DOI: https://doi.org/10.52756/ijerr.2024.v42.001 *Int. J. Exp. Res. Rev.*, Vol. 42: 01-17 (2024)

Original Article | *Peer Reviewed* | *Open Access*

International Journal of Experimental Research and Review (IJERR)

ISSN: 2455-4855 (Online)
www.iaph.in
ISSN 2455-4855

SQL Injection Attack Detection Based on Similarity Matching Between Vectors Extracted From Design Time and Run-Time Queries

Jayanto Kumar Chowdhury[1]*, Dilip Kumar Yadav[1] and Chandra Mouli P.V.S.S.R[2]

[1]Department of Computer Science and Engineering, National Institute of Technology, Jamshedpur-831014, Jharkhand, India; [2]Department of Computer Science, Central University of Tamil Nadu, Thiruvarur-610005, Tamil Nadu, India

E-mail/Orcid Id:

JKC, jayantochowdhury@gmail.com, https://orcid.org/0009-0002-1570-1873;
DKY, dkyadav.cse@nitjsr.ac.in, https://orcid.org/0000-0002-1334-7500; *CM,* chandramouli@cutn.ac.in, https://orcid.org/0000-0001-7909-9733

Article History:

Received: 20th Mar, 2024
Accepted: 27th Jul., 2024
Published: 30th Aug, 2024

Keywords:

Database security, dynamic method, security vulnerabilities, SQL injection, web application

How to cite this Article:

Jayanto Kumar Chowdhury, Dilip Kumar Yadav and Chandra Mouli P.V.S.S.R (2024). SQL Injection Attack Detection Based on Similarity Matching Between Vectors Extracted From Design Time and Run-Time Queries. *International Journal of Experimental Research and Review*, *42*, 01-17.
DOI:
https://doi.org/10.52756/ijerr.2024.v42.001

Abstract: Everyone uses web-based applications to carry out daily business and personal tasks. These programmes are vulnerable to attack by hackers, who may also misuse the data. The most serious attack with the greatest damaging potential on digital platforms is the structured query language injection attack (SQLiA). The backend databases could be corrupted or destroyed by SQLiA if it manages to breach security protections. Using SQLiA tactics, hackers can get unauthorized access, steal important data, and take over the network completely or partially. An automatic SQL injection prevention and detection technique is needed to safeguard web-based applications from SQLiA. This research suggests a novel similarity-matching algorithm of vectors extracted from design time and run-time queries. This technique allocates the weights of different SQL keywords used in design time and run-time queries and further design time and run-time vectors have been created from respective queries. The similarity between the design time and run time vector is determined by calculating the angle between these two vectors. The angle of deviation between the design time vector and run time vector is calculated and if the angle of deviation is zero, then it is concluded as no SQL injection otherwise, it indicates the existence of SQLiA vulnerability. The proposed algorithm is validated against the GitHub dataset. In the first dataset, out of 1300 injected queries, the proposed method identifies 1219 injected queries; out of 300 normal queries, it identifies 290 normal queries with 93.76% and 96.66% detection accuracy, respectively. Similarly, for the second dataset, out of 10489 injected queries, it identifies 10280 injected queries and out of 301 normal queries, it identifies 280 normal queries with 98.01% and 93.02% detection accuracy, respectively.

Introduction

In the digitization world, web applications are mostly used to perform day-to-day activities in e-commerce, banking, healthcare etc. Many users use web applications to perform their tasks and share their valuable personal and business information over the web. However, there are security loopholes in the web applications. In most cases, the cyber-attacks are performed by expert hackers using various techniques like Injection attacks, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Broken authentication and Denial of service (DoS) etc. These are the general intrusion detection techniques commonly used by hackers. Among the various attacks, SQL injection attacks are among the most dangerous threats to web applications. They have been listed among the top ten vulnerability attacks by OWASP (Open Web Application Security Project), an international organization for web application developers. SQLiAs are critical threats to organizations as well as military and defence systems. Due to a lack of protective systems, SQLiA attacks can potentially damage underlying databases, steal valuable information and compromise

***Corresponding Author:** jayantochowdhury@gmail.com

individual machines or the entire network. Hackers can get unauthorized access to web applications and underlying databases, steal sensitive information and corrupt the databases. Some reputed organizations impacted by SQLiAs are Travelocity, FTD.com, Creditcards.com, Guess Inc. and RIAA.

In most cases, hackers target the free text input fields present in the web applications to exploit the web applications and underlying databases. Such text input fields without proper validations become security threats or loopholes of any web application. In such cases, expert hackers use their techniques to enter malicious inputs in free text input fields to exploit the underlying databases. Hence, every text input should be thoroughly validated to avoid SQL injection attacks. Developers may use input validation and parameterized queries to prevent SQL injection attacks.

In legacy web applications, developers use string concatenation techniques to send the actual user inputs during the use of web applications which is bad programming practice and always provides a path for hackers to exploit the web applications. In many cases, modern developers still use the string concatenation technique to pass the user inputs to the web applications and are prone to SQL injection attacks. To overcome these situations, programmers should use stored procedures and parameterized queries to pass input parameters to SQL queries.

The web applications, at times, may display the actual database-level errors in the web pages. It is due to inappropriate exception handling, the presence of syntax errors logical errors etc. Hackers exploit this technique to gather the initial database schema, table details and role authorization. It is the entry point for hackers to gather valuable information about the underlying databases. Based on this analysis, hackers gain initial knowledge and further exploit their actions to attack the underlying databases.

In the case of developed legacy applications or applications deployed in a production environment, source code modification is essential to protect those applications from SQL injection attacks. However, it is a challenging and time-consuming task to re-engineer any production application. This is also very difficult to modify the production application in different locations of one or many source code files of the application to protect from SQL injection attacks. To overcome such situations, we require an automatic tool that acts as a shield to protect systems from SQL injection attacks in database-driven applications. Such a shield may perform as a filter to separate the SQL-injected queries and safeguard the applications. Informal survey results show that 97% of such free text input fields are potentially vulnerable to SQL injection attacks. However, tool development includes the process of continuous improvement to keep up with emerging threats and black-hat techniques, and it is a tedious task. It is possible that a single tool may not address all types of SQLiA attacks (Thomas et al., 2009; Abdul Bashah Mat Ali et al., 2011; Dimitris et al., 2009; Lee et al., 2012; Huang et al., 2003). Researchers always try to address critical or high-impact vulnerabilities like SQL Injection. As per the study, 70% of database-centric applications are under threat due to SQL injection vulnerabilities.

Based on the literature survey, SQLiA detection and prevention methods can be categorized as follows:

Detection of SQLiA using Static analysis approach

Identification of SQLiA using Dynamic analysis method

Combination of static and dynamic approaches (Ghafarian, 2017)

Machine learning techniques

The static analysis approach analyses the whole SQL query that exists in the database-driven application. It verifies the user input type to prevent SQL injection attacks. It is difficult to prevent if malicious user input contains the correct input type. This technique does not apply to emerging and new types of SQLiA attacks. Dynamic analysis techniques are employed to detect security loopholes during program executions. Example: CANDID tool techniques. Based on the literature available, the dynamic approach seems to be the best fit for web applications (Thomas et al., 2009; Elia et al., 2010; Park et al., 2006). Vulnerability Assessment and Penetration Testing (VAPT) tool may be used to detect the loopholes, vulnerable codes etc., in existing web applications so that developers can fix those areas to strengthen the security issues before exposure to the external world.

