Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X
Volume 3, Issue 10, 2025

HYBRID FRAMEWORKS FOR DETECTING AND PREVENTING SQL INJECTION ATTACKS IN DATABASE-DRIVEN APPLICATIONS

Mennam Fatima[1], M. Saleem Ahmed[2], Khalid Bin Muhammad[*3], Humayoun Saeed[4]

[1,2,4.]*BS Candidate, Department of Computer Science, Ziauddin University, Karachi, Pakistan.*
[*3.] *Associate Professor, Department of Computer Science, Ziauddin University, Karachi, Pakistan.*

[*1]mennamfghanchi@gmail.com., [*2]saleemkhanx001@gmail.com., [*3]khalid.muhammad@zu.edu.pk., [*4]humayounsaeed008@gmail.com

DOI:https://doi.org/10.5281/zenodo.17636572

Keywords
SQL injection attack(SQLIA), database/SQL queries, web applications, Blind SQL, XSS attack, Deep Learning, Graph Convolutional Network (GCN), LSTM, FastText.

Article History
Received: 11 September 2025
Accepted: 21 October 2025
Published: 04 November 2025

Copyright @Author
Corresponding Author: *
Khalid Bin Muhammad

Abstract
SQL Injection Attacks (SQLIAs) remain a major treat to database-driven applications, enabling data theft, authentication bypass, and system disruption. This study evaluates common SQLi techniques (Union Error-based, Blind) using penetration-testing tools such as Burp Suite, and reviews existing defenses including input validation, parameterized queries, and anomaly detection. We suggest a hybrid deep learning framework that combines LSTM, Graph Convolutional Networks (GCN), and FastText embeddings for better SQLIA detection and prevention in order to get around the drawbacks of traditional rule-based systems. We highlight their strengths and limitations, and propose improved mitigation strategies through a structured analysis of attack vectors and countermeasures. The findings provide practical guidance for developers and security professionals to strengthen application resilience against SQLIAs.

INTRODUCTION

Software Databases underpin nearly all modern systems and websites, storing vast volumes of structured and unstructured data. However, they remain vulnerable to SQL Injection Attacks (SQLIAs), which exploit flaws in applications that communicate with databases. By inserting malicious SQL commands, attackers can gain unauthorized access, delete, or alter data, threatening confidentiality, integrity, and availability. Despite defenses such as input validation, parameterized queries, and intrusion detection systems, SQLIAs still rank among the top web security threats, with research showing they account for nearly 25% of cyberattacks.

High-profile breaches and OWASP rankings underscore their severity: in 2021 alone, 738 new SQLi vulnerabilities were reported in the CVE database, reflecting a rising trend. While tools like SQLmap help identify common vulnerabilities, they often miss complex or evolving attack patterns. Traditional defenses and rule-based detection also fall short against sophisticated and adaptive SQLIAs. Recent studies have explored deep learning–based detection, but many models suffer from high computational cost, fixed input constraints, or overfitting. To address these limitations, our work proposes a hybrid deep learning framework combining LSTM, Graph Convolutional Networks (GCN), non-local networks, and FastText embeddings. By modeling SQL queries as graph structures, the framework achieves high detection accuracy (>99%) with low

inference time, making it practical for real-world deployments.
Motivation: The rising frequency of SQLi vulnerabilities and their severe consequences—financial losses, reputational damage, and data theft—make it imperative to strengthen defenses. This study aims to deliver practical and adaptable prevention strategies to safeguard web applications against evolving SQLIA threats.

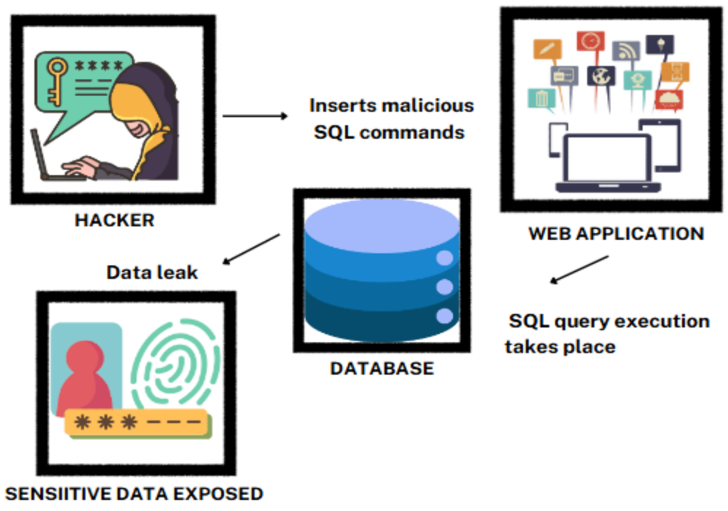
Inserts malicious SQL commands
HACKER
WEB APPLICATION
Data leak
DATABASE
SQL query execution takes place
SENSIITIVE DATA EXPOSED

Fig 1; Flow of SQL attacks

Contribution:
C1 Presenting a detailed knowledge of SQL injectiontactics, such as union-based, blind SQL, and tautology- based assaults.
C2 The current research examines the advantages, disadvantages, and areas for development of several prevention methods now in use, including input validation, parameterized queries, stored procedures, and web application firewalls (WAFs).
C3 In order to demonstrate how SQL injection vulnerabilities have changed over time and why current security solutions are still inadequate, the article examines historical data as well as contemporary event.
C4 Advanced protection strategies like Max-length, AMNESIA techniques, SAFELI tool, Target systems, and enhanced dynamic query categorization for SQL injection are recommended by this study.

