemented using the *lua* language, which is embedded in the Mysql-Proxy 0.8.5.

1) Parsing the Captured SQL Statements: To parse the captured SQL statement, the first step is to analyze Head tags by extracting the information separated by separators in *Ht* (such as the "_" character used in this paper). In *Ht*, we can get the type *T* of the SQL statement, the data source of each variable $\psi = \{\psi_1, \psi_2, ..., \psi_n\}$, and the parameter θ used to generate the Inner tag.

Based on θ in Head tags, we use **Equation (5)** to calculate the Inner tag ϕ. Through ϕ, we can identify the reserved words, i.e., *cond*, *conn*, and *oper*, in the SQL statement and parse the SQL statement accordingly to exact the variable values in the

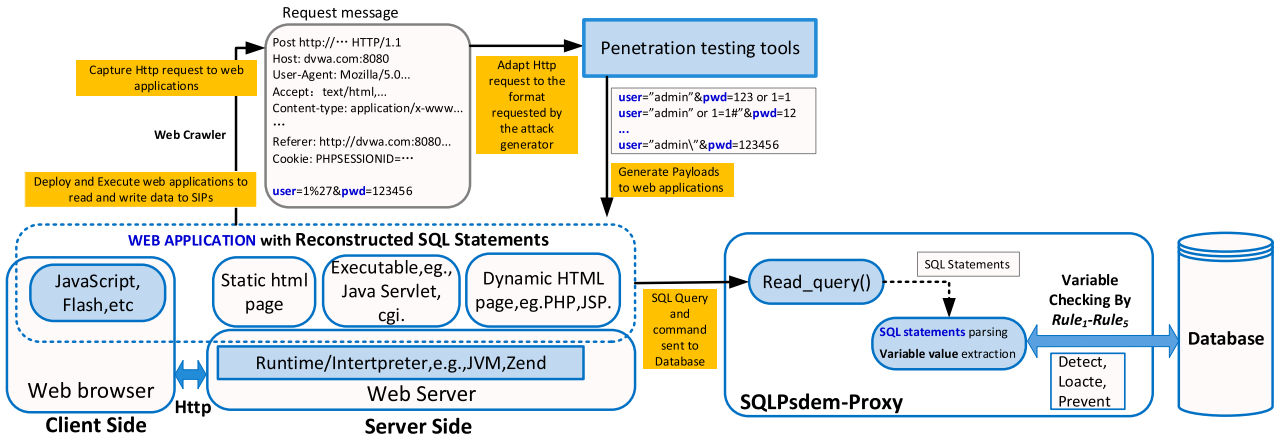


Fig. 5. Mechanism of generating payloads and capturing SQL Queries and commands sent to Database in dynamic execute module.

statement. For example, if a SQL clause is S_0 = {ϕ*where*ϕ username ϕ=ϕ "admin" ϕ*and*ϕ password ϕ=ϕ "pwd"}, by stripping off ϕ, we can identity the reserved word *where*, which is one of the condition words. By parsing S_0 further, we know that S_0 contains multiple "attribute name-attribute value" pairs connected by *conn* connectors, which are encapsulated by ϕ, such as *username* ϕ=ϕ *"admin"* ϕ*and*ϕ *password* ϕ=ϕ *"pwd"*. A set of "attribute name-attribute value" pairs can be obtained by further partitioning S_0 via ϕ*and*ϕ. The specific attribute value of the variable $v_i \in V$ can be obtained by partitioning the set of "attribute name-attribute value" pairs again via identifying *oper* operators and striping off ϕ. Then, we obtain the specific attribute value of the variable $v_i \in V$. For instance, v_i = *"admin"* can be obtained by partitioning *username* ϕ=ϕ *"admin"*.

2) SQL Injection Detection and Location: To identify SQL injection vulnerabilities, SQLPsdem focuses on checking whether there are malicious characters such as ', ", or # in the variable set V = {v_1, v_2,..., v_n} that will cause syntax or semantic changes in the SQL statement. First, we need to identify characters which are prone to cause SQL injections, in the variable $v_i \in V$, and assign $\delta(v_i)$, the discriminative character, to the variable v_i. We apply the following rules to assign the value of the discriminative character.

- If v_i is a string data starting with ", such as v_i ="*admin*", then $\delta(\boldsymbol{v_i})$**="**;
- If v_i is a string data starting with ', such as v_i ='*admin*', then $\delta(\boldsymbol{v_i})$**='**;
- If v_i is a numeric data or a string starting with a number, such as v_i = 123, then $\delta(v_i)$ is *null*.
- If v_i is a time function call, such as v_i = curDate(), then $\delta(\boldsymbol{v_i})$**=0**, and $\delta(\boldsymbol{v_i})$ is not an attack.

Based on the value of the discriminative character $\delta(v_i)$, we convert v_i to $\boldsymbol{v_i}$**'** to get the actual value from SIPs.

- If $\delta(v_i)$ =" or $\delta(v_i)$ =', we remove the first and last characters of $\boldsymbol{v_i}$, in order to get $\boldsymbol{v_i}$**'**, the actual value of $\boldsymbol{v_i}$. For instance, if v_i = ***"admin" or 1=1#"***, then v_i*'* = ***admin" or 1=1#***.
- If $\delta(\boldsymbol{v_i})$**=*null*** or $\delta(\boldsymbol{v_i})$**=0**, then v_i*'* equals v_i, such as v_i*'* = v_i = 123.

Based on the seven types of SQL injections in **Table I**, we design five types of rules (*Rule*$_1$-*Rule*$_5$). We find that the attack strings in SQL statements are structural, especially for the three types of injections (τ_1 - τ_3), whose constituent units are relatively fixed. Thus, we design the rules through unit setting and regular expressions. The unit setting means that we use '[]' to represent an independent unit in rules. For instance, in terms of *Tautologies* or *Union queries* attacks, they contain the unit "or", "and" or "Union" for attack strings. Before or after these core units, some prefix and suffix units are attached to represent the structure of v_i*'*. Among each Unit, regular expressions are performed. For the other four types of SQL injections (τ_4 - τ_7), it is difficult to construct clear matching rules to detect them due to their complex and changeable attack forms, but their attack values are also related to single quotes, double quotes or special numbers.

We match v_i*'* with the rules to identify if v_i*'* contains SQL injection attacks. When $\delta(v_i)$ = ", $\delta(v_i)$ = ', $\delta(v_i)$ = *null*, or $\delta(v_i)$ = 0, it means that v_i starts with ", ', numeric data or a string starting with a number, and a time function call. For the corresponding v_i*'*, if its value matches the following rules, we believe there is an injection attack.

- **Rule**$_1$**(Attack type of** τ_1**)**: p_1 = [$\forall$] [or|and] [$\equiv$][#|- -]. This pattern matches the strings containing connectors "or" or "and". When the calculation result of $\equiv$ (e.g., "1=1", "2 > 1") is true, no matter what the $\forall$ (any string) is, v_i in a SQL statement can result in a *tautology* injection. # or - - are the annotation symbols.
- **Rule**$_2$**(Attack type of** τ_2**)**: p_2 = [$\forall$] [union|union all] [Sk] [#|- -]. This pattern matches the strings containing the command "union" or "union all", which will connect other SQL statements beginning with SQL keywords (Sk). For Rule$_2$, the regular expression implies that any input starting with $\forall$ followed by either *union* or *union all* with a *Sk* behind it will be classified as malicious. For example, when the v_i = *"admin" union select * from table #"*, v_i in a SQL statement can lead to an *union queries* injection.
- **Rule**$_3$**(Attack type of** τ_3**)**: p_3 = [$\forall$] [;] [Ss|Sc] [#|- -]. This pattern matches the strings containing the character

";", and strings as multiple SQL statements (Ss) or SQL commands (Sc). For $Rule_3$, the regular expression means that any input containing *Ss* or *Sc* after the character ";" will be considered as malicious. For example, when the v_i = *"admin"; select * from user #"*, the extra SQL query "*select * from user*" can leak the private information of users.

- **$Rule_4$(Character attack type of $\tau_4, \tau_5, \tau_6, \tau_7$)**: p_4 = [$\delta(v_i)$] && p_4 != [\|\|Ms]. This pattern matches the strings containing malicious symbols (Ms) such as single quote, double quote and \, which will change the meaning of input. For Rule $_4$, the regular expression means that if a string input contains extra single or double quotes, it will lead to attacks. For instance, $\delta(v_i)$ =" after *admin* in v_i =" **admin"** " will cause the SQL statement to report an error, while v_i =" **admin\"** " is a benign string, because the " \ " is an escape symbol which makes the extra double quote normal.
- **$Rule_5$(Numeric attack type of $\tau_4, \tau_5, \tau_6, \tau_7$)**: p_5 != [0 - 9]{4} - [0 - 9]{1,2} - [0 - 9]{1,2}. This sub-type of rule matches date time, such as 2022-05-23. p_6 != [-]{0,1}[0 - 9]{n}. This sub-type of rule matches positive and negative integers, such as -21 or 21. p_7 != [-]{0,1}[0 - 9]{0,n}[.][0 - 9]{n}. This sub-type of rule would match positive and negative decimals, such as -21.3 or 21.3.
  p_8 != [-]{0,1}[0 - 9]{0,n}[.][0 - 9]{n}[E|e][-]{0,1}[0-9]{1,n}. This sub-type of rule matches the number of scientific notation, such as 5.21E5.
  p_9 != [TRUE|FALSE|NULL|TF]. This particular sub-type of rule is designed to match values of the "Boolean" type, "NULL" values, and time functions (TF) such as "now()" and "curDate()".