The advantages of dynamic or penetration testing are as follows:

No impact on the development of the life cycle

Avoidance of static analysis challenges

Source code sanitizing is not necessary

Deployment-security

E-shield (Jamar et al., 2017) devices or honeypots are currently in use to protect the production system from SQLiA attacks. A code-based analysis to automatically detect the existing SQLiA attacks (Su et al., 2006; Halfond et al., 2008). Recently, machine learning techniques have been prevalent in SQLiA detection. Example: Wave accessibility evaluation tool (WAVES

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

tool) (Thomas et al., 2009; Zhang et al., 2010; Natarajan et al., 2012; Jana et al., 2020; Hlaing et al., 2020). The combined static and dynamic analysis method is more robust because it can use the functionality of both. A model is prepared based on various machine learning algorithms in machine learning techniques. Then, the dataset is divided into two sets training and test set. The training set is for training the model, and the test set is for validation of the model. This hybrid approach can detect SQL injection attacks.

Related Works

To incorporate a thorough examination of SQL injection, we have evaluated papers from several journals, conferences, and data sources. The following is the organization of various papers:

Huang and others focus on web application vulnerability assessment to find the loopholes and coding practices (Huang et al., 2003) prone to SQL injection attacks. This paper focuses on analyzing the design of web applications to identify poor coding practices, use of software testing tools etc to expose the SQLiA vulnerabilities. This study helps to identify the existing gaps of web applications and those vulnerabilities can be fixed further to avoid the SQLiA attacks. But for real-time applications, this tool cannot detect such SQLiA attacks that did not surface during source code analysis. Wassermann and others describe a technique of input validation approach using static analysis (Wassermann et al., 2004) method to detect and prevent SQLiA vulnerability. Similarly, the JDBC checker (Su et al., 2004) is also using a static analysis approach. Nguyen and all present a fully automated approach to securely hardening (Nguyen-Tuong et al., 2005) web applications.

Protection measures to reduce vulnerabilities are crucial: for instance, in the case of Android malware detection, new approaches like the Borutashap algorithm turned out to be effective (Sharma et al., 2023). This approach puts emphasis on the array needed to protect digital systems whether from malware or injection attacks. Buehrer and others describe a technique to detect the manipulation performed by hackers in SQL queries. The technique is based on comparing, at run time, the parse tree (Buehrer et al., 2005) of the SQL statement before inclusion of user input with that resulting after inclusion of input. The parse tree comparison is very efficient and adds about 3 milliseconds overhead to database query execution time. It is easy to implement as developers need minimal effort to change the source code section of database interaction.

Valeur and others focused on an anomaly-based (Valeur et al., 2005) system which learns the normal database access by web applications using different models. These models protect the underlying database from unknown attacks. Many researchers also focus on dynamic taint analysis to detect SQLiA vulnerabilities. Halfond and others describe the AMNESIA tool (Halfond et al., 2005) which is based on static and dynamic analysis approaches to detect SQL injection vulnerabilities. This tool builds a model based on static analysis to generate legitimate queries. Park and others present a detection methodology SQL injection using pairwise (Park et al., 2006) sequence alignment of amino acid code formulated from a web application parameter database sent via the web server. The experiment shows that this method can identify existing SQLiA vulnerabilities and unknown attacks.

Wasserman and Su use the static analysis technique to generate finite state automata (Wasserman et al., 2006) for modelling the set of valid SQL commands for each set of data access. They present the first formal definition of command injection attacks in the context of web applications and gives a sound and complete algorithm for preventing them based on context-free grammar and compiler parsing techniques. It cannot handle many queries, such as those with LIKE. This limitation is an implementation issue, and it is reasonable to assume that support for these yet unsupported queries will be available in the future. However, it is a problem of static design and it cannot model dynamic queries. The technique to prevent tautology attacks is very complicated.

Halfond and others focus on a highly automated method to protect applications against SQL injection. It is based on the novel idea of positive tainting (Halfond et al., 2008) and the concept of syntax-aware evaluation. Thomas and others suggested an algorithm of prepared statement replacement to avoid SQLiA vulnerabilities by replacing SQL statements with prepared statements (Thomas et al., 2009). Prepared statements have a static structure, which prevents SQL injection attacks from changing the logical structure of a prepared statement.

Mitropoulos proposes a novel method to prevent SQLiA attacks by placing a database driver (Dimitris et al., 2009) between the application and underlying databases. It creates SQL signatures that are used to distinguish between normal queries and injected queries. The driver neither depends on the web application nor the underlying databases and due to this reason, it can be integrated easily with any web application/system. It acts

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

as a shield between web applications and the backend databases to protect from SQL injection attacks.

Elia and others described the experimental evaluation of five detection tools (Elia et al., 2010) concerning vulnerabilities that exist in applications, databases and networks. The results emphasize the shortcomings of current intrusion detection technologies in identifying SQL Injection attacks as the analyzed tools have relatively poor effectiveness and only perform effectively in certain situations. Web applications employing database-driven content have become widely deployed on the Internet, and organizations use them to provide a broad range of services to people. Along with their growing deployment, there has been a surge in attacks that target these applications. One type of attack, particularly SQL injection, is especially harmful. SQL injections can give attackers direct access to the database underlying an application and allow them to leak confidential or even sensitive information. SQL injection can evade or detour IDS or firewalls in various ways. Hence, a detection system based on regular expressions or predefined signatures cannot prevent SQL injection effectively. Zhang and others described a tool D-WAV (Zhang et al., 2010) to detect cross-site scripting and SQL injection vulnerabilities.

Ali et al. (2011) shared the idea of a new web scanning tool (MySQL injector) with enhanced features that can perform penetration testing on PHP applications. This tool generates the result of penetration testing of any web application. After analysis of the result, injection techniques and new hacking techniques can be captured. Cyber experts can refer to these existing and new hacking techniques to gather knowledge of protection mechanisms from SQLiA vulnerabilities.

Lee and others suggested a very simple and effective way to detect the SQLiA vulnerabilities where it removes the SQL query attributes (Lee et al., 2012) of web applications or web pages during page submission and compares the parameters with the pre-determined ones. It uses a combination of static and dynamic analysis approach and experiments show its effectiveness and simplicity. To detect and prevent SQL injection attacks, Natrajan and others suggested a SQL-injection-free (SQL-IF) secure algorithm (Natrajan et al., 2012). The generated algorithm can be integrated into the runtime environment while the implementation has been done through Java. This method also describes several procedures to avoid SQL injection attacks.

Ghafarian has developed a novel method for identifying and preventing SQLIA implementation. The methodology is a hybrid (Ghafarian, 2017) of the static and dynamic approaches. There are three steps to the suggested method. It is advised that all database tables be expanded to include a record with only a few images, such as a dollar sign, for the initial stage (static). This needs to be completed before implementation and during database design. The author suggested creating an algorithm once and configuring it to work for any query for the second step (dynamic).

Jana and others focus on a code-based analysis (Jana et al., 2020; Kumar et al., 2023) approach to detect injection attacks in a query before execution. This approach analyses the user input by assigning a complex number to each input element. Hlaing et al. (2020) present an approach that detects a query token with a reserved words-based lexicon (Hlaing et al., 2020) to detect SQLIA attacks. At first, it creates a lexicon and in the second step tokenizes the input query statement and each string token is detected to a predefined words lexicon to prevent SQLiA. Shreya and others depicted the existing tools and methods available to detect and prevent SQLiA vulnerabilities (Chowdhury et al., 2021). Gogoi et al. (2022) suggested a machine learning-based approach for the detection of web shells written in PHP language. The proposed approach analyses the function call and the use of super global variables commonly used in PHP web shells using a deep learning technique. Saxena et al. (2022) described web security flaws like SQLi, XSS, malicious URLs, phishing attacks, path traversal and CMDi in detail. They also elaborated on the existing security methods for detecting these threats using machine learning approaches for URL classification and the potential research opportunities for ML and DL-based techniques in this category, based on a thorough examination of existing solutions.

SQLi Attack Types

This section emphasizes the most common types of SQL injection attacks used by hackers. The main intention is to summarize the different types of SQL injection attacks to manipulate the data, gather the database information, access underlying databases and execute system-level commands to destroy the databases. Table 1 summarizes some of the SQLiA types.