1.1. Brief History on SQL injection attacks:

Online traffic has grown due to increased use of online apps and smartphones, as well as user technical skills. Businesses that switch from offline to online systems give threat actors an area of assault to target. The Top 10 SQL injection attacks and defenses are released by OWASP every few years to guard against the top ten security threats to web applications[1]. Due to many web server's vulnerabilities, the web server's scripts attack has been increased using ASP or PHP scripting injection. There are 70% SQL injection sites[2]. Thousands of security breaches have been discovered which takes place every day. 75% of the company's
websites and online apps are susceptible to security breaches[3].
One of the biggest cybersecurity risks is still SQL injection, which gives hackers the ability to access systems without authorization, alter data, or even take over whole organizations. Deep learning and machine learning techniques have been studied recently, although many of them suffer from fixed input size restrictions, computational cost, or overfitting. The model exhibits great promise for real-world cybersecurity systems by achieving accuracy above 99% while keeping low inference times by modeling SQL queries as graph structures

and utilizing GCN for flexible sequence management [3].

Table 1, SQL injection attacks happened in the last years and recently are as follows,[1]

The login system is typically the target of the most frequent attack that threatens the database system. Most attackers on the login page attempt to use brute force, which is defined as guessing the password by attempting every possible combination. The confidentiality, integrity, and availability of sensitive data are compromised by SQL injection (SQLI), a serious cybersecurity problem that takes advantage of inadequate input validation in database-driven systems. High-profile breaches demonstrate how serious it is; research indicates that SQLI is used in about 25% of cyberattacks.

Following is the distribution of attacks[4]

YEAR	DESCRIPTION
2009	The largest identity theft in American history, 130 million credit card numbers that were taken [1].
2009	32 million passwords and user accounts that were not secured and kept in a text database were compromised from RockYou's web site [1].
2011	MySQL DBMS page suffered a breach containing individual accounts and passwords of prominent persons in the field [1].
2011	A database breach at Sony, the company's website, exposed millions of unencrypted user accounts and associated text-formatted data [1].
2012	A database containing 450,000, accounts and their text-formatted passwords was leaked by Yahoo [1].
2015	Five million user accounts were compromised in the database of Vtech, a multinational producer of electrical supplies [1].
2016	A database containing administrator accounts was leaked by the US National Election Assistance Commission, which was responsible for supporting the administration of municipal and state elections [1].
2019	Georgia Technology is one of technical universities which experienced a database leak of 1.3 million people [1].
2020	French sporting goods retailer Decathlon faced a data leak affecting 123 million records, including employee and customer information, due to an unsecured database exposed to the internet [1].
2021	Social media platform Gab was targeted in an SQL injection attack, exposing around 70GB of user data, including private messages and account details. This attack was politically motivated and involved anti-hate activists [1].
2024	Kind of an SQL injection attack targeted a well-known financial institution, compromising millions of credits and debit card details. The incident has prompted calls for stronger financial industry security protocols [1].

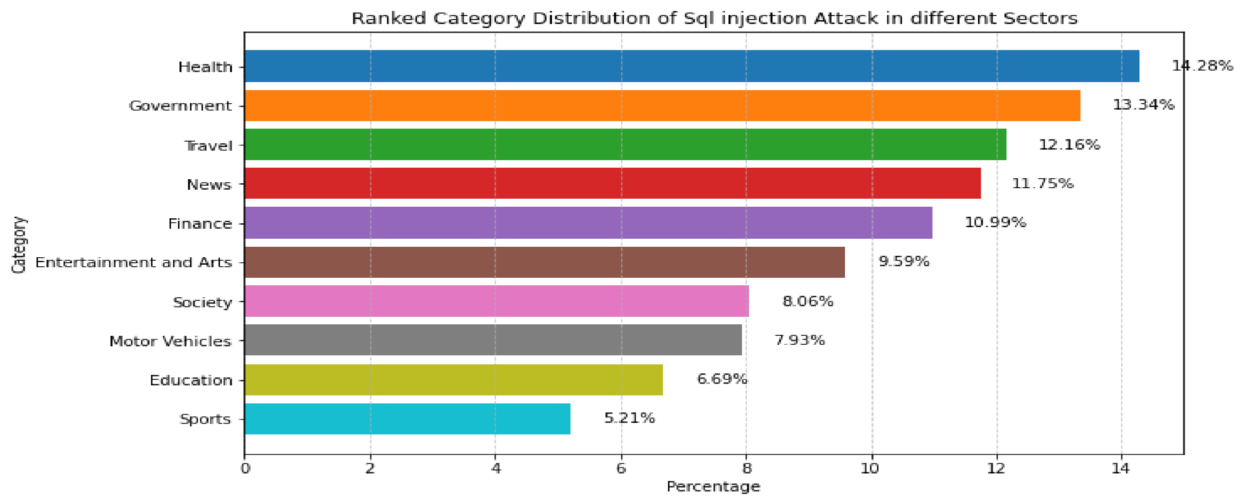

Fig 2: Distribution of Attacks in Different sectors

Literature Review

Prior studies [6] identify multiple approaches to mitigate SQL Injection Attacks (SQLIAs), including penetration testing methods such as Black Box, Grey Box, and White Box. Tools like Burp Suite are often used to detect vulnerabilities during secure testing phases, where attempts such as OR 1=1 help reveal weaknesses before moving to exploitation [6].

SQLIAs exploit applications by inserting malicious queries, highlighting the need for effective detection and mitigation

[12]. While techniques like sanitization and runtime monitoring exist, they vary in overhead and adaptability. Tokenization, behavioral profiling, and machine learning–based approaches generally achieve broader coverage and fewer false positives than traditional rule-based systems [12].

Recent deep learning–based frameworks [13] use semantic representations and neural networks to detect SQLi patterns overlooked by standard tools. Models like DeepSQLi demonstrate improved detection rates and scalability across

systems. Similarly, early research systems. Similarly,

early research [14] showed how vulnerabilities in stored procedures, when combined with dynamic SQL and functions like EXEC, allow attackers to bypass authentication. These findings emphasize the importance of secure coding and automated detection tools.