Based on the matching results using $Rule_1$-$Rule_5$ and the information about the data sources of v_i, we can conclude:

- If v_i satisfies $Rule_1$-$Rule_5$ and ψ_i = 2, which can be found in the value of ψ_i in the reconstructed SQL statement, meaning that the value of v_i comes from PDS, then v_i can cause a second-order SQL injection attack.
- If v_i satisfies $Rule_1$-$Rule_5$ and ψ_i = 1, meaning that the value of v_i comes from user input directly, then v_i can cause a first-order SQL injection attack.
- If v_i does not satisfy $Rule_1$-$Rule_5$ or ψ_i = 0, then v_i is a benign string.

To report the detection results, we output the variable v_i that can cause the SQL injection in the format [*injectionType* | *injectionPos* | v_i | $Rule_j$ | *executeSQL* | *file* | *appName* | *toolName*], where *injectionType* is the type of attack that v_i can cause, such as first or second-order SQL injections, *injectionPos* is the location of v_i in the SQL statement, $Rule_j$ is the rule that v_i matches, *executeSQL* is the SQL executed in the back-end database, *file* indicates the *executeSQL* located in the current web application, *appName* indicates the web application to be detected, and *toolName* indicates the automatic penetration test tool used to generate attack cases. Then the location (code line number) of vulnerabilities can be retrieved by these information.

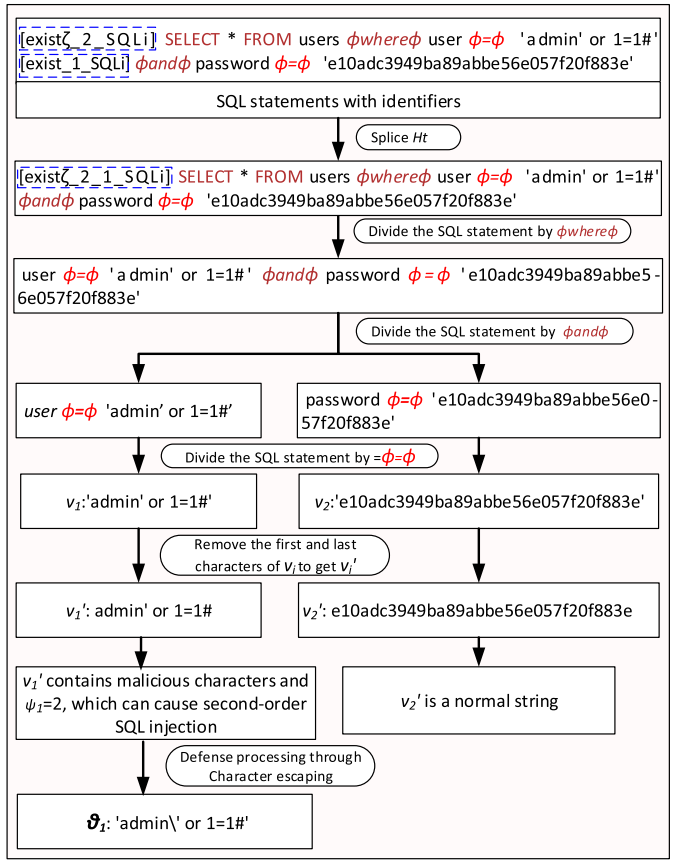


Fig. 6. An example of parsing SQL statement, detecting and preventing injections in dynamic execution module.

3) SQL Injection Prevention: The essence of SQL injection is to change the original semantic structure of SQL statements and execute extra SQL commands. We apply character escaping and truncation defense strategies to defend against exploitations of first and second-order SQL injection vulnerabilities. The defense strategies and their examples are shown in **Table III**.

An example in **Fig. 6** shows the detection and prevention process explained in Sections III-B1, III-B2, and III-B3. This example presents how to detect and prevent SQL injections through the reconstructed SQL statement in **Fig. 4**. We splice the principal SQL statement and its clauses, followed by their Head tags. Thus, we can get merged Head tags *Ht* = *[existζ_2_1_SQLi]* of the SQL statement, the SQL statement type $T = \zeta$ and the variable data sources ψ_1 = 2 and ψ_2 = 1. Second, based on θ = SQLi and *pKey*, we calculate the Inner tag ϕ, and divide the SQL statement by matching $\phi where \phi$, $\phi and \phi$ and $\phi = \phi$, and two variable values v_1'(*admin' or 1 = 1*), v_2'(e10adc3949ba89abbe56e057f20f883e) are extracted. Third, the rule matching is conducted on v_1' and v_2' to detect SQL injections. A second-order SQL injection vulnerability in v_1' is detected by ψ_1 = 2 and $Rule_1$. Then, prevention is performed by character escaping to defend against the exploitation.

TABLE III
SQLPSDEM DEFENSE IMPLEMENTATION STRATEGY

$\delta(v_i)$	Example of Malicious Values	After Implementing Defenses	Defense Operation
" or '	"admin … \" "admin' … \" "admin" or 1=1#" "admin" or 1="1" "admin" union select …#" "admin" ; delete table…#" …	"admin … \\" "admin' … \\" "admin\" or 1=1\#" "admin\" or 1=\"1" "admin\" union select …\#" "admin\" ; delete table…\#" …	**Character escaping.** Escape malicious characters such as single quote ('), double quote ("), backslash (\), comment character (#, - -), by adding a preceding backslash to neutralize their syntactical effect.
null or 0	123 \ 123 or 1=1 123 or 1='1' 123 union select… 123 delete table… 123.456 or 1=1 -123.456 union select … +123.456 delete table … curDate() union select … …	123 123 123 123 123 123.456 -123.456 +123.456 curDate() …	**Character truncation.** Remove malicious characters including comment character (#, - -), operator (and, or, =), separator character (;, -), SQL keywords (union, union all, select, et al.) and scientific calculation identifier (E, e).

TABLE IV
DESCRIPTION OF THE WEB APPLICATIONS

Apps.	Runtime Environment	CVE-ID (SQLi)	Lines of Code
Dvwa1.1.0 [36]	Php 7.4.18	-	77837
Pikachu [37]	Php 7.4.18	-	24533
Bwapp [37]	Php 5.4.31	-	172838
Sqlilabs [37]	Php 5.4.31	-	74058
Schoolmate1.3 [38]	Php 5.4.31	-	8885
Faqforge1.3.2 [39]	Php 5.4.31	-	1990
Wackopicko [40]	Php 5.4.31	-	7408
Mybloggie2.1.4 [40]	Php 5.4.31	CVE-2006-4042 CVE-2005-1499	12873
Webchess1.0 [41]	Php 5.4.31	CVE-2019-20896 CVE-2023-22959	11551
COVID1.0 [42]	Php 7.4.18	CVE-2021-33470	176952
Doctormms1.0 [43]	Php 7.4.18	CVE-2021-27320 CVE-2021-27319 CVE-2021-27316 CVE-2021-27315 CVE-2021-27314 CVE-2021-27124	7317
Pet shop1.0 [44]	Php7.4.18	CVE-2021-35458 CVE-2022-41377 CVE-2022-40935 CVE-2022-40934 CVE-2022-40933 CVE-2022-41408 CVE-2022-41407 CVE-2022-41378	732469

IV. EVALUATION RESULTS

A. Environment Deployment and Test Beds Description

We evaluate SQLPsdem on 12 widely used open-source web applications developed in PHP language [3]. Our evaluation focuses on SQLPsdem's detection performance for first and second-order SQL injection vulnerabilities, SQLPsdem's prevention performance and the overhead incurred by SQLPsdem in terms of page response time. The information on these web applications is shown in **Table IV**, which includes the applications' name and their runtime environment. The CVE-ID is the ID of the known vulnerabilities corresponding to the web application in CVE [35]. All the web applications used for evaluation are installed and deployed in XAMPP.

Table V shows the number of test cases generated by the penetration testing tools listed in [34], namely, Sqlmap, JSQL injector (JSQLi), Super SQL injection (SSQL), and Damn Small SQLi Scanner (DSSS), to attack web applications and the mapping between the test cases and $Rule_1$-$Rule_5$. Results show that there are many diversities in generated 52,771 test cases. For τ_1, τ_2 and τ_3, there are fewer test cases generated than others. This is because the statistics of τ_1, τ_2 and τ_3 come from precise rule matching. Other attack cases are matched and classified as $\tau_4, \tau_5, \tau_6, \tau_7$ (Character or Number), and there are bound to be a large number of redundant cases. The precise rule matching may result in fewer cases related to τ_1, τ_2 and τ_3. However, each type of these test cases is tested in several of the 12 web applications, which can verify the defensive performance of SQLPsdem when facing these three types of test cases.