Table 1. SQLi Attack Types.

Sl. No.	Type	Purpose
1	Illegal or logically In-correct	Reveal relevant database information through error messages generated from the underlying database
2	Piggybacked	To delete the information from a database with harmful intention
3	Tautology	To get the application access

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

		without a valid username and password
4	Union	To disclose sensitive information using the UNION operator
5	Stored Procedure	To gain access to the host operating system by performing a command execution
6	Alternate encoding	To hide the aggressor's pattern via alternate encodings, such as hexadecimal, ASCII
7	Boolean Injection or Inference	To bypass the authentication mechanism to gain access to database information

I. Illegal / logically incorrect queries: It is a primitive way to gather database information applied by hackers or adversaries. In this case, an adversary injects junk inputs into the regular queries. Due to this, the underlying database throws error messages containing the database schema information and reveals other related database details. Based on this initial information, hackers may further exploit the database using different types of SQLiA. The purpose of this attack is to collect the structure of the database schema.

SELECT * FROM employee WHERE employeeId = 'mec001199' AND password = 'abc@123'AND CONVERT (char, no);

II. Piggybacked queries: A malicious SQL query is inserted into a normal SQL query. In structured query language, the database may execute multiple SQL queries simultaneously if the operator separates the queries ";". Note that this operator is inserted at the end of each query. Using this attack the hacker can drop tables gather table data or even destroy the database. On execution of Query 2, the table user will be dropped.

SELECT * FROM employee WHERE employee Id = 'mes001199'AND password 'abc@12'; DROP TABLE user;

III. Tautology attacks (Qbea'h et al., 2016) bypass the authentication mechanism to access the database information. The SQL injection query evaluates as TRUE by adding one or more SQL commands like (2 = 2 or 1 =1 or –) in the WHERE clause of the SQL statement. The results of Query 3 will display all the information in the user table.

SELECT * FROM users WHERE name='abcd' OR 1 = 1;

IV. Union queries: The union injection attacks performed by the hackers to join the two SQL queries using the UNION operator. The malicious query joined with the normal query with the UNION operator.

SELECT * FROM user WHERE id= 'mes01199 'UNION SELECT * FROM previlege WHERE id='admin'–"AND password= 'abc1234';

All the strings after "—" are considered as comments and two SQL queries are executed. The result of the query process shows the administrator's information on the DBMS.

V. Stored Procedure: The attacker uses built-in stored procedures and executes built-in functions with malicious SQL injection codes.

CREATE PROCEDURE DBO @userName varchar2, @pass varchar2, AS EXEC("SELECT * FROM user WHERE id="'+ @userName + "'AND password="'+ @password + "'); GO

This scheme is very vulnerable to attacks such as piggy-backed queries.

VI. Alternate encoding: The attackers modify the normal query using alternate encoding such as hexadecimal, ASCII and Unicode to avoid detection by the defensive techniques.

SELECT accounts FROM users WHERE username = 'john'; exec (char (Ox73687574646j776e))

VII. Boolean Injection / Inference: It is a type of inferential SQL injection technique where the hackers inject malicious payload that forces the database to return a different result set depending on whether the query returns a TRUE or FALSE result.

SELECT accounts FROM "users" WHERE username = 'john'OR 1 = 1

Materials and Methods

The proposed system is based on a dynamic analysis approach and can detect SQL injection attacks in real-time scenarios. It detects all the SQLiA vulnerabilities listed in Table 1. The proposed system involves the following steps:

A. Weight Assignment

B. Extraction of vectors from design time and run-time queries

C. SQL injection vulnerability detection Engine

A. Weight Assignment

Each SQL keyword has some impact on SQLiA vulnerabilities. Assignment of a Weight to each SQL keyword is done depending on its severity. A weight of 5 indicates the highest severity and a weight of 1 indicates the low severity. The weight has been assigned for each SQL keyword to identify the severity of the presence of the keyword in SQL injection queries. The SQL keywords are then stored as key-value pairs in a dynamic data table termed as dictionary. The procedure to create the dictionary is defined in Algorithm 1. The key-value pair can be visualized as in Table 2.

Algorithm 1. Prepare SQL Keyword HashTable().

Pseudo Code:
HashTable prepare_SQL_Keyword_HashTable()
{
Step 1: Create a hashtable
Step 2: Store all the SQL keywords as key-value (weight) pairs in this hashtable where value refers to the weight assigned.
}
Step 3: Return the hashtable

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

Table 2. List of keywords having their weights two or above.

Keyword (key)	Weight (value)	Keyword (key)	Weight (value)
ALTER	5	MODIFY	3
CONNECT	5	NOAUDIT	3
CREATE	5	OR	3
DROP	5	SELECT	3
SHUTDOWN	4	UNION	3
ADD	3	AUDIT	2
ALL	3	NOCOMPRESS	2
AND	3	NOT	2
BETWEEN	3	NOWAIT	2
DELETE	3	NULL	2
INSERT	3	ORDER BY	2
MODIFY	3	UPDATE	2

Table 2 shows a few keywords along with their weights. The rest of the keywords are assigned the weight 1. A lookup table is required to store all the SQL keywords along with the corresponding weights. To accomplish this, a dictionary has to be created using a hash table. The hash table returned by the Algorithm1 is used for this purpose. It stores all the SQL keywords as key-weight pairs. It is used for extracting the weights for the available SQL keywords in the given design time run-time query.

B. Extraction of vectors from design time and run-time queries

The different vectors extracted from design time and run-time queries are as follows:

i. In the case of real-time application, the DLL (dynamic link library) extractor is plugged in with the application. The exposed methods of the extractor are called before the database interaction source code snippet. It captures the design time query and sends it to the dynamic data table to store the design time query temporarily.

ii. From the same source code snippet, the exposed method of the extractor captures the run-time query which consists of all the input parameters and sends to them to the dynamic data table corresponding to the same design time query.

iii. For Example, consider the following design time and run-time query extracted from the query extractor.

Design Time query: SELECT * FROM employee WHERE emp_name = @val1;

Run time query (Injected query): SELECT * FROM employee WHERE emp_name = 'lucia01' OR 1 >0;

iv. The split function of the string is used to separate the SQL Keywords that exist in design time and run-time query and are stored in a temporary location. With the help of Table 2, the Weight Vector is created for respective queries after assigning the weights. The *get_Keyword_Count ()* function of Algorithm 2 is used to create the count vector. Similarly, the Keyword Vector is created. After all the above steps, the Design Time Vector(dv) is created which is the product of the design time weight and count vector. Similar steps are followed to create the Run Time Vector(rv), a product of run time weight and count vector. The resultant Design Time Vector (dv) and Run Time Vector(rv) with associated vectors are shown below:

Keyword Vector (Design Time): $[select, from, where]^T$
Count Vector (Design Time): $[1, 1, 1]^T$
Weight vector (Design Time): $[3, 1, 1]^T$
Design time vector(dv) = Count Vector * Weight Vector = $[3, 1, 1]^T$

Keyword Vector (Run Time): $[select, from, where, or]^T$
Count Vector (Run Time): $[1, 1, 1, 1]^T$
Weight vector (Run Time): $[3, 1, 1, 3]^T$
Run time vector(rv) = Count Vector * Weight Vector = $[3, 1, 1, 3]^T$

The above design time and run time vectors are used in subsection C for further processing.

C. SQL injection vulnerability detection engine

Figure 1 shows the system architecture to detect SQLiA. A user requests web pages via HTTP web request from a web server. Web pages are submitted to the webserver to serve the user request. A web server uses HTTP (Hypertext Transfer Protocol) and a few other protocols to process the web requests generated from the client side over the World Wide Web. The web server is connected to the internet to serve incoming and outgoing web requests through a firewall. The underlying Web server software is responsible for processing the web request and generating the response to end users or clients. It can handle multiple requests simultaneously.