Stack-based exploitation [4] demonstrates how attackers exploit escape characters and semicolon-separated queries to execute multiple commands. This technique requires careful probing but can expose vulnerabilities in poorly sanitized inputs. Weakly typed languages like PHP further exacerbate risks [1], where unsanitized queries enable structural manipulation. Defensive methods such as prepared statements help, but advanced static analysis tools like FREESQLI enhance detection by applying type systems to identify vulnerabilities [1].

Other studies [15] examine SQLi attacks targeting HTTP/HTTPS protocols to bypass firewalls. Using datasets of over 54,000 requests, deep learning, hybrid, and pattern-based techniques have been tested for prevention. These models show promise in reducing data leakage and unauthorized access by improving detection accuracy and supporting web-based security platforms [15].

Table 2, Summary of existing Literature reviews;

TYPES OF SQL INJECTIONS

3.1 BLIND SQL injection:

Blind SQL injection occurs when an attacker can modify database queries but cannot directly see query results; the application returns only limited indicators (e.g., true/false or different page behavior). Attackers infer data by

S.No	Authors	Methodology
1	Aziz Anaoval A	Descriptive and quantitative methods, penetration testing (Black Box, Grey Box, White Box), use of Burp Suite for vulnerability testing and exploitation analysis.
2	Alenezi M	Comparative analysis of SQL injection detection and mitigation techniques, focusing on machine learning models, tokenization, and behavioral profiling.
3	Liu M	DeepSQLi, an AI-driven deep learning framework for detecting SQL injections, has been developed utilizing semantic representation and neural networks.
4	Wei K	SQL injection attacks in stored procedures are reviewed, weaknesses in dynamic SQL execution are highlighted, and automatic detection techniques are suggested.
5	Sommervoll A	Stack-based exploitation assessment, which examines how escape character manipulation and semicolon-separated queries affect SQL injection assaults.
6	Silvestre A	Analysis of type systems, input sanitization methods, and the FreeST programming language with the purpose of detecting SQL injections via static analysis.
7	Demilie W	Machine learning-powered detection and prevention, training and testing models with a dataset of 54,306 data points from HTTP(S) requests, blogs, and cookies.

observing these responses. Traditional blind techniques guess characters one-by-one using Boolean checks, which is effective but slow.

A faster method treats characters at the bit level: instead of testing full characters, attackers probe individual bits (1 or 0) using Boolean conditions. The **sql-anding** technique (introduced in 2013) applies bitwise operations to test each bit independently. This reduces the number of queries, allows parallel

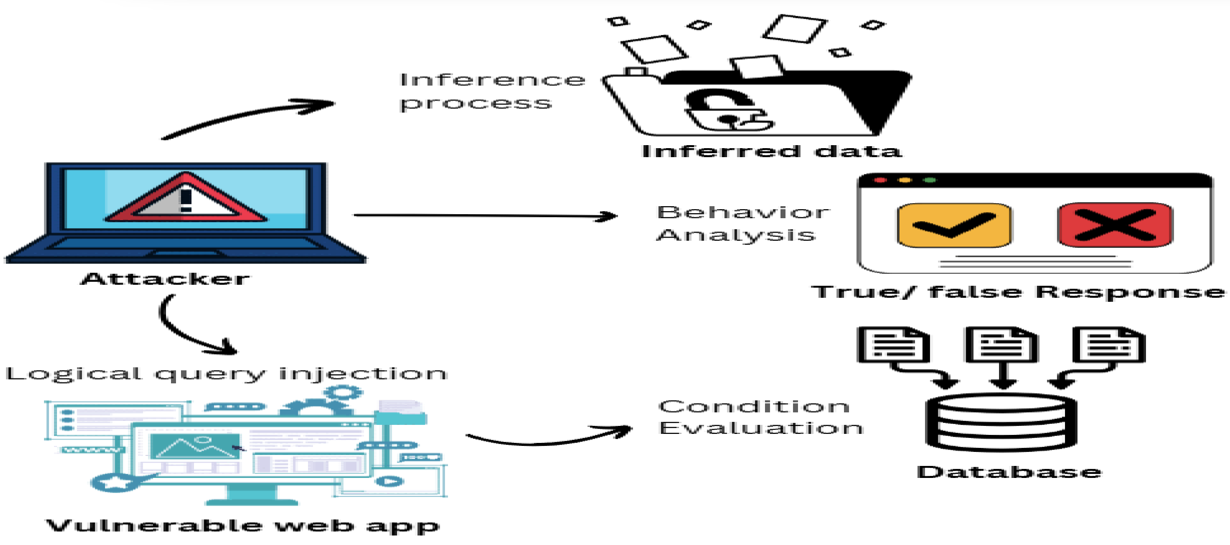

Fig 3: Blind SQL Injection

probing, and is often faster than range-based tests (e.g., BETWEEN) because bitwise operations are more efficient on many database engines [16].
Blind SQL Injections come in two types:
1. Boolean Blind - The application only gives true/false answers, often seen in login pages. Where UNION keywords are not allowed since it is too complex to inject UNION in the middle of it that's why the injection is placed in multiple queries resulting in errors. By revealing and reconstructing data from the server's binary responses, the attacker can obtain a flag using the Boolean-based blind vulnerability. To be successful, the agent must identify the escape character and provide collected responses to the proper truth value[5].
2. Non-Boolean Blind - Faster data extraction because multiple true responses reveal more information. This is embedded by a GET parameter is forwarded through a URL to check which information is available and can be shown by the ID in the application which can assist the hacker into exploiting the application or data faster [5].
In the sample below, attackers attempt to guess the database's table names. In this case, he had two attempts. On the first attempt, he requested that the database choose the first record because the table "admin" did not exist in the database, resulting in a
false result as part of the AND condition.