B. SQL Injection Vulnerability Detection Performance

When selecting tools for comparison, we primarily consider the following aspects: open source, capability to detect SQL injection attacks, popularity, and industry relevance. Ultimately, we utilize two well-known open-source tools, namely RIPS [45] and phpvulhunter [46]. In addition, we also incorporate Ardilla [47], a notable scientific tool, and Acunetix [48], a widely recognized and popular commercial tool. The detailed information and the setting of the parameters of each tool are as follows.

- RIPS is a popular open-source static code analysis tool designed for PHP applications. It focuses on detecting various vulnerabilities, including common SQL injection attacks, through static analysis techniques. It is deployed in Php 7.4.18. In our evaluation, its vulnerability type is set to the SQL injection, and all the other parameters are set to default.
- phpvulhunter is an open-source tool aimed explicitly at PHP vulnerability scanning. It is deployed in Php 5.4.31. In our evaluation, all its parameters are set to default.
- Ardilla is an automated tool designed to detect SQL injection and XSS attacks in PHP Web applications. It is not available as open source, so it is used by default settings. Ardilla is based on a scientific paper [47], and it has

TABLE V
DISTRIBUTION OF ATTACK CASES GENERATED IN CORRESPONDING WEB APPLICATIONS BY PENETRATION TESTING TOOLS

	Attack Form	Dvwa	Pikachu	Bwapp	Sqlilabs	Schoolmate	Faqforge	Wackopicko	Mybloggie	Webchess	Petshop	Doctormms	Covid
Sqlmap	τ_1	1	0	0	0	208	0	0	0	85	0	2	0
	τ_2	0	0	0	0	110	0	0	0	266	0	32	0
	τ_3	2	10	4	17	233	4	6	2	93	14	86	34
	$\tau_4, \tau_5, \tau_6, \tau_7$ (C)	15	165	11	880	1143	7	23	20	532	298	565	106
	$\tau_4, \tau_5, \tau_6, \tau_7$ (N)	2907	710	990	2837	7999	310	285	185	2563	1045	1852	773
	Total	**2925**	**885**	**1005**	**3734**	**9693**	**321**	**314**	**207**	**3539**	**1357**	**2537**	**913**
JSQLi	τ_1	0	0	0	0	252	0	0	0	30	0	16	77
	τ_2	0	0	0	0	72	0	0	0	0	0	0	36
	τ_3	4	6	6	10	148	0	0	0	0	12	8	83
	$\tau_4, \tau_5, \tau_6, \tau_7$ (C)	10	30	58	162	1019	0	0	0	54	90	98	192
	$\tau_4, \tau_5, \tau_6, \tau_7$ (N)	155	295	698	625	1237	57	12	0	114	324	424	315
	Total	**169**	**331**	**762**	**797**	**2728**	**57**	**12**	**0**	**198**	**426**	**546**	**703**
SSQL	τ_1	0	61	12	0	85	0	12	0	73	0	8	109
	τ_2	0	0	0	0	36	0	0	0	0	0	0	0
	τ_3	0	14	0	7	95	5	2	0	14	12	6	18
	$\tau_4, \tau_5, \tau_6, \tau_7$ (C)	0	160	60	708	515	3	20	0	110	87	138	159
	$\tau_4, \tau_5, \tau_6, \tau_7$ (N)	376	268	123	338	2282	191	34	101	727	297	263	309
	Total	**376**	**503**	**195**	**1053**	**3013**	**199**	**68**	**101**	**924**	**396**	**415**	**595**
DSSS	τ_1	1	40	27	1	195	0	6	0	20	16	32	48
	τ_2	0	0	4	0	37	0	0	0	0	0	0	0
	τ_3	2	0	6	2	109	4	2	0	0	12	0	0
	$\tau_4, \tau_5, \tau_6, \tau_7$ (C)	5	247	80	2	735	7	28	0	100	170	160	240
	$\tau_4, \tau_5, \tau_6, \tau_7$ (N)	238	530	96	3300	1430	215	56	129	1130	487	331	494
	Total	**246**	**817**	**213**	**3305**	**2506**	**226**	**92**	**129**	**1250**	**685**	**523**	**782**

TABLE VI
THE NUMBER OF SQL INJECTION VULNERABILITIES REPORTED BY THE EVALUATED TOOLS

Apps.	**RIPS**	**php-vulhunter**	**Ardilla**	**Acuntix**	**SQLPsdem**
Dvwa	4	12	-	1	7
Pikachu	5	15	-	14	20
Bwapp	9	0	-	11	14
Sqlilabs	40	17	-	16	56
Schoolmate	124	0	6	1	125
Faqforge	12	0	1	11	19
Wackopicko	3	0	-	1	5
Mybloggie	1	0	-	2	5
Webchess	19	30	12	12	27
COVID	18	0	-	26	31
Doctormms	11	1	-	36	49
Pet shop	0	0	-	16	43
Total	246	75	19	147	401

TABLE VII
THE DETAILED DETECTION RESULTS OF SQLPSDEM

Apps.	SQLPsdem				
	Number of SQLi	FO	**FFP**	SO	**SFP**
Dvwa	7	5	0	2	0
Pikachu	20	18	0	2	0
Bwapp	14	14	0	0	0
Sqlilabs	56	52	0	4	0
Schoolmate	125	117	0	3	5
Faqforge	19	19	0	0	0
Wackopicko	5	5	0	0	0
Mybloggie	5	5	0	0	0
Webchess	27	15	0	12	0
COVID	31	31	0	0	0
Doctormms	49	46	0	3	0
Pet shop	43	43	0	0	0
Total	401	370	0	26	5

been cited more than 550 times, which demonstrates its influence and relevance in the field.

- Acunetix is a widely recognized and commercially available web vulnerability scanner. It is renowned for its comprehensive scanning capabilities and extensive built-in vulnerability detection rules. As one of the industry-leading tools, Acunetix is widely used for web application security testing [49]. We include Acunetix in the comparison to benchmark our approach against widely used commercial tools in the industry. In our evaluation, Acunetix's parameters are set as follows. The Business Criticality parameter is set to Normal, the Default Scan Profile is set to the SQL injection, Scan Speed is set to Slow, Proxy is set to Google Chrome, and Case Sensitive Paths is set to Yes, while other parameters are set to default.

The vulnerability detection results are shown in **Tables VI** and **VII**, where "-" indicates that the tool cannot work on these web applications. In **Table VII**, "Number of SQLi" is the number of SQL injection vulnerabilities detected by the tool, "FO" and "SO" are the numbers of first and second-order vulnerabilities detected by SQLPsdem respectively, "FFP" and "SFP" are the numbers of false positives for first and second-order SQL injections.

As shown in **Table VI**, SQLPsdem detects 401 no-repeated SQL injection vulnerabilities, which are more than the results detected by the four tools in comparison. In addition, the four tools cannot differentiate first and second-order SQL injection vulnerabilities. In **Table VII**, we find that among the 401 vulnerabilities, 370 are first-order SQL injection vulnerabilities, and 31 are second-order SQL injection vulnerabilities. Although the phpvulhunter tool reports more SQL injection vulnerabilities than SQLPsdem does in Dvwa (12 vs. 7) and Webchess (30 vs. 27), we analyze that 9 of 12 vulnerabilities reported by the phpvulhunter tool in Dvwa are false positives, and only 3 are true positives, while there are five duplicate vulnerabilities among the 30 vulnerabilities discovered in Webchess. In our experiment, SQLPsdem reports zero false positives for first-order SQL injection vulnerabilities. Five vulnerabilities identified by SQLPsdem, which are first-order ones, are misclassified by SQLPsdem as second-order ones in Schoolmate. SQLPsdem, which relies on intra-procedural analysis in static code analysis stage, mistakenly identifies the *$courseid* parameter in the *deleteCourse($courseid)* function and the *$teacherid* parameter in the *deleteTeacher($teacherid)* function in the *DeleteFunctions.php* file as database sources. During static analysis, such misidentification occurs because these parameters are fetched by the *mysql_fetch_row* query

function. However, the actual trigger for these vulnerabilities lies in the calls to the *deleteCourse* and *deleteTeacher* functions in the *ManageClasses.php* and *ManageTeachers.php* files, where the *$courseid* and *$teacherid* parameters are assigned the value of $_POST["delete"] directly, resulting in first-order injection vulnerabilities.

To evaluate the coverage and effectiveness of SQLPsdem, we use the known CVEs of the applications Mybloggie2.1.4, Webchess1.0, COVID1.0, Doctormms1.0, and Pet shop1.0, as shown in **Table IV**, as the ground truth. We find that all the CVEs listed in **Table IV** are included in the 401 SQL injection reported by SQLPsdem.

In **Table IV**, CVE-2023-22959 is a recently discovered first-order SQL injection vulnerability in Webchess 1.0. At the time of submitting our work, this vulnerability had not yet been included in the CVE database. SQLPsdem successfully identifies this SQL injection vulnerability in *mainmenu.php*. The disclosure of the latest CVE ID supports our findings and confirms the validity of these newly discovered vulnerabilities.