The design time query and run-time query can be discriminated as follows: Assume that a user has to submit his username and password in a login form. The user enters the text fields username and password and presses the submit button.

The code for the submit button might be written as

SELECT * FROM login WHERE user name = @username AND password = @password. This is termed a design time query.

Assume that the username of the user is "user1" and the password is "pass1234". The user provides this information and presses the submit button. Then the code for the submit button is transformed as

SELECT * FROM login WHERE user name = 'user1' AND password = 'pass1234'.

This is termed a run-time query. The query extractor extracts design time and run-time queries and stores both SQL queries in system memory.

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

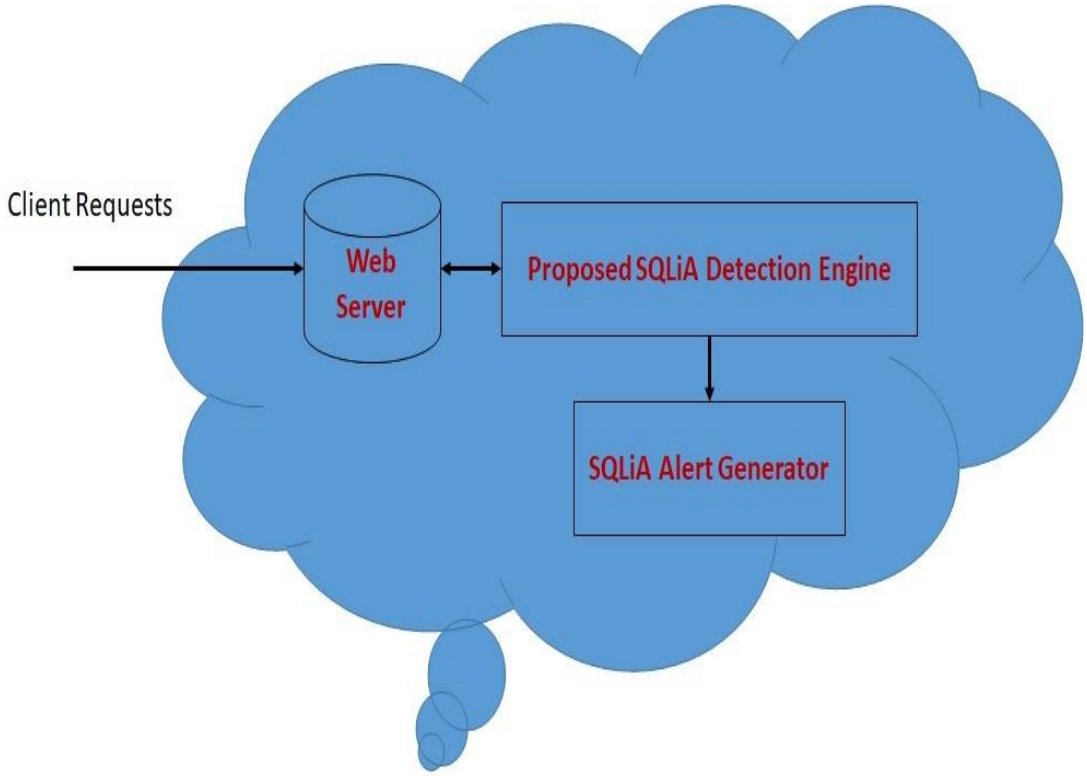

Figure 1. Skeleton view of the proposed method and its location of operation.

The illustration of the proposed method and generic steps involved in processing the design time and run-time queries are as mentioned below:

I. Consider Table 3 for sample design time and run-time queries. The proposed engine tokenizes each word in the design time query and forms a keyword vector.

II. A function *get_Keyword_Count(string, vector)* is defined in Algorithm 2 to count the occurrences of each SQL keyword and store it as a count vector. From the repository of weights i.e., from Table 2, the weight for each keyword is extracted and stored as a weight vector.

III. Create a keyword vector, weight vector, count vector and design time vector (dv) as in Table 4 from the given design time queries of Table 3. Table 4 shows the step-by-step extraction of the keyword vector, weight vector, count vector and finally, design time vector i.e., dv. Here, the weight and count of the keywords are extracted and stored in the weight and count vector. The weight vector and count vector's dot product are taken and stored as dv. The count vector is calculated with the help of Algorithm 2.

IV. Create a keyword vector, weight vector, count vector and run time vector (rv) as in Table 5 from the given run time queries of Table 3.

Algorithm 2: get_Keyword_Count(string query, vector keyword vector)

```
Pseudo Code:
vector get_Keyword_Count(string query, vector
keyword_vector)
{
Step 0: count_vector = []; count=0
Step 1: For each element i in keyword_vector
Step 2: count the occurrences in the query
Step 3: count_vector[i]=count
Step 4: count = 0
Step 5: Return count_vector
 }
```

Table 5 shows the keyword vector, weight vector, count vector extracted for the run-time queries and finally time vector i.e., rv. Here the weight and count of the keywords are extracted and stored in the weight and count vector. The dot product of the weight vector and count vector is taken and stored as rv. The count vector is calculated with the help of Algorithm 2.

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

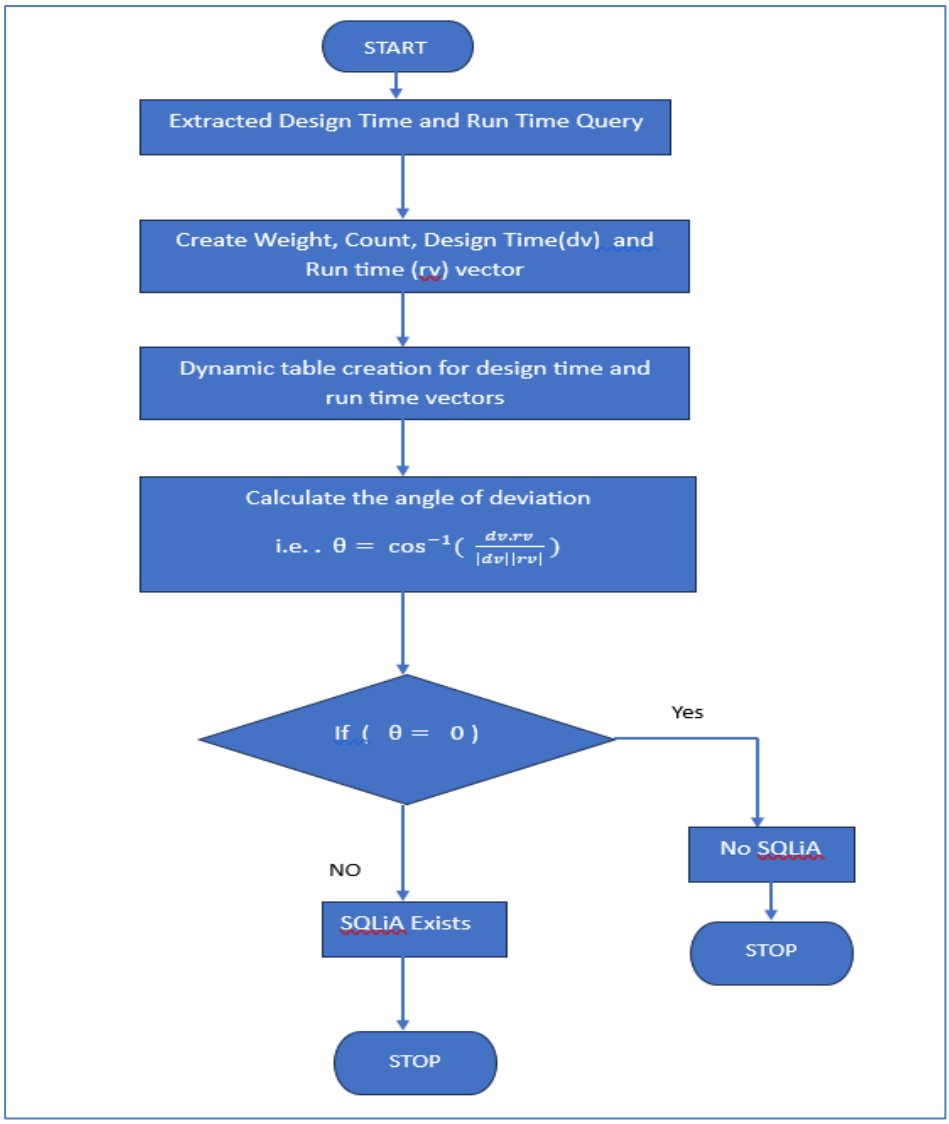

Figure 2. Flow chart of the proposed method.