http://example.com/news.php?id=132 AND (select 1 from admin limit 0,1) = 1

Assuming the "users" a table exists in the database; the second query can choose the first entry. Once the website loads, the attacker is aware that the database
has a table named "users".

http://example.com/news.php?id=132 AND (select 1 from user limit 0, 1) =1

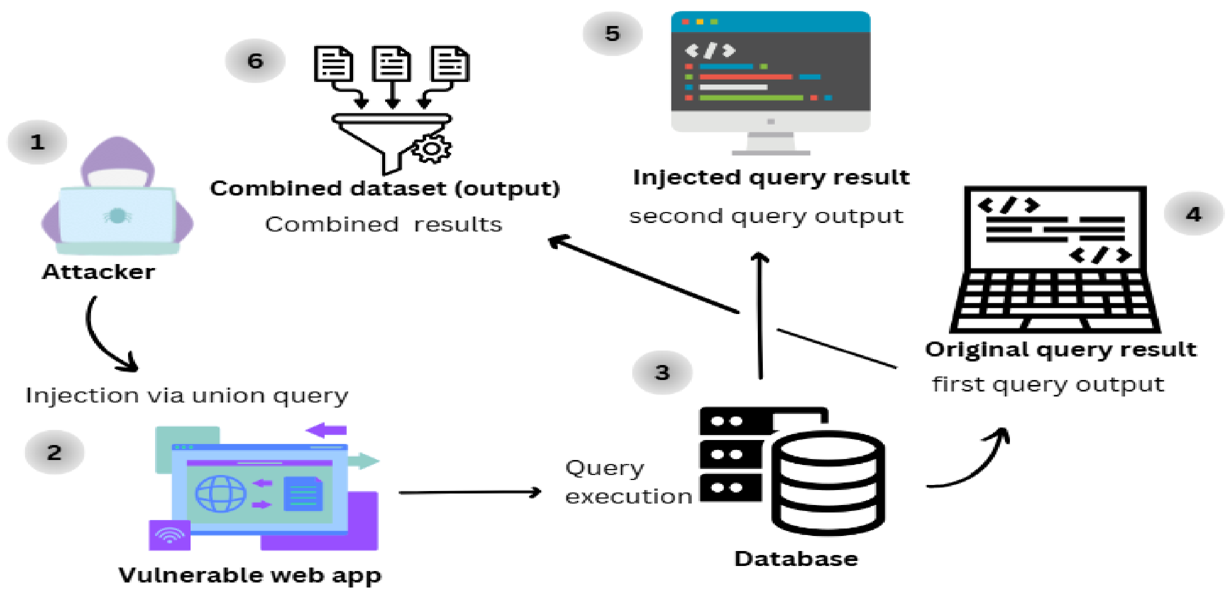

Fig 4: Process of Union SQL Injection Query

In the sample below, attackers attempt to guess the database's table names. In this case, he had two attempts. On the first attempt, he requested that the database choose the first record because the table "admin" did not exist in the database, resulting in a false result as part of the AND condition.

3.2. UNION based SQL injection:

In the SQL language, two separate queries can be joined together using the UNION operator. By inserting a subsequent query and union with the initial SQL statement, the attacker utilizes "UNION" to obtain information from other tables using a union query[6]. Union query attacks attempt to access data while bypassing authentication[7].

Using this method, the attacker forces the database to retrieve data from tables other than those indicated in the acceptable SQL query. An attacker may insert the string {' UNION SELECT username, password from user info where user_name='abc'--} into the login field, as shown in the running example[8]. This would result in the following query,

```
SELECT * FROM users WHERE login='' UNION SEELCT
password from user_info where user_name='abc'-- AND pass=
```

The data set provided query is entirely under the attackers' control. This for a given query can be altered by an attacker using weak arguments in union query attacks, which can be used to fool the computer into producing data from a table that was not intended. Attackers do this by inserting a phrase that resembles this: UNION SELECT < what was exactly injected>. The second injected attack causes the database to return a dataset that combines the output of the injected second query with the output of the original first query[8]. In the following example, the attacker adds a union SQL query to the URL in the browser's address bar [6].

Fig 4; Brief working of union-based attacks.

3.3. Cross Site scripting attack:

One of the top 10 vulnerabilities for web application use are XSS attacks[10]. XSS is one of the attacks where, an attacker constructs and runs a JavaScript fragment within the targeted domain's security context, inserting harmful material into web pages displayed by a trusted online application. Most web applications are vulnerable to XSS attacks because they do not properly filter user input before loading web pages. When a website is penetrated, users may be sent to malicious websites that open automatically, their login