The information of the 26 identified second-order and 5 misreported first-order SQL injection vulnerabilities is shown in **Table VIII**, where "SQL Query" is the vulnerable SQL statement, "Injection Point" is the variable that can cause injection, and "File Location (Vul. num)" is the file where the SQL statement is located and the number of vulnerabilities. In addition to the 12 reported vulnerabilities (CVE-2019-20896) in Webchess and one vulnerability (CVE-2023-40944, a new CVE ID applied for by us) reported by Yan et al. [12], SQLPsdem has also detected 13 new second-order vulnerabilities. We have successfully applied for three CVE IDs for these vulnerabilities, i.e., CVE-2023-39852, CVE-2023-40945 and CVE-2023-40946. Moreover, CVE-2023-39850 is also a new CVE ID applied for by us for the 5 misreported first-order SQL injection vulnerabilities.

These newly identified SQL injection vulnerabilities can be classified into two categories.

1) Second-Order SQL Injection Vulnerabilities Cannot be Identified Without Analyzing Session Values:

- Schoolmate (CVE-2023-40946): Schoolmate v1.3 is vulnerable to a second-order SQL Injection in the variable *$username* from the session in *ValidateLogin.php*.
- Doctormms (CVE-2023-39852, CVE-2023-40945): Doctormms v1.0 is discovered to contain one second-order SQL injection vulnerability through the *$userid* parameter from the session in *doctors\myAppoinment.php*, and two second-order SQL injection vulnerabilities through the *$userid* parameter from the session in *doctors\myDetails.php*. The *$userid* parameter originates from "$_SESSION["userid"]=$_POST["userid"]" at line 68 in *doctors\doctorlogin.php*. It is not a session variable controlled by the server and does not represent an auto-increment type number in the database. It's vulnerable.

These vulnerabilities are formed by two stages. First, the attacker's malicious input through the POST request is stored in the session. Then, the program retrieves and utilizes the value from the session, which is a complex process that triggers the vulnerability. To our knowledge, three methods [17], [19], [20] can detect SQL injection vulnerabilities in session. However, one [17] cannot detect malicious injections that do not contain SQL keywords, and its detection of vulnerability attack types is limited. Another [19] heavily relies on static analysis and cannot capture specific runtime behaviors and inputs during dynamic execution, resulting in a higher false positive rate. The other [20] suffered from the limitation of capturing numerous potential vulnerability paths that could not be triggered by generating sufficient attack cases, which would lead a low coverage. SQLPsdem can detect different types of attacks originating from sessions by generating tests using state-of-the-art generators, which can create more complex and comprehensive test cases covering seven types of SQL injections. The dynamic execution capability of SQLPsdem allows it to directly observe the actual behavior of malicious inputs in the program, such as whether they alter the structure or semantics of the SQL query, leading to more accurate vulnerability detection.

Eight other newly vulnerabilities in Dvwa, Pikachu, and Sqlilabs are also obtained by analyzing the session and cookie data. However, as Dvwa, Pikachu, and Sqlilabs are intentionally created as vulnerable web applications for penetration training, the vulnerabilities in these web applications are, therefore, not eligible for applying for CVEs.

2) Vulnerabilities Cannot be Identified Without Our Comprehensive Static Analysis:

- Schoolmate (CVE-2023-39850): Schoolmate v1.3 was discovered to contain multiple first-order SQL injection vulnerabilities via the *$courseid* and *$teacherid* parameters from POST requests at *DeleteFunctions.php*.

SQLPsdem can identify newly discovered vulnerabilities because it covers all the SQL queries (including concatenated SQL clauses) and their contextual variables in web applications during static analysis. It also extracts suspicious variables from dynamic SQL queries, offering comprehensive coverage of SIPs (constant values, user input, non-user input like database field data, file, Cookie, and Session data). As a result, SQLPsdem can detect a broader range of SQL injection vulnerabilities effectively.

Apart from the two categories of newly identified vulnerabilities, SQLPsdem is also capable of detecting specific special attacks that cannot be detected by other methods [16], [17], [18], which are explained in Section V-B using examples. However, the web applications we evaluate do not have such vulnerabilities. Thus, we cannot give evidence in the paper that SQLPsdem detects them, although SQLPsdem can, in theory, catch them.

C. Comparison Results on Second-Order SQL Injection Vulnerability Detection or Prevention Methods

Due to the unavailability and potential outdatedness of many second-order SQL injection detection or prevention methods [12], [13], [14], [15], [16], [17], [18], [19], [20], reproducing the experiments of those studies is challenging. To evaluate the superiority of SQLPsdem over other methods using the same benchmark, we gathered the reported numbers of second-order

TABLE VIII
THE DETAIL INJECTION LOCATIONS OF THE 26 SECOND-ORDER AND 5 MISREPORTED FIRST-ORDER SQL INJECTIONS DETECTED BY SQLPSDEM.

Apps.	SQL Query	Injection Position	File Location (Vul. Num)
Dvwa	SELECT first_name,last_name FROM users WHERE user_id ='$id' LIMIT …	$id=$_SESSION['id'] $id=$_COOKIE['id']	∼\vulnerabilities\sqli\source\high.php ∼\vulnerabilties\sqli_blind\source\high.php
Pikachu	SELECT * FROM users WHERE username='$username' AND sha1(password)='$password'	$username=$_COOKIE['ant']['uname'] $password=$_COOKIE['ant']['pw']	∼\inc\function.php (*2)
Sqlilabs	UPDATE users SET PASSWORD = … WHERE username='$username' …	$username=$_SESSION["username"]	∼\Less-42\pass_change.php ∼\Less-43\pass_change.php ∼\Less-44\pass_change.php ∼\Less-45\pass_change.php
Schoolmate	UPDATE schoolinfo SET … where schoolname = '$schoolname'	$schoolname=mysql_rsult($query,0) *Reported by Yan* et al. [12] CVE-2023-40944	∼\header.php
	SELECT type from users where username = $_SESSION[username]	$_SESSION[username] CVE-2023-40946	∼\ValidateLogin.php
	SELECT userid from users where username =$_SESSION[username]	$_SESSION[username] CVE-2023-40946	∼\ValidateLogin.php
	DELETE FROM ... WHERE courseid =$courseid ...	$courseid=mysql_fetch_row($q1) *Misreported (First-order)* CVE-2023-39850	∼\DeleteFunctions.php (*4)
	DELETE FROM teachers WHERE teacherid =$teacherid LIMIT ...	$teacherid=mysql_fetch_row($query) *Misreported (First-order)* CVE-2023-39850	∼\DeleteFunctions.php
Webchess	SELECT nick FROM players, games WHERE playerID =... AND gameID =$_SESSION['gameID']	$_SESSION['gameID'] CVE-2019-20896	∼\chess.php
	SELECT * FROM history WHERE (...gameID = $_SESSION['gameID']...)	$_SESSION['gameID'] CVE-2019-20896	∼\capt.php
	DELETE FROM history WHERE gameID = $_SESSION['gameID']	$_SESSION['gameID'] CVE-2019-20896	∼\newgame.php
	SELECT * FROM messages WHERE gameID = $_SESSION['gameID'] AND ...	$_SESSION['gameID'] CVE-2019-20896	∼\chessdb.php (*2)
	SELECT * FROM ... WHERE gameID =$_SESSION['gameID']	$_SESSION['gameID'] CVE-2019-20896	∼\chessdb.php (*2)
	SELECT ..., ... FROM games WHERE gameID =$_SESSION['gameID']	$_SESSION['gameID'] CVE-2019-20896	∼\chessdb.php (*2)
	DELETE FROM pieces WHERE gameID =$_SESSION['gameID']	$_SESSION['gameID'] CVE-2019-20896	∼\chessdb.php
	INSERT INTO pieces (gameID,...) values ($_SESSION['gameID'],...)	$_SESSION['gameID'] CVE-2019-20896	∼\chessdb.php
	UPDATE games SET lastMove = ... WHERE gameID = $_SESSION['gameID']	$_SESSION['gameID'] CVE-2019-20896	∼\chessdb.php
Doctormms	SELECT * FROM doctor WHERE userid=$_SESSION["userid"]	$_SESSION["userid"] (CVE-2023-40945)	∼\doctors\myDetails.php
	UPDATE doctor SET … WHERE userid= $_SESSION["userid"]	$_SESSION["userid"] (CVE-2023-40945)	∼\doctors\myDetails.php
	SELECT * FROM booking WHERE userid=$_SESSION["userid"]	$_SESSION["userid"] (CVE-2023-39852)	∼\doctors\myAppoinment.php

SQL injection vulnerabilities from papers reporting those methods, utilizing approaches [12], [13], [20] as the benchmarks. The results are presented in **Table IX** below.