VI. To find the similarity between dv and rv, they should be of equal length. If required, the vector dv is padded with zeros to maintain the same length as that of rv. The similarity between the dv and rv is determined by determining the angle between two vectors. This can be determined using the equation (1) formula.

$$\text{i.e. } \theta = \cos^{-1}\left(\frac{dv.rv}{|dv||rv|}\right) \qquad \text{----------(1)}$$

Here The mathematical formula $\text{i.e., } \theta = \cos^{-1}\left(\frac{dv.rv}{|dv||rv|}\right)$ is used to measure the angle of deviation between design time and run time vector. If the angle of deviation is zero, then it implies that there is no SQLiA vulnerability and if the angle of deviation is zero, then it means that SQLiA vulnerability exists.

Table 6 shows the angle between the design time and run-time queries. In Table 6, the angle of deviation θ is non-zero for all the queries i.e., from Q1 to Q10 except Q5. In the case of Q5, θ is zero, meaning there is no SQL injection vulnerability. But for others i.e., from Q1 to Q10 (except Q5), SQLiA vulnerability exists. The above steps are also depicted in the below flowchart.

In this proposed method, the similarity matching is calculated based on the angle of deviation. If the angle between the design time and run-time query is 0 degrees, it means there is no deviation and it implies the design time and run-time queries are similar. This indicates that there are no SQL injection vulnerabilities. On the other hand, if the angle between design time and runtime query is a non-zero degree, there is some deviation that indicates that design time and run-time queries are not similar. That means SQL injection vulnerabilities exist in this scenario. This mathematical model to determine the SQL injection vulnerabilities is unique because till now, no researcher used this concept for this purpose. This model can detect all kinds of SQL injection vulnerabilities. This model can be used in real-time scenarios and is able to prevent the web application from SQL injection vulnerabilities.

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

Table 3. Sample design time and run-time queries.

Query No.	Design time Query	Run time Query
Q1	SELECT * FROM employee WHERE emp_name = @val1	SELECT * FROM employee WHERE emp_name = 'lucia01' OR 1 >0
Q2	SELECT * FROM employee WHERE emp_name = @val1	SELECT * FROM employee WHERE emp_name = 'lucia01'; UNION all SELECT * FROM employee
Q3	SELECT * FROM employee WHERE emp_name = @val1	SELECT * FROM employee WHERE emp_name = 'lucia01'; EXEC SP_Help
Q4	SELECT * FROM employee WHERE emp_name = @val1	SELECT * FROM employee WHERE emp_name = 'lucia01'; DROP TABLE user
Q5	SELECT * FROM employee WHERE emp_name = @val1	SELECT * FROM employee WHERE emp_name = 'lucia01'
Q6	SELECT * FROM employee WHERE emp_name = @val1	SELECT * FROM employee WHERE emp_name = 'lucia01'; SHUTDOWN
Q7	SELECT * FROM employee WHERE emp_name = @val1	SELECT * FROM employee WHERE emp_name = 'lucia01' WAITFOR DELAY '00:00:10:00'
Q8	SELECT * FROM employee WHERE emp_name = @val1	SELECT * FROM employee WHERE emp_name = 'lucia01' UNION SELECT * FROM employee
Q9	SELECT * FROM employee WHERE emp_name = @val1	SELECT * FROM employee WHERE emp_name= 'lucia01' OR 1 = COVERT(int,(SELECT TOP 1 table name FROM INFORMATIONSCHEMA.tables))
Q10	SELECT * FROM employee WHERE emp_name = @val1	SELECT * FROM employee WHERE emp_name = 'lucia01'; SELECT DB_NAME()

Table 4. Keyword vector, Weight vector, Count vector and design time vector for design-time queries are given in Table 3.

Query	Keyword vector	Weight vector	Count vector	Design time vector(dv)
Q1	$[select, from, where]^T$	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$
Q2	$[select, from, where]^T$	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$
Q3	$[select, from, where]^T$	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$
Q4	$[select, from, where]^T$	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$
Q5	$[select, from, where]^T$	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$
Q6	$[select, from, where]^T$	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$
Q7	$[select, from, where]^T$	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$
Q8	$[select, from, where]^T$	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$
Q9	$[select, from, where]^T$	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$
Q10	$[select, from, where]^T$	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

Table 5. Keyword vector, Weight vector and Count vector for run-time queries are given in Table 3.

Query	Keyword vector	Weight vector	Count vector	Run time vector(rv)
Q1	*[select, from, where, or]T*	$[3, 1, 1, 3]^T$	$[1, 1, 1, 1]^T$	$[3, 1, 1, 3]^T$
Q2	*[select, from, where, union, all]T*	$[3, 1, 1, 3, 3]^T$	$[2, 2, 1, 1, 1]^T$	$[6, 2, 1, 3, 3]^T$
Q3	*[Select, from, where, sp help]T*	$[3, 1, 1, 2]^T$	$[1, 1, 1, 1]^T$	$[3, 1, 1, 2]^T$
Q4	*[select, from, where, drop, table, user]T*	$[3, 1, 1, 5, 1, 1]^T$	$[1, 1, 1, 1, 1,1]^T$	$[3, 1, 1, 5, 1, 1]^T$
Q5	*[select, from, where]T*	$[3, 1, 1]^T$	$[1, 1, 1]^T$	$[3, 1, 1]^T$
Q6	*[select, from, where, shutdown]T*	$[3, 1, 1, 4]^T$	$[1, 1, 1, 1]^T$	$[3, 1, 1, 4]^T$
Q7	*[select, from, where, wait for, delay]T*	$[3, 1, 1, 2, 2]^T$	$[1, 1, 1, 1, 1]^T$	$[3, 1, 1, 2, 2]^T$
Q8	*[select, from, where, union]T*	$[3, 1, 1, 3]^T$	$[2, 2, 1, 1]^T$	$[6, 2, 1, 3]^T$
Q9	*[select, from, where, or]T*	$[3, 1, 1, 3]^T$	$[1, 2, 1, 1]^T$	$[3, 2, 1, 3]^T$
Q10	*[select, from, where]T*	$[3, 1, 1]^T$	$[2, 1, 1]^T$	$[6, 1, 1]^T$

Table 6. The angle between design time query and run-time query.