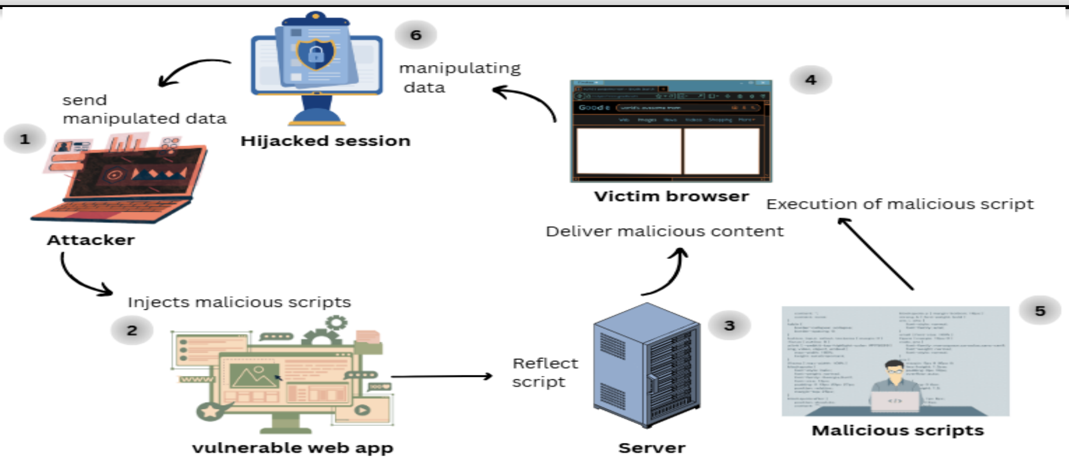

Fig 5: Union SQL Injection Query flow

stolen, and their entire session hijacked. Cross-site scripting attacks include inserting harmful content into the content being sent by the hacked site. When the resulting material gets in the client-side web browser, which has been delivered from a trustworthy source and so functions under the rights allowed to that system[10]. The content is processed like regular content because it is purported to originate from a reliable server. For instance, the pseudo code below, uses a straightforward server-site script to demonstrate how the most recent comments are shown on a website[11]:

```
echo "<html><body>";
echo "<h1> Most recent comments<h1>";
echo database. lastestComments;
echo " </html></body>";
```

The scripts presume that there is only text in the comments. However, an attacker can submit his remark because the user's input is immediately incorporated. As a result, visitors to the page will get the following response:[11]

```
echo"<html><body>";
echo " <h1> Most recent comments </h1>";
echo " <script< dosomethingevil();</script>';
echo"</html></body>";
```

XSS is categorized as follows;

• Reflected, is when malicious code is passed to the
server from the user's browser and returned to the server[10].

Persistent is when a code stays in one place, like a code saved in a database and repeatedly executed on the client browser which increases its risk [10].

• A DOM-based XSS attack, which includes both persistent and reflected attacks; the attacker can exploit DOM data and change DOM components[10].

XSS (Cross Site Scripting) attacks target the client side of online apps/sites, attempting to deceive internet users and cause security breaches[12].

Developers' failure to validate input in web applications leads to effective XSS attacks. There are too many existing techniques that are either not publicly available or difficult to adopt[10].

3.4. Tautology- based SQL injection:

The goal of tautology-based injection is to defeat authentication, identify injectable parameters, and extract data. SQL injection attacks circumvent requirements by making the "WHERE" clause always true[6]. The attacker leverages a field that can be used in a query's WHERE conditional, where it is changed into not a tautology because all of the rows in the query targets the table. In a tautology-based attack, code is injected into conditional expressions to ensure that they always evaluate true[13], condition such as (1=1) or (- -). The following query shows the illustration of tautology SQLIA[9].

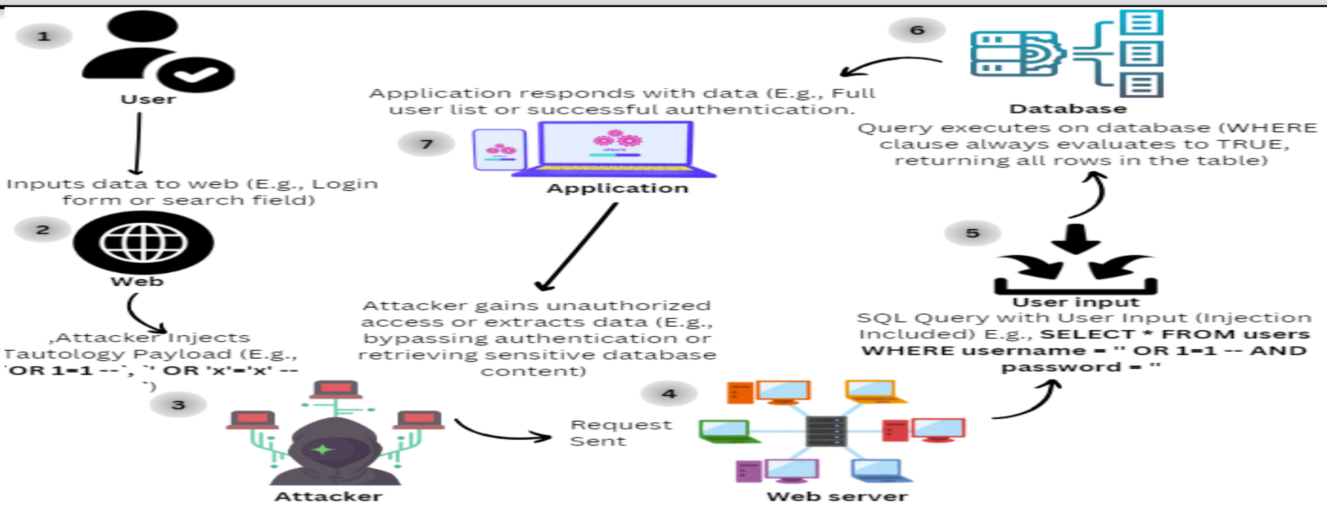
1
User
Inputs data to web (E.g., Login form or search field)
2
Web
Attacker Injects Tautology Payload (E.g., 'OR 1=1 --', ' OR 'x'='x' -- ')
3
Attacker
Request Sent
4
Web server
SQL Query with User Input (Injection Included) E.g., SELECT * FROM users WHERE username = '' OR 1=1 -- AND password = ''
5
User input
Query executes on database (WHERE clause always evaluates to TRUE, returning all rows in the table)
6
Database
Application responds with data (E.g., Full user list or successful authentication.
7
Application
Attacker gains unauthorized access or extracts data (E.g., bypassing authentication or retrieving sensitive database content)

Fig 6: Flow of tautology attacks

```
Select from employees where employee_ID='1' or '1=1' AND
employee_password='1234';
```