As shown in **Table IX**, SQLPsdem successfully identified three vulnerabilities in Schoolmate, containing the vulnerability reported by Yan et al. [12]. However, Li et al. [13] and DISOV

TABLE IX
SECOND-ORDER SQL INJECTIONS DETECTED BY SQLPSDEM AND OTHER APPROACHES

Apps.	SQLPsdem	Yan et al. [12]	Li et al. [13]	DISOV [20]
Schoolmate	3	1	6	1
Webchess	12	0	1	0
Total	15	1	7	1

TABLE X
THE RESULTS OF PREVENTION PERFORMANCE FOR SQLPSDEM

Apps.	TRN	NOP				DFR(%)
		Sqlmap	JSQLi	SSQL	DSSS	
Dvwa	13	2925	169	376	246	100
Pikachu	25	885	331	503	817	100
Bwapp	19	1005	762	195	213	100
Sqlilabs	71	3734	797	1053	3305	100
Schoolmate	44	9693	2728	3013	2506	100
Faqforge	12	321	57	199	226	100
Wackopicko	5	314	12	68	92	100
Mybloggie	6	207	0	101	129	100
Webchess	14	3539	198	924	1250	100
COVID	17	913	703	595	782	100
Doctormms	14	2537	546	415	523	100
Petshop	8	1357	426	396	685	100

TABLE XI
COMPARISON RESULTS OF RESPONSE TIME FOR WEB APPLICATIONS WITH AND WITHOUT SQLPSDEM

Apps.	**ART_N[s](NOP)**	**ART_Y[s](NOP)**	**TD(s)/PD(%)**
Dvwa	0.31(2260)	0.27(1516)	-0.04/-12.90
Pikachu	2.46(188)	0.51(40)	-1.95/-79.26
Bwapp	2.63(728)	0.55(236)	-2.08/-79.09
Sqlilabs	0.19(847)	0.24(323)	0.05/26.31
Schoolmate	0.86(170)	0.41(913)	-0.45/-52.33
Faqforge	0.22(3837)	0.31(1409)	0.09/40.90
Wackopicko	0.25(855)	0.26(354)	0.01/4.00
Mybloggie	0.28(500)	0.27(241)	-0.01/-3.60
Webchess	2.43(481)	0.47(578)	-1.96/-80.66
COVID	1.75(1809)	0.30(117)	-1.45/-82.85
Doctormms	2.04(215)	1.11(261)	-0.93/-45.59
Petshop	1.62(560)	0.31(498)	-1.31/-80.86

[20] did not provide detailed information regarding their detected second-order SQL injection vulnerabilities in Schoolmate and Webchess. Thus, we cannot check if their reported vulnerabilities are accurate. Regarding vulnerabilities in Webchess, SQLPsdem identifies a higher number of second-order SQL injection vulnerabilities than [12], [13], [20]. Our manual inspection confirms all SQLPsdem identified vulnerabilities in Webchess.

D. SQLPsdem Defense Performance

The performance of defending generated attacks is shown in **Table X**. TRN (Test Request Number) in **Table X** is the number of web requests having interaction with the database. NOP (Number of Payloads) is the number of attack cases generated by four penetration testing tools. DFR (DeFense Rate) is the SQLPsdem's prevention rate against these attack cases, with a greater value indicating better effectiveness of prevention. **Table X** shows that SQLPsdem can 100% defend against these attacks. The results indicate that as long as the identifier is added correctly and effectively to relevant SQL statement in the static analysis phase, the interception and corresponding prevention of malicious attack cases in the proxy-based dynamic execution phase would be effective. SQLPsdem employs both escaping and truncation strategies, which are indeed widely recognized as effective practices for preventing SQL injection in the industry. This effectively prevents malicious attacks by different cases and thwarts attackers' objectives, ensuring that user input does not alter the structure of SQL statements and avoids causing database crashes or information leakage.

E. SQLPsdem's Overhead

SQLPsdem parses Head tags and Inner tag that are added by the static analysis module, in the proxy-based dynamic execute module, and it extracts information for analysis, which may cause delays in web application page response. Therefore, we record and compare web application page response time with and without SQLPsdem. In order to increase the reliability of the results, we repeat the experiment five times, and calculate the average time, which are expressed by ART_Y and ART_N. JSQLi, SSQL, and DSSS cannot manually select the type of SQL injection attack to generate attack cases. So, we use only Sqlmap in the experiments. To avoid the intervention of time inference attacks on page response time, we exclude *time inference* attacks generated by the Sqlmap to test SQLPsdem. The results are shown in **Table XI**, where NOP is the number of test cases generated by Sqlmap, TD is the value of the time difference between ART_Y and ART_N, and PD is the percentage difference of TD.

As shown in **Table XI**, the page response time in nine web applications using SQLPsdem is generally less than that without using SQLPsdem. The overhead is no more than 0.09s in the three other web applications (Sqlilabs, Faqforge and Wackopicko). The popular "2-5-8 principle" of response time in software testing filed [50] means that when users can get a response within 2 seconds, they will feel that the system has a quick response. For Pikachu, Bwapp, Webchess and Doctormms, their average response time exceed 2 seconds. The time are shortened a lot when using SQLPsdem. This indicates that the detection and prevention process in the dynamic execute module of SQLPsdem does not cause significant overhead.

Applying plug-ins or agents in web systems usually leads to overhead. However, ART_Y is smaller than ART_N in nine web applications, mainly because SQLPsdem detects and blocks the attack. Without SQLPsdem, the inputs to SIPs may cause SQL injections, execute some extra SQL statements, increase time consumption, and make ART_Y smaller than ART_N.

F. SQLPsdem's Coverage on CIAO and BroNIE Injection Benchmarks

This section compares SQLPsdem with existing studies using the examples (11 CIAOs [29] and 5 BroNIEs [30]) from Ray and Ligatti [29], [30] to show that SQLPsdem has equal or better coverage of these examples than existing approaches do. **Table XII** shows how well eight prior tools and SQLPsdem cover the benchmark cases.

The results in [51], [52], [53], [54], [55] are obtained from DIGLOSSIA [56] and SEPTIC [18]. Except SEPTIC and our approach, all tools have not analyzed the BroNIEs, which is marked as "-". How SQLPsdem and other tools detect the vulnerabilities is explained below. In the following explanations, the underlined terms are the user inputs.

1. SELECT balance FROM acct WHERE password='' OR 1=1 --'

TABLE XII
THE CIAOS AND BRONIES CLASSIFIED BY PRIOR METHODS AND SQLPSDEM

	CIAOs											BroNIEs				
Methods	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Ray and Ligatti [29], [30]	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes
Halfond et al. [51]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes	-	-	-	-	-
Nguyen et al. [52]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes	-	-	-	-	-
Xu et al. [53]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes	-	-	-	-	-
SQLCHECK [54]	Yes	No	No	Yes	No	No	No	No	No	No	No	-	-	-	-	-
CANDID [55]	Yes	Yes	Yes	No	No	No	Yes	No	No	No	Yes	-	-	-	-	-
DIGLOSSIA [56]	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	No	-	-	-	-	-
SEPTIC [18]	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	No	No	-	Yes	Yes	-
SQLPsdem	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	No	No	Yes	Yes	Yes	Yes

This case is the classical SQL injection attack with a back-quote that ends a string and injects user input as a code into the query. All tools detect this code injection. SQLPsdem detects it matched by $Rule_1$.

2. SELECT balance FROM acct WHERE pin= exit()

User input injects exit(), which is a built-in function call. SQLPsdem detects it because the injected code is matched by the $Rule_5$.

3. ...WHERE flag=1000>GLOBAL

Based on $Rule_5$, it is obvious that "1000>GLOBAL" has non-digital characters (such as >), which is recognized by SQLPsdem as an attack.

4. SELECT * FROM properties WHERE filename='f.e'

Although "*f.e*" is a suspicious format for the object reference, quotes force it to become a string. According to $Rule_4$, SQLPsdem correctly classify the input as a literal string instead of an attack.

5. ...pin=exit()
6. ...pin=aaaa()
7. SELECT * FROM t WHERE flag=TRUE
8. SELECT * FROM t WHERE flag=aaaa
9. SELECT * FROM t WHERE password=password

According to $Rule_5$, SQLPsdem can detect the four inputs in example 5, 6, 8, 9 as code injections by taking them as expressions, and can identify the input in example 7 as non code injection, which is the same as DIGLOSSIA [56] and SEPTIC [18].

10. CREATE TABLE t (name CHAR(40))

Unlike Ray and Ligatti [29], all the tools consider integer literals, even in SQL type definitions, to be secure. Thus, this case is not an injection attack. SQLPsdem also thinks "40" is a pure number, not a CIAO.

11. SELECT * FROM t WHERE name='x'

Ray and Ligatti [29], SQLCHECK [54], DIGLOSSIA [56], SEPTIC [18] and SQLPsdem recognize that the input "x" is a string, and there are no characters causing semantic structure changes in it, so it is not a CIAO.

12. SELECT * FROM files WHERE numEdits > 0 and name='file.ext'

Same as example 4, SEPTIC [18] does not consider this instance as an attack. According to $Rule_4$ and $Rule_5$, SQLPsdem also determines that it is not an attack.

13. SELECT * FROM t WHERE c='' and now()<exp - -this code is smokin'

For the input value "'", SQLPsdem identifies that it is a string, and additional single quotes are forming a closed interval, which leads to a change in the SQL semantic structure. According to $Rule_4$, SQLPsdem recognizes this case as an attack.

14. INSERT INTO users VALUES ('evilDoer', TRUE)- -', FALSE)

SEPTIC [18] detect this type as an attack via query structural matching. According to $Rule_4$, SQLPsdem recognizes that it is a string, and there are single quotes that form an extra closed interval, which is an attack.