Query	padded dv vector	rv vector	$Cos\theta$	θ
Q1	$[3, 1, 1, 0]^T$	$[3, 1, 1, 3]^T$	0.742	42^0
Q2	$[3, 1, 1, 0, 0]^T$	$[6, 2, 1, 3, 3]^T$	0.82	34.92^0
Q3	$[3, 1, 1, 0]^T$	$[3, 1, 1, 2]^T$	0.86	30.68^0
Q4	$[3, 1, 1, 0, 0, 0]^T$	$[3, 1, 1, 5, 1, 1]^T$	0.54	57.32^0
Q5	$[3, 1, 1]^T$	$[3, 1, 1]^T$	1	0^0
Q6	$[3, 1, 1, 0]^T$	$[3, 1, 1, 4]^T$	0.64	50.21^0
Q7	$[3, 1, 1, 0, 0]^T$	$[3, 1, 1, 2, 2]^T$	0.76	40.54^0
Q8	$[3, 1, 1, 0]^T$	$[6, 2, 1, 3]^T$	0.9	23.84^0
Q9	$[3, 1, 1, 0]^T$	$[3, 2, 1, 3]^T$	0.75	41.41^0
Q10	$[3, 1, 1]^T$	$[6, 1, 1]^T$	0.98	11.48^0

In the form of DLL (Dynamic-link Library), this algorithm can be integrated with a web application or any legacy application to detect and prevent SQLiA on a real-time basis in any particular application. The proposed method filters the SQLiA and acts as a shield to protect the web application from SQLiA attacks. It is language-independent and easily integrates with any third-party apps without any source code modification of existing web applications. The .Net framework is used to implement the proposed algorithm.

Results and Discussion

The proposed method is evaluated on an open-source data set from GitHub. The results obtained after applying this procedure are awesome and it is compared with different existing methods based on the scope and capabilities. This method uses the mathematical model or formula to match the similarities between design time and run-time queries, which is a dynamic analysis method hence, machine learning parameters like recall, false positive, F1-score, precision etc, are not applicable in this scenario. Those measures are applicable in a machine learning-based approach to calculate the capabilities and accuracy of machine learning models. However, in this section, the main focus is on the dataset used for validation and detection accuracy of the proposed method. Also focused on the comparison of different tools/methods capabilities with the proposed method. This section includes several subsections as follows:

A. Dataset used
B. Validation using opensource GitHub dataset
C. Comparison of different tools/methods with the proposed method

A. Dataset used

We use GitHub open-source datasets to test the proposed method. The payloads are available in the raw

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

format. Data pre-processing is required to convert the raw format into .csv file and further transform the queries into design time and run-time queries. The datasets are extracted from URLs given below.

a. https://github.com/ChrisAHolland/ML-SQL-Injection- Detector/blob/master/data/burp-suite-payload.txt

b. https://github.com/Morzeux/HttpParamsDataset/blob/master/payloadfull.csv

A. Validation using opensource GitHub dataset

The proposed algorithm is validated against the GitHub dataset. In the first dataset, out of 1300 injected queries, the proposed method identifies 1219 injected queries; out of 300 normal queries, it identifies 290 normal queries with 93.76% and 96.66% detection accuracy, respectively. Similarly, for the second dataset, out of 10489 injected queries, it identifies 10280 injected queries and out of 301 normal queries, it identifies 280 normal queries with 98.01% and 93.02% detection accuracy, respectively as shown in Table 7. In a few cases, the proposed system fails to detect the SQL injection attacks where keywords are not separated with white space and in case of new keywords which is not included in the keyword list. This is because this method is unable to properly extract the design time and run time vectors in such cases. Since the design time and run time vector are not distinguished correctly there is a chance of wrong calculation of the angle of deviation between design time and run time vector. Hence, there is a scope for improvement to address the weakness of the proposed system. The main advantage of the proposed method is that it can be easily integrated with legacy applications with minimal source code modification. This method can be used in the form of DLL (dynamic link library) to call the exposed methods to extract the design time and run-time queries from real-time applications. Hence, in a few places, the source code modification is required to integrate this application with any web application. This DLL can be integrated with any web-based application. Hence the complete source code modification in legacy application is not required and it can protect the legacy web application from SQLiA vulnerability. In Table 8 many methods have been depicted and some can detect a few SQLiA vulnerabilities. Some can detect the vulnerability partially and fail in some other type of SQLiA vulnerability. However, the proposed method can detect and prevent the applications from all types of SQLiA vulnerability. The working procedure and logic of the proposed method are very simple and easy to understand as they work on similarity matching techniques to design time and run-time queries. Apart from that, like the machine learning model, the training dataset is not required hence, model train activity is not applicable in this method.

B. Comparison of different methods or tools with the proposed method

The capabilities of different tools/methods are given in Table 8. Tools like AMNESIA (Halfond et al., 2005), CSSE (Pietraszek et al., 2005), SQL-Check (Halfond et al., 2005), SQL-Guard (Buehrer et al., 2005), SQL-Rand (Park et al., 2006) uses static and dynamic analysis approach and able to detect almost all types of SQL injection attacks but fails to detect stored procedure related vulnerabilities. The tautology checker (Wassermann et al., 2004; Qbea'h et al., 2016) successfully detects the tautology attacks vulnerability but fails for the rest of all types of vulnerabilities. It can handle tautology attacks very efficiently and can be used to detect tautology attacks. This tool could not be used alone to detect all types of SQL injection vulnerabilities. Tools like JDBC checker (Gould et al., 2004) and Java Static Tainting (Livshits et al., 2005) use the static analysis approach to detect SQLiA vulnerability. The above tools can only examine the queries present in the application. Hence, for real-life injection scenarios, such tools are not useful. JDBC checker (Gould et al., 2004) and SafeQuery objects (Cook et al., 2005) only minimize the risk of SQLiA by checking the type of SQL queries. IDS (Valeur et al., 2005) and WAVES (Huang et al., 2003) are based on machine learning methods and require a large amount of injected data to learn the system. The demerits of the machine learning method are that they need a large amount of data to train the system and the result will solely depend on the training and testing data set. The performance of this mechanism depends on the capability of the training model and machine learning algorithm. Table 8 shows the applicability of different SQL injection detection tools with the types of SQL Injection vulnerabilities. A particular single tool is unable to detect all types of SQL injection attacks. Some tools can detect tautology, piggybacked, and union queries but fail to detect stored procedure vulnerabilities. Few tools only detect the queries at the application end but fail to detect them in real-life scenarios. Table 8 depicts the clear comparison of different SQL injection detection tools with the proposed method. The proposed method is able to detect and prevent all types of SQL injection vulnerabilities and also has an alert mechanism to notify the users or system administrator. The working principle of the proposed method is a very simple mathematical formula i.e., angle

Table 7. Experimental Result of GitHub Data Set.

Dataset Name	Dataset URL	Total No. of Queries	No. of Injected Queries	No. of Normal Queries	No. of Normal Queries Detected & Detection %	No. of Injected Queries Detected & Detection%
ML-SQL-Injection-Detector/data/ burp-suite-payload.txt	https://github. com/ ChrisAHolland/ ML-SQL-Injection- Detector/ blob/master/data/ burp-suite -payload.txt	1600	1300	300	290 & 96.66%	1219 & 93.76%
HttpParams Dataset/ payload full	https://github. com/ Morzeux/ HttpParams Dataset/ Injection -Detector/ blob/master/ payload full.csv	10790	10489	301	280 & 93.02%	10280 & 98.01%

Table 8. Comparison of different types of methods of SQLiA.

Sl. No.	Ref. Method	Tautology	Incorrect Queries	Union SQLiA	Piggy Backed	Stored Proc.	Inference	Alt. Encoding
1	AMNESIA (Halfond et al.,2005)	•	•	•	•	×	•	•
2	CSSE (Pietraszek et al., 2005)	•	•	•	•	×	•	×
3	IDS (Valeur et al., 2005)	◦	◦	◦	◦	◦	◦	◦
4	Java Dynamic Tainting (Haldar et al., 2005)	+	+	+	+	+	+	+
5	SQL-Check (Halfond et al., 2005)	•	•	•	•	×	•	•
6	SQL-Guard (Buehrer et al., 2005)	•	•	•	•	×	•	•
7	SQL-rand (Park et al., 2006)	•	×	•	•	×	•	×
8	Tautology checker (Wassermann et al., 2004; Qbea'h et al., 2016)	•	×	×	×	×	×	×
9	Web App Hardening (Nguyen-Tuong et al., 2005)	•	•	•	•	×	•	×
10	JDBC Checker (Gould et al., 2004)	+	+	+	+	+	+	+

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

11	Java Static Tainting (Livshits et al., 2005)	•	•	•	•	•	•	•
12	SafeQuery OB (Cook et al., 2005)	•	•	•	•	×	•	•
13	Security gateway (Scott et al., 2002)	+	+	+	+	+	+	+
14	SecuriFly (Martin et al., 2005)	+	+	+	+	+	+	+
15	SQLDOM (McClure et al., 2005)	•	•	•	•	×	•	•
16	WAVES (Huang et al., 2003)	◦	◦	◦	◦	◦	+	◦
17	Proposed Method	•	•	•	•	•	•	•
Symbolic Representation: •: Possible, ◦: Partially Possible, ×: Impossible, +: Not Applicable								

Table 9. Analysis of functionalities of different detection and prevention mechanisms.