For a tautology-based attack to work, an attacker must not only include the susceptible parameters, but also include the coding blocks that assess query results; it is only successful when the code displays all of the returned records or does some action if at least one record is received[13]. For example, in the input field for login the attackers submit "'or 1=1 - - "which gives the query in the form of;

```
SELECT accounts FROM users WHERE login='' or 1=1-- AND
pass='' AND pin=
```

In this example, the returned set contains non-null
value, indicating successful user authentication. The application would use the displayAccounts() method
to display all accounts in the database's returned
set[13].
Another example is a query that gets all the columns whose username is equal to "Administrator" and their password is equal to "root"[6].

```
SELECT * FROM TABLE_USERS WHERE USERNAME
'ADMINISTRATOR' and PASS = 'ROOT'
```

This is the final query that results in the concatenation of the outside variable and returns all rows from the users table with their username as administrator, without considering the password.

```
SELECT * FROM TABLE_USERS WHERE USERNAME =
'ADMINISTRATOR' and PASS = '' OR 1=1 --'
```

SQL allows for a wide variety of function calls and values, making it difficult to detect tautologies.
Checking user input for patterns such as n=n or an equal sign is insufficient[14].
procedures are predefined SQL queries that do specific

3.5. Stored procedures:

Stored procedures are predefined SQL queries that do
operations. SQLIAs of this type attempt to execute stored procedures in the database. Once an attacker has determined which backend database is being used, they can generate SQLIAs to execute stored procedures, including ones that interact with the operating system. It is a common misconception that using stored procedures to construct Web applications renders them immune to SQLIAs. Developers may be surprised to realize that stored procedures are just as susceptible to assaults as regular programs. Stored procedures, typically constructed in scripting languages, may have vulnerabilities such as buffer overflows

enabling attackers to execute arbitrary code or escalate privileges[13].
Assume the attacker inserts the following input:

sport'; SHUTDOWN; - -

This query retrieves all sports-related news from the news table.

3.6 Proposed Hybrid Framework:

Graph Convolutional Networks (GCN), FastText embeddings, and Long Short-Term Memory (LSTM) networks are all integrated in this study's hybrid deep learning-based framework to overcome the drawbacks of conventional and single-model detection methods. The hybrid method makes use of SQL queries' rich semantic representations, graph-like dependency structures, and sequential nature.

A. Data Collection and Preprocessing

The dataset used for this study's experiments comes from a variety of sources, including synthetically generated SQL queries and publicly accessible SQL injection datasets like the CSIC 2010 HTTP Dataset and Kaggle SQLi Benchmarks. To provide balanced learning, the dataset includes both malicious (injection-based) and benign (legitimate) SQL statements [5].
Prior to being fed into the hybrid model, each SQL query underwent a number of preprocessing steps. Initially, the queries were tokenized, which separated each statement into distinct symbols like string literals, operators (=, --,'OR '1'='1), and keywords (SELECT, FROM, WHERE). Tokenization was followed by normalization, which reduced noise by converting text to lowercase and eliminating unnecessary whitespace and comments.
Each token was then converted into a 300-dimensional dense vector representation using FastText embeddings. FastText was chosen due to its ability to capture semantic relationships, even for uncommon or out-of-vocabulary terms, by representing words using a combination of subword n-grams [3]. This step ensures that the embedding layer effectively differentiates between legitimate SQL terms and attack payloads (e.g., UNION SELECT vs. UPDATE DROP) based on contextual semantics.

B. Model Architecture

This is the final query that results in the concatenation of the outside variaSeveral layers make up the suggested hybrid architecture (shown in Fig. 2), which is intended to function in concert for SQL injection detection.
FastText Embedding Layer: FastText is used to embed the input SQL tokens into continuous 300-dimensional vectors. Both morphological and semantic similarities between query tokens are captured by this representation. In contrast to Word2Vec, FastText can identify partial token similarities thanks to its subword-based training, which enhances its ability to identify obfuscated injection payloads [3].
LSTM Layer: A bi-directional LSTM layer receives the embedded sequences. Because LSTMs can learn long-term dependencies in sequential data, they can be used to model the syntactic order of SQL queries. Suspicious token transitions and patterns, including tautology attacks, piggybacked queries, and concatenation-based manipulations, are detected by the LSTM [4].
SQL queries are then represented as graph structures using the raph Convolutional Network (GCN) Layer, where each node stands for a token and the edges show the syntactic or dependency relationships between tokens. The GCN can learn contextual dependencies across non-adjacent tokens thanks to this graph representation, which enables it to recognize patterns that purely sequential models are unable to see [6]. By combining data from nearby nodes, GCN is able to comprehend intricate connections between SQL clauses such as WHERE, ORDER BY, and injected subqueries.
Dense and Softmax Layers: To combine learned features from both models, the outputs from LSTM and GCN are concatenated and sent through a dense fully connected layer. The likelihood that each query is malicious or benign is then output by a Softmax classifier. For better discrimination, the combined model makes use of both relational (GCN) and temporal (LSTM) representations [1].

C. System Workflow

Stored procedures are predefined The suggested hybrid SQL injection detection system's entire process starts with the collection of SQL queries from multiple sources, including database access logs, proxy monitors, and web application firewalls (WAFs). After a query is acquired, it goes through a preprocessing stage in which the FastText embedding model is used to tokenize, normalize, and transform the input into dense vector representations while maintaining semantic and subword information. After processing the preprocessed query, the LSTM layer records temporal and sequential dependencies in the tokenized query structure. Following their fusion and passage through dense layers, the extracted features from the LSTM and GCN layers are used by as a classifier to predict whether a given query is malicious or benign. Lastly, the system either executes the query if it is a valid query or blocks and logs it if it is determined to be a possible SQL injection attempt based on the classifier's output.