15. INSERT INTO trans VALUES (1, -5E-10)

In this case, the original intention of the second parameter in Ray [30] is an arithmetic operation. If a character "E" is inserted into an arithmetic operation (minus and plus), it would change its meaning. For SEPTIC [18], the attack is identified via complex structural verification because arithmetic operations generate more nodes in a tree model than a scientific notation number. By matching $Rule_5$, SQLPsdem considers it is an attack.

16. SELECT balance FROM acct WHERE password='\' (A similar example with *$data = '\'; security Check (); $data. = '& f = exit # '; \n f();* in [30])

Entering "\" would make the escape of the following single quote become ordinary characters with no special function and change the structure of the statement. According to $Rule_4$, SQLPsdem considers it is an attack.

In summary, SQLPsdem can effectively detect and identify CIAO or BroNIE attacks on different conditions.

V. RELATED WORK

Compared with existing studies on first and second-order SQL injections, our approach addresses their weaknesses to minimize false results.

A. Studies on the General Detection of SQL Injection Attacks

Static analysis, dynamic analysis and even their combined analysis are usually used to detect SQL injection vulnerabilities. AMNESIA [5] first uses static analysis to build Non-Deterministic Finite Automaton (NDFA) models to express, at the character level, all the possible values the considered string can assume at hotspots and then compares all queries to be sent

to the database against the models. The queries that match the models will be regarded as legitimate and those mismatching ones will be reported as SQL injection attacks. When generating the models, AMNESIA [5] did not annotate the SQL tokens acquired from the source code to ensure that the tokens would not be undistinguishable from the SQL tokens in user inputs, resulting in false negatives of attack detections. The authors of AMNESIA [5] admitted that *"Our technique can produce false negatives when a legitimate query happens to have the same SQL structure of an attack."* The NDFA models tend to be overly generic and oversimplified, leading to false positive detection results. Likewise, WebSSARI [57], a tool that detects input-validation-related errors using information flow analysis, relies its analysis on an unsound assumption that as long as an untrusted value passes a certain kind of validation [22], it is actually safe. Compared with SQLPsdem, WebSSARI [57] overly depends on the predefined set of validation functions subject to those provided by PHP which are not always completely safe and effective. The implementation of state-of-the-art rules enables SQLPsdem to identify more SQL injection vulnerabilities accurately. Besides, the injection-points of WebSSARI [57] are mainly from $_Get, $_POST and $_REQUEST, while SQLPsdem additionally covers $_Session, $_Cookie, file names and query function like *mysql_query_result*, which is more comprehensive.

Bandhakavi et al. [55] designed a CANDID prototype to prevent SQL injection using dynamic analysis. First, benign inputs are used to calculate the intended SQL query of the target. When the web application is running, a series of candidate inputs are filled and compared with the expected SQL structure to determine whether there is an injection. This method requires maintaining the same control path as the intended structure when dynamically identifying the injection, and its detection efficiency depends on manual intervention. Liu et al. [21] proposed an SQL Proxy-based Blocker (SQLProb) consisted of two stages to prevent SQL injection. At the data collection stage, benign inputs are used to collect all the SQL statements of the entire web application. The collected SQL statements are stored in the warehouse. At the query evaluation stage, when the web application is running, SQLProb intercepts SQL statements through the proxy and generates the corresponding syntax tree, and then match the SQL syntax with those stored in the warehouse. The user input extracted through syntax tree comparison is analyzed to determine whether there is a SQL injection. The prevention effect of this method depends on whether SQL statements collected at the first stage are complete. The overhead caused by the matching algorithm is high. The basic idea of methods proposed in [21], [55] is that the web application needs to be run at both stages, and the expected SQL structure must be obtained by using benign inputs at the execution stage. These SQL structures depend entirely on the number of benign inputs, and the coverage rate is difficult to be guaranteed. SQLPsdem focuses on the entire web application source code through static analysis, and analyzes and discovers all complete SQL statements and complicated (joint or connected) SQL statements. Then, at the dynamic analysis stage, test cases of seven attack patterns are generated to trigger and detect SQL vulnerabilities. Compared with [21], [55], SQLPsdem's coverage is more representative and complete and more flexible to adapt to new vulnerabilities and attacks.

Son et al. [56] proposed a DIGLOSSIA tool based on taint tracking and dynamic analysis. The tool first obtains query models by mapping all query statements' characters to shadow characters but keeps the user's input value as its original value to identify the user input. Second, for SQL injection detection, it builds the actual query parse tree and the shadow query parse tree to validate whether the user input includes injection code. However, it has not considered the data source of the variables. Thomé et al. [58] utilized static analysis to extract the minimal program segments related to security from a web application and generate attack conditions. Subsequently, they applied hybrid-constraint solving to determine the satisfiability of attack conditions for vulnerability detection. However, this approach predominantly depends on static analysis, posing difficulties in identifying second-order SQL injections, where malicious payloads originate from application execution and data sources like sessions and cookies. In contrast, SQLPsdem offers greater flexibility and ease of extension to incorporate new attack patterns/rules, and with no risk of solver timeouts. Kemalis et al. [59] proposed a specification-based detection approach, which can neither discriminate the data source of the variables. Ray et al. [29] defined that only values (numeric and string literals) and reserved values (NULL, TRUE, etc.) are non-code, which covers only limited vulnerabilities. SQLPsdem coveres more SIPs and can distinguish first and second-order SQL injections based on seven types of attacks by categorizing the source of variables.

B. Comparison With Studies Focusing on Second-Order SQL Injection Detection

Studies focusing on second-order SQL injection detection are shown in **Table XIII**, where the "Detection Strategy" column lists the methods used for detecting SQL injections, the "Data sources covered" column refers to where the inputs come from, the "The type of SQL injection" column indicates whether each method can detect or prevent the seven types of SQL injection attacks listed in **Table I**, and the "Stored" column indicates whether the malicious input needs to be stored in the database in detection.

Approaches in [12], [13], [14], [15] detect second-order SQL vulnerabilities through attack testing. Yan et al. [12] highlight that "The principle of triggering the second-order injection is that parts of the malicious input is interpreted into the code and changes the semantics of the original SQL statement." However, they did not incorporate methods to automatically identify whether the malicious inputs alter the semantics of the original SQL statement. Based on their paper's evaluation section, it appears that the identification of semantic changes was conducted manually. Such manual steps are inadequate for identifying vulnerabilities in large and intricate web applications. In contrast, one of the primary innovative aspects of SQLPsdem is its automated verification of potential semantic changes in

TABLE XIII
THE COMPARISON RESULTS ON SECOND-ORDER SQL INJECTION VULNERABILITY DETECTION OR PREVENTION METHODS

Approach	Detection Strategy	Data Sources Covered				The Type of SQL Injection							Stored
		DB	Session	Cookie	File names	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	
Yan et al. [12]	Attack testing	✓	×	×	×	*	*	*	*	*	*	*	✓
Li et al. [13]	Attack testing	✓	×	×	×	✓	✓	✓	✓(B)	*	✓	*	✓
Draib et al. [14]	Attack testing	✓	×	×	×	*	*	*	*	*	*	*	✓
Liu et al. [15]	Attack testing	✓	×	×	×	*	*	*	*	*	*	*	✓
Tian et al. [16]	Structure matching	✓	×	×	×	✓	*	*	*	*	*	*	✓
Chen [17]	Structure and Syntax matching	✓	✓	×	×	✓	*	✓	*	*	*	*	×
Medeiros et al. [18]	Structure and Syntax matching	✓	×	×	×	✓	*	*	*	✓	*	✓	×
Dahse et al. [19]	Taint analysis-based sanitization	✓	✓	×	✓	*	*	*	*	*	*	*	×
Chen et al. [20]	Attack testing based on vulnerability paths	✓	✓	×	✓	*	*	*	*	*	*	*	×
SQLPsdem	Attack testing and rule-based sanitization	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×

τ_4:Inference (Blind injection on different conditions [B], Timing inference [T]); *: unknown; ✓: yes; x: no; DB: Database;
Stored: whether the malicious input needs to be stored in the database during detection.

SQL statements. It achieves this by initially distinguishing SQL tokens in the source code from those originating from other resources (utilizing Header tags and Inner tag), followed by applying a rule-based approach to scrutinize the remaining tokens. In addition, SQLPsdem outperforms [12], [13], [14], [15] because it is flexible and allows easy integration and removal of attack generators without modifying its code. We can also easily update the rules to accommodate new attack categories once they occur. Moreover, the SIPs in approaches [12], [13], [14], [15] all originated from databases. SQLPsdem expands its coverage to include additional potential injection sources, such as cookies, sessions, and file names. Storing malicious inputs in the database can bring security issues when testing, e.g., regression testing critical applications. Unlike [12], [13], [14], [15], SQLPsdem's detection and defense are performed before queries reach the database.