Sl. No.	Detection/Prevention Methods	Code Modification	SQLiA detection mechanism	SQLiA prevention mechansim	Any other elements
1	AMNESIA (Halfond et al., 2005)	Not required	Self-acting	Self-acting	Not Applicable
2	CSSE (Pietraszek et al., 2005)	Not required	Self-acting	Self-acting	Req. PHP interpreter
3	IDS (Valeur et al., 2005)	Not required	Self-acting	Based on report	IDS knowledge
4	JDBC Checker (Gould et al., 2004)	Not required	Self-acting	Code modification	Not Applicable
5	Java Dynamic Tainting (Haldar et al., 2005)	Not required	Self-acting	Self-acting	Not Applicable
6	Java Static Tainting (Livshits et al., 2005)	Not required	Self-acting	Code modification	Not Applicable
7	SafeQuery OB (Cook et al., 2005)	Required	Not Applicable	Self-acting	Developer training
8	SecuriFly (Martin et al., 2005)	Not required	Self-acting	Self-acting	Not Applicable
9	Security gateway (Scott et al., 2002)	Not required	Manual process	Self-acting	Proxy filter
10	SQL-Check (Halfond et al., 2005)	Required	Self-acting (partial)	Self-acting	Key management
11	SQL-Guard (Buehrer et al., 2005)	Required	Not Applicable	Self-acting	Not Applicable
12	SQLDOM (McClure et al., 2005)	Required	Self-acting	Self-acting	Developer training
13	SQL-rand (Park et al., 2006)	Required	Self-acting	Self-acting	Key management

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

14	Tautology checker (Wassermann et al., 2004; Qbea'h et al., 2016)	Not required	Self-acting	Code modification	Not Applicable
15	WAVES (Huang et al., 2003)	Not required	Self-acting	Based on report	Not Applicable
16	Web App Hardening (Nguyen-Tuong et al., 2005)	Not required	Self-acting	Self-acting	Req.PHP interpreter
17	Proposed Method	Required	Self-acting	Self-acting	Not Applicable

of deviation between design time and run-time queries. It works perfectly on all types of SQL injection attacks. It does not use any machine learning algorithm. Hence large dataset is not required to train the model.

Due to this reason false positive and false negative parameters are not applicable in this case. The proposed method is different from existing methods or tools because the same SQL injection detection engine is capable enough to detect as well as prevent web applications from SQL injection attacks. This method applies a simple logic to check the similarity between the design time and run time vectors in terms of the angle of deviation. During the study of various tools or methods, we have seen that one single tool cannot detect all types of SQL injection attacks, but the proposed mechanism can detect all types of SQL injection vulnerabilities. Also, this tool can be integrated with any web application to protect them in real-life scenarios. The existing tools based on machine learning algorithms require large data to train and test the model and their performance is based on the training dataset. The accuracy of such models depends on training the model in a particular dataset in various scenarios, which is a very difficult and time-consuming task. Here, the proposed mechanism used the dynamic approach and a large dataset is not required to train the model. It can directly work on test datasets to identify the SQL injection vulnerability. Like machine learning models, the false positive and false negative criteria do not apply to the proposed method. Similarly, the other criteria like F1-score, precision, recall etc., generally used in machine learning algorithms, are not applicable in this scenario. The accuracy can be measured in terms of percentage, which can be calculated out of certain numbers of injected queries and how many of them are identified by this method. The detailed comparison of different methods in terms of capabilities and functionalities is shown in Table 9. The detection and prevention mechanism of the proposed method is fully automatic with minimal source code modification and can be integrated as DLL form with any web application with minimal change in the source code of the existing application.

Conclusion and Future Work

The paper proposes a new method to detect SQLiA attacks by comparing design time and run-time queries. If there is any deviation in design time and run-time queries then, it raises an alert and stops the further execution of the SQL statement. The deviation is calculated in terms of the angle of deviation between design time and run-time queries. The performance and capability of the proposed method are compared with other methods, as mentioned in Table 7 and Table 8. The angle of deviation plays a crucial role in identifying the normal and injected queries. Sometimes, the method fails when there is a new keyword and if the white space is missing between keywords. For such cases, the proposed method is unable to create the correct design time and run time vector, which fails to calculate the proper angle of deviation between the two vectors and hence this method fails in such scenarios. This part can be taken care of during the data preprocessing phase. This is a small gap that can be addressed if any researcher uses this technique to determine SQLiA vulnerability. This method only uses a dynamic analysis approach to identify SQLiA vulnerability.

It may be possible that this can be used with other techniques to produce some hybrid approach for identifying SQL injection attacks. As the main outcome, this technique can be integrated with other detection techniques to build a robust mechanism for detecting and preventing SQLiA vulnerability. It can be used as a plugin with any existing application. The proposed system can detect Illegal queries, Boolean injection, tautologies, union queries, piggy-backed queries and stored procedure-related SQLiA attacks. It is fast and can be integrated into web applications or wherever we want to use it.

DOI: https://doi.org/10.52756/ijerr.2024.v42.001

Conflict of Interest

The authors declare no conflict of interest.

References

Ali, A. B. M., Shakhatreh, A. Y. I., Abdullah, M. S., & Alostad, J. (2011). SQL-injection vulnerability scanning tool for automatic creation of SQL-injection attacks. *Procedia Computer Science*, *3*, 453–458. https://doi.org/10.1016/j.procs.2010.12.076

Buehrer, G., Weide, B. W., & Sivilotti, P.P.A.G. (2005). Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM '05),* pp.106–113. https://doi.org/10.1145/1108473.1108496

Chowdhury, S., Nandi, A., Ahmad, M., Jain, A., & Pawar, M. (2021). A Comprehensive Survey for Detection and Prevention of SQL Injection. In *Proceedings of the 7th International Conference on Advanced Computing and Communication Systems (ICACCS, 2021)*, pp. 434–437. https://doi.org/10.1109/icaccs51430.2021.9442012

Cook, W., & Rai, S. (2005). Safe query objects: statically typed objects as remotely executable queries. In *Proceedings of the 27th International Conference on Software Engineering, 2005*, pp. 97-106. https://doi.org/10.1109/icse.2005.1553552

Elia, I. A., Fonseca, J., & Vieira, M. (2010). Comparing SQL Injection Detection Tools Using Attack Injection: An Experimental Study. *In Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, pp. 289–298. https://doi.org/10.1109/issre.2010.32

Ghafarian, A. (2017). A hybrid method for detection and prevention of SQL injection attacks. In *Proceedings of the Computing Conference, 2017,* pp. 833-838. https://doi.org/10.1109/sai.2017.8252192

Gogoi, B., Ahmed, T., & Dinda, R. G. (2022a). PHP web shell detection through static analysis of AST using LSTM based deep learning. *In Proceedings of the 14th Conference on USENIX Security Symposium,* pp. 14. https://doi.org/10.1109/icaitpr51569.2022.9844206