Comparative analysis

Types of SQL injection	Principal impacts	Examples of affected data	Potential consequences	Prevention techniques
Blind SQL injection	unapproved access to data without direct data sight.	PII, private company information, and login information.	confidentially loss, possible breach alerts, sanctions from the authorities, and a decline in user confidence.	Make use of robust error handling, parameterized queries, stored procedures, and input validation.
Union-based SQL injection	combining queries to retrieve data from several tables.	Internal business data, financial records, and user credentials.	Identity theft, data leakage, financial fraud, reputational damage, compliance breaches.	Use web application firewalls (WAFs), implement parameterized queries, enforce input sanitization, and embrace the least privilege concept.
Tautology-based SQL injection	overcomes authentication procedures, permitting unwanted access.	User accounts, admin credentials, sensitive user data.	Complete database access, illegal alteration or removal of data, and a possible system takeover.	Utilize parameterized queries, access control policies, an input validation, and ORMs to guarantee appropriate handling of
Stored procedures	If input is not sanitized, stored SQL logic may be manipulated.	any information—possibly sensitive records—processed by stored methods.	Command execution without authorization, data manipulation, and possible system control through SQL functions (e.g., xp_cmdshell).	Employ robust parameter types, make sure that all input is thoroughly cleaned, audit stored procedures on a regular basis, restrict access, and stay away from dynamic SQL inside procedures.
Cross-site scripting (XSS)	indirect effects on database security through script execution and session hijacking.	input forms, cookies, and user session data.	Phishing, virus propagation, session hijacking, and stolen passwords.	Use HTTPOnly and secure cookies, implement Content Security Policy (CSP), validate and clean input, and avoid user-generated content.

Vulnerability types	Preventions
Blind SQL injection	Make use of stored procedures, parameterized queries, input validation, and efficient error management.
UNION based SQL Injection	Use a web application firewall, implement parameterized queries, clean input, and enforce the least privilege principle.
Tautology- based SQL injection	Use Object-Relational Mappers (ORMs), parameterized queries, input validation, and access controls.
Cross-site scripting (XSS)	Verify input, encrypt output, utilize HTTP Only cookies, enforce Content Security Policy (CSP), and clean up user-generated content.
Stored procedures	Limit access, utilize strong parameter types, make sure input is sanitized, and do routine audits and monitoring.

SQL Injection Prevention Techniques:

Several techniques have been made available to make SQL injection attacks less severe. One of the common ways to prevent using prepared statements, a SQL injection attack separates the SQL input from the SQL structure.[10] Following are the techniques to prevents various attacks,Following are the methods for avoiding SQL injection attacks: Max length: Restricting input field length reduces SQLi risks. Combined with stored procedures, this ensures queries remain controlled, though poorly written procedures may still introduce vulnerabilities [24].

AMNESIA: AMNESIA (Analysis and Monitoring for Neutralizing SQL Injection Attacks) combines static and dynamic analysis for Java-based applications, verifying queries against pre-built models before execution [5]. Later research proposed machine learning–based detection, which showed better results than SQLCheck and AMNESIA, particularly for Blind SQLi [10].

XSS and SQLIA Prevention:Tools like Ardilla detect SQLi and XSS attacks, while execution-flow techniques using finite-state automata help mitigate JavaScript-based threats. Noxes provides partial protection by monitoring HTTP requests, though attackers can bypass it with HTML tags. Dynamic Data Tainting tracks malicious inputs but cannot detect all hidden channels [10].

SAFELI: SAFELI analyzes source code at compile time to detect SQLi vulnerabilities. However, its implementation is incomplete and cannot handle tautology-based attacks [25].

Stored Procedures: Stored procedures, similar to prepared statements, isolate SQL code in the data layer and apply the least-privilege principle, reducing damage if exploited. By filtering only valid inputs,
they prevent malicious code injection and offer stronger protection than web server–executed queries [8][2].5.6 Target Systems: Prevention strategies are tested on different systems: toy systems (for testing
only), limited-scale systems (functional but less reliable), and large-scale production systems. The latter provide the most accurate evaluation of SQLIA defenses [12].

5.7 Dynamic Query Classification: This approach classifies queries at runtime, using SQLi sanitizers to compare SQL statement execution with safe patterns. It supports black box testing, proxy filters, and
intrusion detection systems for enhanced protection [26].

Results:

The findings confirm that SQL Injection Attacks (SQLIAs) remain a critical threat to web applications, as demonstrated by the steady rise in reported incidents and vulnerabilities. While traditional defenses such as

parameterized queries, input validation, and Web Application Firewalls (WAFs) mitigate common attacks, they prove insufficient against more advanced techniques, including blind, union-based, tautology-based, and stored procedure exploits.
Our analysis shows that AI-driven detection methods, such as DeepSQLi, significantly improve detection accuracy and reduce false negatives. However, these approaches also introduce computational overhead and occasional false positives, highlighting the need for optimization. Complementary strategies—including maximum length constraints, runtime monitoring (e.g., AMNESIA), and emerging blockchain-based security models—show additional promise in strengthening overall protection.

Discussion:
The study suggests that no single technique can fully safeguard against SQLIAs; rather, a layered defense strategy is required. This includes static and dynamic analysis, continuous penetration testing, and real-time monitoring. Traditional measures like sanitization and parameterized queries remain necessary, but they are increasingly challenged by evolving, adaptive attack methods.
AI- and ML-based frameworks represent a promising future direction, offering higher accuracy and adaptability. Yet, these solutions must be balanced against system performance and operational costs. Moreover, security is not only a technical challenge but also an organizational one—regular policy updates, developer training, and secure coding practices are essential to reduce the attack surface. A holistic approach, combining technical innovation with process improvement, is critical to ensuring resilience against SQL injection threats.