Tian et al. [16] proposed an improved parameterization defense model through blacklist filtering and structure matching based on SQL key words, table names and column names within user inputs. It cannot detect attacks that do not change the SQL query structure. Chen [17] proposed a second-order SQL injection attack detection method based on instruction set randomization (ISR), which first randomized trusted SQL keywords contained in the web application to construct a new SQL instruction set dynamically and then added a proxy server before DBMS to try to detect whether the received SQL instruction was an attack by matching the instruction. However, the approach cannot detect malicious user inputs that do not contain SQL keywords. SEPTIC [18] designed two matching models, i.e., Query Structure (QS) and Query Model (QM), to match the SQL structure and the SQL syntax extracted from static and dynamic stages to detect second-order SQL injections. It cannot detect attacks that make the number of nodes and the ordered elements identical in QM and QS. For a SQL query "*...WHERE login='α' AND pass=pass*", if the value *pass* is identical to the field-item *pass* and has not been verified as a number, it can result in a tautology attack. This attack does not alter the query structure or syntax, making it undetectable by the approaches [16], [17], [18]. In contrast, SQPsdem can detect the attack by $Rule_5$. The full coverage of the seven types of SQL injections allows SQLPsdem to accurately identify more attacks than approaches in [16], [17], [18], which rely solely on structure and syntax matching.

Dahse et al. [19] developed a persistent data stores (PDS) collector upon the vulnerability detection prototype (RIPS) to identify second-order injections. This is achieved through a static taint analysis of the data obtained from the PDS. However, this approach cannot detect path-sensitive sanitization of data written to the database, leading to a higher false positive rate when multiple email addresses stored in the database are regarded as tainted. SQLPsdem can address this issue by not filtering this type of inputs based on rules and considering it benign. As a result, SQLPsdem eliminates false positives associated with such cases. Chen et al. [20] proposed DISOV, which utilizes a web application property graph to capture the data propagation and inter-state dependencies. This representation aids in identifying potential paths of second-order vulnerabilities, which are validated through fuzz testing. However, there is a low efficiency and coverage when attempting to identify and trigger potential vulnerability paths. Additionally, the details of the fuzz testing are not provided, making the representativeness and completeness of attack cases questionable. Different from DISOV [20], SQLPsdem employs a more effective attack testing method that specifically targets SIPs and includes comprehensive analysis.

In summary, SQLPsdem can effectively detect a greater variety of SQL injection vulnerabilities. It covers more complete SIPs (constant, user input, non-user input such as database field data, file, Cookie and Session data) and uses four penetration testing tools, which complement each other, to generate comprehensive test cases designed for attack testing. SQLPsdem applies precise rules to cover the seven types of SQL injection attacks to ensure proper data sanitization before queries reach the database. These combined efforts allow SQLPsdem to accurately identify a higher number of attacks compared with approaches that solely rely on single attack testing [12], [13], [14], [15], [20], structure and syntax matching [16], [17], [18] or taint analysis-based sanitization [19]. Additionally, SQLPsdem provides defense and localization mechanisms, which are highly necessary for research and industry.

VI. DISCUSSION

A. Implications to Academic and Industry

SQLPsdem analyzes the essence of SQL injection vulnerabilities deeply to prevent attackers from changing the syntax

and semantics of SQL statements by systematically crafting malicious inputs containing special characters to steal or destroy database information.

Compared with prior techniques, SQLPsdem can detect and identify CIAO or BroNIE attacks on different conditions and detect more second-order SQL injection vulnerabilities. In the current design and implementation of SQLPsdem, the rules are tailored to precisely match the structure and semantics of the attack patterns of SQL injections. The rules can easily be adapted to reflect new types of attacks without needing to change the SQLPsdem implementation. SQLPsdem is modular and flexible, and allows practitioners to easily adapt equations (1), (2), (3) in SQLPsdem to fit different types of databases.

B. Threat to Validity

The four types of attacks (Inference, Alternate Encodings, Illegal/Logically Incorrect Queries and Stored Procedures) cannot be effectively classified by SQLPsdem in the detection process. This is because SQLPsdem matches the detection vulnerabilities by making rules. As these types of attacks can be arbitrary, it is difficult to construct the fixed rules to distinguish them. However, this will not influence the practical use of SQLPsdem because most practitioners are more interested in knowing whether the inputs are SQL injection attacks than noticing the detailed classification of these attacks. Moreover, during the dynamic analysis, guiding the generation of seven types of SQL injections require manual effort to copy information from web debugger tools and adapt the information into penetration tools.

In our evaluation, we used the known CVEs in **Table IV** to evaluate the false negative rate of SQLpsdem. However, the evaluation may not be precise because there are probably SQL injection vulnerabilities in web applications which have not been identified and reported as CVEs. Injecting a known number of vulnerabilities to test suites can be a strategy to measure the false negative rate of vulnerability detectors when the number of vulnerabilities is unknown [3], but it is not applicable to this study. We used real web applications to evaluate SQLpsdem. Even if we inserted a known number of vulnerabilities in them, we still do not know if SQLpsdem can identify all the unknown vulnerabilities that are not inserted.

When the values containing malicious characters are intended to be considered benign in some scenarios, SQLPsdem will report them as vulnerabilities, because these inputs will either result in syntax errors (e.g., names like "O'Reilly") or fail to produce the desired results (e.g., a username with two single quotes, "ad"min", which is benign but can cause a semantic error if executed in the database). However, in our evaluation, the four attack generators do not generate this type of test cases, so all the test cases are attacks, and SQLPsdem reports zero false positives.

VII. Conclusion

Existing studies that focus on detecting and preventing SQL injections still report high false positives and negatives, especially regarding second-order injections. This paper first analyzed seven types of SQL injections and proposed a proxy-based static and dynamic execution detection and prevention mechanism for second-order SQL injections (SQLPsdem). We evaluated SQLPsdem using 12 open-source web applications. The evaluation results show that SQLPsdem can defend 100% of all SQL injection vulnerabilities in our experiments. SQLPsdem detects 375 first-order SQL injection vulnerabilities and locates 26 second-order SQL injection vulnerabilities (including 13 newly discovered vulnerabilities) with fewer false positives and negligible overheads. Besides, it can locate and defend against identified vulnerabilities effectively. Considering that SQLPsdem is rule-based, it may have limited performance in detecting zero-day attacks. In future, we will try to combine this approach with machine learning and deep learning approaches to extend its capability to detect zero-day attacks effectively.

Appendix A

SQLPsdem tool is available on https://github.com/KLSEHB/SQLPsdem.

Acknowledgment

The authors are grateful to the valuable comments and suggestions of the reviewers.

References

[1] "Common weakness enumeration." The MITRE Corporation. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html
[2] "OWASP top 10." OWASP. [Online]. Available: https://owasp.org/Top10/
[3] B. Zhang, J. Li, J. Ren, and G. Huang, "Efficiency and effectiveness of web application vulnerability detection approaches: A review," *ACM Comput. Surv.*, vol. 54, no. 9, pp. 1–35, Oct. 2021, doi: 10.1145/3474553.
[4] G. Ollmann, "Second-order code injection attacks," *NGS Insight Secur. Res.*, 2004.
[5] W. G. Halfond and A. Orso, "AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks," in *Proc. 20th IEEE/ACM Int. Conf. Automated Softw. Eng.,* 2005, pp. 174–183.
[6] Y.-S. Jang and J.-Y. Choi, "Detecting SQL injection attacks using query result size," *Comput. Secur.*, vol. 44, pp. 104–118, Jul. 2014.
[7] M. S. Aliero and I. Ghani, "A component based SQL injection vulnerability detection tool," in *Proc. 9th Malaysian Softw. Eng. Conf. (MySEC),* Piscataway, NJ, USA: IEEE Press, 2015, pp. 224–229.
[8] M. Ceccato, C. D. Nguyen, D. Appelt, and L. C. Briand, "SOFIA: An automated security oracle for black-box testing of SQL-injection vulnerabilities," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE),* Piscataway, NJ, USA: IEEE Press, 2016, pp. 167–177.
[9] D. Kar, S. Panigrahi, and S. Sundararajan, "SQLiDDS: SQL injection detection using document similarity measure," *J. Comput. Secur.*, vol. 24, no. 4, pp. 507–539, 2016.
[10] P. Li et al., "Application of hidden Markov model in SQL injection detection," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2, Piscataway, NJ, USA: IEEE Press, 2017, pp. 578–583.
[11] L. Zhang, D. Zhang, C. Wang, J. Zhao, and Z. Zhang, "ART4SQLi: The ART of SQL injection vulnerability discovery," *IEEE Trans. Rel.*, vol. 68, no. 4, pp. 1470–1489, Dec. 2019.
[12] L. Yan, X. Li, R. Feng, Z. Feng, and J. Hu, "Detection method of the second-order SQL injection in web applications," in *Proc. Int. Workshop Structured Object-Oriented Formal Lang. Method*, Queenstown, New Zealand: Springer-Verlag, 2013, pp. 154–165.
[13] X. Li, W. Zhang, and L. Zheng, "Vulnerability detection using second-order SQL injection combining dynamic and static analysis," *J. Huaqiao Univ. (Natural Sci.)*, vol. 39, no. 4, pp. 600–605, 2018.