Gogoi, B., Ahmed, T., & Dinda, R. G. (2022b). PHP web shell detection through static analysis of AST using LSTM based deep learning. In *Proceedings of the 2022 First International Conference on Artificial Intelligence Trends and Pattern Recognition (ICAITPR),* pp. 1–6. https://doi.org/10.1109/icaitpr51569.2022.9844206

Gould, C., Su, N. Z., & Devanbu, PP. (2004). JDBC checker: a static analysis tool for SQL/JDBC applications. *In Proceedings of the 26th International Conference on Software Engineering*, pp. 697-698. https://doi.org/10.1109/icse.2004.1317494

Haldar, V., Chandra, D., & Franz, M. (2006). Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05),* pp. 309–311. https://doi.org/10.1109/csac.2005.21

Halfond, W. G. J., & Orso, A. (2005). AMNESIA: analysis and monitoring for Neutralizing SQL-injection attacks. *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05). Association for Computing Machinery*, pp. 174–183. https://doi.org/10.1145/1101908.1101935

Halfond, W., Orso, A., & Manolios, PP. (2008). WASP: Protecting Web Applications using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering*, *34*(1), 65–81. https://doi.org/10.1109/tse.2007.70748

Hlaing, Z. C. S. S., & Khaing, M. (2020). A Detection and Prevention Technique on SQL Injection Attacks. In *Proceedings of the IEEE Conference on Computer Applications (ICCA, 2020)*, pp. 1–6. https://doi.org/10.1109/icca49400.2020.9022833

Huang, Y., Huang, S., Lin, T., & Tsai, C. (2003a). Web application security assessment by fault injection and behavior monitoring. *Mathematical and Computer Modelling*, *55*(1-2), 58–68. https://doi.org/10.1145/775152.775174

Huang, Y., Huang, S., Lin, T., & Tsai, C. (2003b). Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th International Conference on World Wide Web (WWW'03), Association for Computing Machinery*, pp. 148–159. https://doi.org/10.1145/775152.775174

Jamar, R., Sogani, A., Mudgal, S., Bhadra, Y., & Churi, PP. PP. (2018). Website attack prevention using E-Shield as a IDPS tool. In *Proceedings of the IEEE International Conference on System, Computation, Automation and Networking (ICSCA, 2018)*, pp. 1-7. https://doi.org/10.1109/icscan.2018.8541152

Jana, A., & Maity, D. (2020). Code-based Analysis Approach to Detect and Prevent SQL Injection Attacks. In *Proceedings of the 11th International Conference on Computing, Communication and*

Networking Technologies (ICCCNT, 2020), pp. 1–6. https://doi.org/10.1109/icccnt49239.2020.9225575

Kumar, A., Dutta, S., & Pranav, P. (2023). Supervised learning for Attack Detection in Cloud. *Int. J. Exp. Res. Rev.*, *31*(Spl Volume), 74-84. https://doi.org/10.52756/10.52756/ijerr.2023.v31spl.008

Lee, I., Jeong, S., Yeo, S., & Moon, J. (2012). A novel method for SQL injection attack detection based on removing SQL query attribute values. *Mathematical and Computer Modelling*, *55*(1–2), 58–68. https://doi.org/10.1016/j.mcm.2011.01.050

Martin, M., Livshits, B., & Lam, M. S. (2005). Finding application errors and security flaws using PQL. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, *Association for Computing Machinery*, pp. 365–383. https://doi.org/10.1145/1094811.1094840

McClure, R., & Kruger, I. (2005). SQL DOM: compile time checking of dynamic SQL statements. In *Proceedings of the 27th International Conference on Software Engineering, 2005 (ICSE 2005)*, pp. 88-96. https://doi.org/10.1109/icse.2005.1553551

Mitropoulos, D., & Spinellis, D. (2009). SDriver: Location-specific signatures prevent SQL injection attacks. *Computers & Security*, *28*(3–4), 121–129. https://doi.org/10.1016/j.cose.2008.09.005

Natarajan, K., & Subramani, S. (2012). Generation of SQL-Injection free secure algorithm to detect and prevent SQL-Injection attacks. *Procedia Technology*, *4*, 790–796. https://doi.org/10.1016/j.protcy.2012.05.129

Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., & Evans, D. (2005). Automatically hardening web applications using precise tainting. *IFIP Advances in Information and Communication Technology*, 295–307. https://doi.org/10.1007/0-387-25660-1_20

Park, J., & Noh, B. (2007). SQL Injection attack detection: Profiling of web application parameter using the sequence pairwise alignment. In *Springer eBooks*, pp. 74–82. https://doi.org/10.1007/978-3-540-71093-6_6

Pietraszek, T., & Vanden Berghe, C. (2006). Defending against injection attacks through Context-Sensitive String Evaluation. In *Lecture Notes in Computer Science*, pp. 124–145. https://doi.org/10.1007/11663812_7

Qbea'h, M., Alshraideh, M., & Sabri, K. E. (2016). Detecting and Preventing SQL Injection Attacks: A Formal Approach. In *Proceedings of the Cybersecurity and Cyberforensics Conference (CCC, 2016)*, pp. 123–129. https://doi.org/10.1109/ccc.2016.26

Rubaiei, M. A., Yarubi, T. A., Saadi, M. A., & Kumar, B. (2020). SQLIA Detection and Prevention Techniques. In *Proceedings of the 9th International Conference System Modeling and Advancement in Research Trends (SMART, 2020*, pp. 115–121. https://doi.org/10.1109/smart50582.2020.9336795

Saxena, A., Arora, A., Saxena, S., & Kumar, A. (2022). Detection of web attacks using machine learning based URL classification techniques. In *Proceedings of the 2nd International Conference on Intelligent Technologies (CONIT, 2022)*, pp. 1-13. https://doi.org/10.1109/conit55038.2022.9847838

Sharma, S., P., Chhikara, R., & Khanna, K. (2023). An efficient Android malware detection method using Borutashap algorithm. *International Journal of Experimental Research and Review*, *34*(Special Vol), 86-96. https://doi.org/10.52756/ijerr.2023.v34spl.009

Scott, D., & Sharp, R. (2002). Abstracting application-level web security. In *Proceedings of the* 11th *International conference on World Wide Web (WWW '02). Association for Computing Machinery*, pp. 396–407. https://doi.org/10.1145/511446.511498

Su, Z., & Wassermann, G. (2006). The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06). Association for Computing Machinery*, pp. 372–382. https://doi.org/10.1145/1111037.1111070

Thomas, S., Williams, L., & Xie, T. (2009). On automated prepared statement generation to remove SQL injection vulnerabilities. *Information and Software Technology*, *51*(3), 589–598. https://doi.org/10.1016/j.infsof.2008.08.002

Valeur, F., Mutz, D., & Vigna, G. (2005). A Learning-Based approach to the detection of SQL attacks. Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2005. In *Lecture Notes in Computer Science*, pp. 123–140. https://doi.org/10.1007/11506881_8

Wassermann, G., & Su, Z. (2004). An Analysis Framework for Security in Web Applications. In *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems*, pp. 70-78. https://api.semanticscholar.org/CorpusID:5102805

Zhang, L., Gu, Q., Peng, S., Chen, X., Zhao, H., & Chen, D. (2010). D-WAV: A Web Application Vulnerabilities Detection Tool Using Characteristics of Web Forms. In *Proceedings of the Fifth International Conference on Software Engineering Advances, 2010*, pp. 501–507. https://doi.org/10.1109/icsea.2010.85

How to cite this Article:

Jayanto Kumar Chowdhury, Dilip Kumar Yadav and Chandra Mouli P.V.S.S.R (2024). SQL Injection Attack Detection Based on Similarity Matching Between Vectors Extracted From Design Time and Run-Time Queries. *International Journal of Experimental Research and Review*, *42*, 01-17.

DOI : https://doi.org/10.52756/ijerr.2024.v42.001