Future work:
Future research should focus on enhancing AI-based SQLi detection frameworks to achieve higher accuracy with lower computational cost. Techniques such as behavioral analysis and anomaly detection can complement machine learning by identifying subtle variations in user interactions. Blockchain technologies may also provide secure, tamper-resistant mechanisms for transaction logging and data integrity.
Developing standardized frameworks and coding best practices will help reduce vulnerabilities during application design and development. Extending penetration testing tools to cover advanced SQLi variants, and adopting hybrid testing strategies that integrate static and dynamic analysis, will further strengthen defenses. Finally, collaboration between academia, industry, and regulatory bodies is vital to address emerging attack vectors and ensure practical, scalable security solutions.

Conclusion:
This study underscores that SQL Injection Attacks (SQLIAs) continue to threaten the confidentiality, integrity, and availability of database-driven systems. By examining past incidents and analyzing common attack types—such as union-based, blind SQLi, and stored procedure exploits—we highlight the persistence and adaptability of these threats.
Our review of existing prevention and detection techniques shows that while traditional approaches remain valuable, they are inadequate against advanced attack strategies. AI-driven frameworks, enhanced runtime monitoring, and hybrid defenses offer promising avenues for improvement. Ultimately, defending against SQLIAs requires a multi-layered approach that combines technical innovation, continuous testing, secure development practices, and organizational commitment to cybersecurity.

REFERENCES
[1] N. Bedeković, L. Havaš, T. Horvat, and D. Crčić, “The Importance of Developing Preventive Techniques for SQL Injection Attacks,” *Tehnicki Glasnik*, vol. 16, no. 4, pp. 523–529, Sep. 2022, doi: 10.31803/tg-20211203090618.
[2] P. C. Xue, “SQL injection attack and guard technical research,” in *Procedia Engineering*, 2011, pp. 4131–4135. doi: 10.1016/j.proeng.2011.08.775.

[3] J. P. Singh, “Analysis of SQL Injection Detection Techniques,” *Theoretical and Applied Informatics*, vol. 28, no. 1 & 2, pp. 37–55, Feb. 2017, doi: 10.20904/281-2037.

[4] A. Sharma, A. Tyagi, and M. Bhardwaj, “Analysis of techniques and attacking pattern in cyber security approach,” *Int J Health Sci (Qassim)*, pp. 13779–13798, Jun. 2022, doi: 10.53730/ijhs.v6ns2.8625.

[5] Å. Å. Sommervoll, L. Erdődi, and F. M. Zennaro, “Simulating all archetypes of SQL injection vulnerability exploitation using reinforcement learning agents,” *Int J Inf Secur*, vol. 23, no. 1, pp. 225–246, Feb. 2024, doi: 10.1007/s10207-023-00738-3.

[6] A. Sadeghian, M. Zamani, and S. M. Abdullah, “A taxonomy of SQL injection attacks,” in *Proceedings - 2013 International Conference on Informatics and Creative Multimedia, ICICM 2013*, IEEE Computer Society, 2013, pp. 269–273. doi: 10.1109/ICICM.2013.53.

[7] M. Alenezi, M. Nadeem, and R. Asif, “SQL injection attacks countermeasures assessments,” *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 21, no. 2, pp. 1121–1131, Feb. 2020, doi: 10.11591/ijeecs.v21.i2.pp1121-1131.

[8] D. Aich, “Secure Query Processing By Blocking SQL injection,” 2009.

[9] B. Sun, S. Zhao, and G. Tian, “SQL queries over encrypted databases: a survey,” *Conn Sci*, vol. 36, no. 1, 2024, doi: 10.1080/09540091.2024.2323059.

[10] Bhanwarlal and Irfan Khan, “XSS and SQL Injection Detection and Prevention Techniques (A Review),” *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pp. 53–60, Jan. 2022, doi: 10.32628/cseit22816.

[11] O. C. Abikoye, A. Abubakar, A. H. Dokoro, O. N. Akande, and A. A. Kayode, “A novel technique to prevent SQL injection and cross-site scripting attacks using Knuth-Morris-Pratt string match algorithm,” *EURASIP J Inf Secur*, vol. 2020, no. 1, Dec. 2020, doi: 10.1186/s13635-020-00113-y.

[12] A. Kumar and S. Binu, “Proposed Method for SQL Injection Detection and its Prevention,” 2018.

[13] W. G. J. Halfond, J. Viegas, and A. Orso, “A Classification of SQL Injection Attacks and Countermeasures,” 2006.

[14] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, “Using parse tree validation to prevent SQL injection attacks,” in *SEM 2005 - Proceedings of the 5th International Workshop on Software Engineering and Middleware*, 2005, pp. 106–113. doi: 10.1145/1108473.1108496.

[15] Y. Tiwari and M. Tiwari, “A Study of SQL of Injections Techniques and their Prevention Methods,” 2015. [Online]. Available: www.ijcaonline.org

[16] M. R. Haque, M. M. Rahman, and M. Ahmed, “Deep Hybrid Models for Web Application SQL Injection Detection,” *IEEE Access*, vol. 10, pp. 9783–9795, 2022.

[17] Y. Kim, J. Lee, and S. Kwon, “Hybrid Deep Neural Network for Intrusion Detection in Web Applications,” *Expert Systems with Applications*, vol. 168, p. 114324, 2021.

[18] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching Word Vectors with Subword Information,” *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[19] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[20] B. Rossi, F. Li, and D. Xu, “SQL Injection Detection Using Deep Learning Techniques,” *Journal of Information Security and Applications*, vol. 53, p. 102518, 2020.

[21] T. N. Kipf and M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” in *Proc. Int. Conf. on Learning Representations (ICLR)*, 2017.