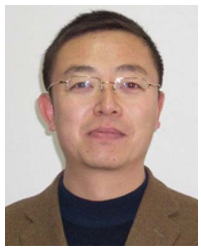
[14] N. Draib, A. Sultan, A. Ghani, and H. Zulzalil, "Security testing of web applications for detecting and exploiting second-order SQL injection vulnerabilities," *J. Eng. Appl. Sci.*, vol. 13, no. 20, pp. 8426–8431, 2018.
[15] M. Liu and B. Wang, "A web second-order vulnerabilities detection method," *IEEE Access*, vol. 6, pp. 70983–70988, 2018.
[16] Y. Tian, Z. Zhao, H. Zhang, and X. LI, "Second-order SQL injection attack defense model," *Netinfo Secur.*, vol. 15, no. 11, pp. 70–73, 2014.
[17] C. Ping, "A second-order SQL injection detection method," in *Proc. IEEE 2nd Inf. Technol., Netw., Electron. Automat. Control Conf. (ITNEC),* Piscataway, NJ, USA: IEEE Press, 2017, pp. 1792–1796.
[18] I. Medeiros, M. Beatriz, N. Neves, and M. Correia, "SEPTIC: Detecting injection attacks and vulnerabilities inside the DBMS," *IEEE Trans. Rel.*, vol. 68, no. 3, pp. 1168–1188, Sep. 2019.
[19] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *Proc. 23rd USENIX Secur. Symp. (USENIX Secur. 14),* 2014, pp. 989–1003.
[20] Y. Chen, Z. Pan, Y. Chen, and Y. Li, "DISOV: Discovering second-order vulnerabilities based on web application property graph," *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.*, vol. 106, no. 2, pp. 133–145, 2023.
[21] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, "SQLProb: A proxy-based architecture towards preventing SQL injection attacks," in *Proc. ACM Symp. Appl. Comput.,* New York, NY, USA: ACM, 2009, pp. 2054–2061.
[22] W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in *Proc. IEEE Int. Symp. Secure Softw. Eng.*, vol. 1, Piscataway, NJ, USA: IEEE Press, 2006, pp. 13–15.
[23] W. Halfond, A. Orso, and P. Manolios, "WASP: Protecting web applications using positive tainting and syntax-aware evaluation," *IEEE Trans. Softw. Eng.*, vol. 34, no. 1, pp. 65–81, Jan./Feb. 2008.
[24] J. Clarke, *SQL Injection Attacks and Defense*. Amsterdam, Netherlands: Elsevier, 2009.
[25] I. Lee, S. Jeong, S. Yeo, and J. Moon, "A novel method for SQL injection attack detection based on removing SQL query attribute values," *Math. Comput. Modelling*, vol. 55, nos. 1-2, pp. 58–68, 2012.
[26] Z. Djuric, "A black-box testing tool for detecting SQL injection vulnerabilities," in *Proc. 2nd Int. Conf. Inform. Appl. (ICIA),* Piscataway, NJ, USA: IEEE Press, 2013, pp. 216–221.
[27] N. F. A. Awang and A. Manaf, "Automated security testing framework for detecting SQL injection vulnerability in web application," vol. 534, no. 9, pp. 160–171, 2015.
[28] S. Jose, K. Priyadarshini, and K. Abirami, "An analysis of black-box web application vulnerability scanners in SQLi detection," in *Proc. Int. Conf. Soft Comput. Syst.,* India: Springer-Verlag, 2016, pp. 177–185.
[29] D. Ray and J. Ligatti, "Defining code-injection attacks," *ACM Sigplan Notices*, vol. 47, no. 1, pp. 179–190, 2012.
[30] D. Ray and J. Ligatti, "Defining injection attacks," in *Proc. Int. Conf. Inf. Secur.,* Cham, Switzerland: Springer-Verlag, 2014, pp. 425–441.
[31] "MySQL statement syntax." Hacker News. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/
[32] "PHP-parser." GitHub. [Online]. Available: https://github.com/nikic/PHP-Parser
[33] R. L. Rivest, "The MD5 message-digest algorithm," *RFC* , vol. 1321, pp. 1–21, Apr. 1992.
[34] "Automatic penetration test tool." GitHub. [Online]. Available: https://github.com/ming-shy/SQL-injection/tree/Penetration-testing-tools-1
[35] "CVE." The MITRE Corporation. [Online]. Available: http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=sql+injection
[36] "DVWA." SourceForge. [Online]. Available: https://sourceforge.net/projects/dvwa/
[37] "Pikachu, bWAPP, sqlilabs." GitHub. [Online]. Available: https://github.com/ming-shy/SQL-injection
[38] "Schoolmate." SourceForge. [Online]. Available: https://sourceforge.net/projects/schoolmate
[39] "Faqforge." SourceForge. [Online]. Available: https://sourceforge.net/projects/faqforge
[40] "Wackopicko." GitHub. [Online]. Available: https://github.com/amdoupe/Wackopicko.git
[41] "Webchess." SourceForge. [Online]. Available: https://sourceforge.net/projects/webchess
[42] "COVID-19 testing management system using PHP and MySQL." PHPGurukal. [Online]. Available: https://phpgurukul.com/covid19-testing-management-system-using-php-and-mysql
[43] "Doctormms." Sourcecodester. [Online]. Available: https://www.sourcecodester.com/php/14182/doctor-appointment-system.html
[44] "Pet shop." Sourcecodester. [Online]. Available: https://www.sourcecodester.com/php/14839/online-pet-shop-we-app-using-php-and-paypal-free-source-code.html
[45] J. Dahse and J. Schwenk, "RIPS-a static source code analyser for vulnerabilities in PHP scripts," in *Proc. Seminar Work (Seminer Calismasi), Horst Görtz Inst. Ruhr-Univ. Bochum*, 2010, pp. 1–12.
[46] "PHPvulhunter." GitHub. [Online]. Available: https://github.com/OneSourceCat/phpvulhunter
[47] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Proc. IEEE 31st Int. Conf. Softw. Eng.,* Piscataway, NJ, USA: IEEE Press, 2009, pp. 199–209.
[48] "Acunetix web vulnerability scanner." Acunetix. [Online]. Available: https://blackhat8.blogspot.com/2020/01/acunetix-web-vulnerability-scanner.html
[49] "2,300+ companies of all sizes automate application security testing with Acunetix." Acunetix. [Online]. Available: https://www.acunetix.com/
[50] N. Chen and Z. Huang, *Software Testing Guide: Fundamentals, Tools and Practice*. Beijing, China: People's Post and Telecommunications Press, 2011.
[51] W. G. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proc. 14th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2006, pp. 175–185.
[52] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *Proc. Secur. Privacy Age Ubiquitous Comput.*, R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, Eds., Springer-Verlag, 2005, pp. 295–307.
[53] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *Proc. 15th Conf. USENIX Secur. Symp.*, vol. 15, 2006, p. 9.
[54] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," *ACM Sigplan Notices*, vol. 41, no. 1, pp. 372–382, 2006.
[55] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. Venkatakrishnan, "CANDID: Preventing SQL injection attacks using dynamic candidate evaluations," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 12–24.
[56] S. Son, K. S. McKinley, and V. Shmatikov, "DIGLOSSIA: Detecting code injection attacks with precision and efficiency," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.,* New York, NY, USA: ACM, 2013, pp. 1181–1192.
[57] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proc. 13th Int. Conf. World Wide Web.*, 2004, pp. 40–52.
[58] T. Julian, S. L. Khin, B. Domenico, and B. Lionel, "An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving," *IEEE Trans. Softw. Eng.*, vol. 46, no. 2, pp. 163–195, Feb. 2020.
[59] K. Kemalis and T. Tzouramanis, "SQL-IDS: A specification-based approach for SQL-injection detection," in *Proc. ACM Symp. Appl. Comput.*, 2008, pp. 2153–2158.

Bing Zhang received the bachelor's degree from the College of Computer and Information Technology, Three Gorges University, China, in 2012, and the Ph.D. degree from the School of Information Science and Engineering, Yanshan University, China, in 2018. He is currently an Associate Professor with the School of Information Science and Engineering, Yanshan University. His research interests include data mining, machine learning, and software security.

Rong Ren received the bachelor's degree from the College of Computer and Information Technology, Three Gorges University, China, in 2012, and the M.S. degree from the School of Information Science and Engineering, Yanshan University, China, in 2015. She is currently working toward the Ph.D. degree with the School of Information Science and Engineering, Yanshan University. Her research interests include data mining, machine learning, and software security.

Jia Liu received the bachelor's degree from Hebei North University, China. He is currently working toward the master's degree with the School of Information Science and Engineering, Yanshan University. He is currently focusing on a software security project. His research interests include web application security and machine learning.

Mingcai Jiang received the bachelor's degree from Wuhan Institute of Technology, China. He is currently working toward the master's degree with the School of Information Science and Engineering, Yanshan University. He is currently focusing on a software security project. His research interests include recommendation systems and machine learning.



Jiadong Ren received the B.S. and M.S. degrees from the Northeast Heavy Machinery Institute, in 1989 and 1994, respectively, and the Ph.D. degree from Harbin Institute of Technology, in 1999. He is a Professor with the School of Information Science and Engineering, Yanshan University, China. His research interests include data mining, temporal data modeling, and software security. His research has been supported by the National Natural Science Foundation of China and Science Foundation of Hebei Province. He is a Senior Member of the Chinese Computer Society and a member of ACM.

Jingyue Li (Senior Member, IEEE) received the Ph.D. degree in software engineering from the Department of Computer Science, NTNU, in 2006. He is a Professor with the Computer Science Department, Norwegian University of Science and Technology (NTNU). His research interests include software engine, software security, and blockchain technologies.