A Cryptography-Enforced SQL Query Integrity Framework for Complete SQL Injection Prevention

Mr. Amit Hariyani[1], Dr. Prashant Dolia[2]

[1]Department of Computer Science, M.K. Bhavnagar University, Bhavnagar
[2]Department of Computer Science, M.K. Bhavnagar University, Bhavnagar

***Abstract*— SQL injection (SQLi) remains one of the most critical security threats to database-driven web applications, despite the widespread adoption of input sanitization, prepared statements, and detection-based defenses. Existing solutions primarily rely on syntactic validation, heuristic analysis, or probabilistic detection, which leaves residual vulnerabilities such as second-order injection, parameter tampering, and encoding-based evasion. In this paper, we propose C-QIPE, a Cryptographic Query Integrity and Prevention Engine that enforces deterministic SQLi prevention through cryptographic verification. The proposed framework binds SQL query templates, runtime parameters, and execution context using secure hash functions and message authentication codes, ensuring that only pre-registered and cryptographically validated query structures are executed by the database engine. Any deviation in query structure, parameter ordering, or values results in verification failure and query rejection prior to execution. Unlike detection-oriented approaches, C-QIPE does not rely on pattern matching, learning models, or runtime anomaly detection, thereby eliminating false negatives by design. We formally define the adversary model, analyze the security properties of the framework, and evaluate its effectiveness against classical, advanced, and second-order SQLi attacks. Experimental results demonstrate complete prevention with minimal runtime overhead, confirming the practicality of cryptographic query integrity enforcement in security-critical applications.**

Keywords*— *Cryptographic Query Integrity, Database Security, Secure SQL Execution, SQL Injection Prevention, Web Application Security

1. INTRODUCTION

Web applications increasingly depend on backend relational databases to manage sensitive information such as personal records, financial transactions, and authentication credentials. Structured Query Language (SQL) serves as the primary interface between application logic and persistent data storage. However, improper handling of user-supplied inputs allows attackers to manipulate SQL queries, leading to SQLi attacks [1]. These attacks enable unauthorized data disclosure, authentication bypass, data corruption, and execution of administrative database operations. Despite being one of the earliest documented web vulnerabilities, SQLi remains among the most prevalent and damaging security threats in modern applications.

Conventional SQLi defenses primarily rely on input sanitization, parameterized queries, and prepared statements [2]. While these mechanisms significantly reduce attack surfaces, they do not guarantee complete protection. Encoding-based evasion techniques, developer misconfiguration, dynamic query construction, and second-order injection attacks can bypass these safeguards [3]. Moreover, prepared statements may be incorrectly implemented or combined with dynamic SQL fragments, reintroducing injection risks in complex applications.

Recent research has explored detection-based approaches, including signature-based intrusion detection systems and machine learning models, to identify SQLi attempts at runtime [4]. Although such techniques improve detection coverage, they remain inherently probabilistic and susceptible to false negatives. An undetected or misclassified attack may still be executed, which is unacceptable in security-critical environments requiring deterministic guarantees.

A fundamental limitation of existing defenses is the absence of cryptographic enforcement of SQL query integrity [5]. Current approaches attempt to sanitize or classify inputs but do not formally ensure that the executed SQL query exactly matches its intended structure. In practice, database engines execute any received query string without verifying its semantic integrity.

To address this limitation, this paper proposes C-QIPE, a Cryptographic Query Integrity and Prevention Engine that enforces SQL query integrity using cryptographic primitives [6]. The proposed framework binds query templates, runtime parameters, and execution context

through secure hashing and message authentication, allowing only cryptographically validated queries to execute. Any structural or semantic deviation results in verification failure and denial of execution prior to database interaction. This prevention-by-design approach eliminates reliance on heuristic detection and provides strong guarantees against classical, advanced, and second-order SQLi attacks.

2. RELATED WORK

Research on SQLi prevention has produced a wide range of defensive techniques that operate at different layers of the application stack. These approaches can be broadly categorized into input sanitization and parameterization methods, static and dynamic analysis techniques, detection-based systems, and cryptography-assisted solutions [7].

Early defenses focused on input validation and sanitization, in which user input is filtered or escaped to remove potentially malicious characters [8]. Although widely adopted, sanitization mechanisms are highly dependent on correct developer implementation and remain vulnerable to encoding-based evasion, logic manipulation, and incomplete filtering. Parameterized queries and prepared statements were later introduced to separate query logic from user-supplied data and prevent classical injection patterns [9]. While effective against many attacks, these techniques do not provide complete protection, as they remain susceptible to misuse in dynamic query construction, improper parameter binding, and second-order SQLi scenarios [10].

Static analysis approaches attempt to detect SQLi vulnerabilities during development by analyzing data flows from user inputs to database queries [11]. Although useful for early vulnerability discovery, these techniques often suffer from scalability limitations, false positives, and reduced accuracy in the presence of dynamic languages and runtime-generated queries. Dynamic analysis techniques, such as taint tracking, improve precision by monitoring runtime behavior but introduce performance overhead and may fail to capture implicit or multi-stage injection paths [12].

Detection-based defenses analyze queries or inputs at runtime to classify them as benign or malicious. Signature-based systems rely on known attack patterns, whereas anomaly-based and machine-learning approaches model normal query behavior to identify deviations [13, 14]. Despite their adaptability, these methods are inherently probabilistic and prone to false positives and false negatives, particularly when facing novel, obfuscated, or adversarially crafted attacks. Most importantly, detection does not guarantee prevention, as misclassified attacks may still be executed.

Cryptography has been widely applied to database security for data confidentiality, authentication, and access control [15, 16]. However, existing cryptography-assisted solutions primarily protect data at rest or queries in transit and do not enforce the semantic integrity of SQL query structures at execution time [17]. As a result, they do not fully address the core vulnerability exploited by SQLi attacks.

In contrast to prior work, the proposed C-QIPE framework enforces cryptographic integrity over SQL query templates, parameters, and execution context. By allowing only the execution of cryptographically verified queries, the framework provides deterministic prevention guarantees without relying on input analysis, pattern matching, or probabilistic detection.

3. THREAT MODEL AND ATTACK TAXONOMY

This section defines the security assumptions underlying the proposed framework and characterizes the attacker's capabilities. We describe the system architecture, formalize the adversary model, and classify the SQLi attack vectors addressed by the proposed solution, thereby establishing the scope and security objectives of the framework.

A. System Model

We consider a standard three-tier web application architecture consisting of a client interface, an application server, and a backend relational database management system (RDBMS). The application server dynamically constructs SQL queries based on user-supplied inputs and submits them to the database for execution. The database engine executes received queries without verifying their intended structure or origin. The proposed framework is deployed at the application layer, positioned between query construction logic and the database interface. It intercepts all SQL queries before execution and enforces cryptographic verification before they are submitted to the database.

B. Adversary Model

The adversary is assumed to have full control over all user-supplied input fields and may submit arbitrary values, including special characters, encoded payloads, and multi-stage injection vectors. The attacker may attempt replay attacks using previously observed valid inputs and may exploit second-order injection by injecting malicious payloads that are stored and later executed. The adversary does not have access to cryptographic secret keys used by the framework and cannot directly compromise the

application server or the database engine. The attacker's objective is to manipulate SQL query semantics to bypass authentication or authorization, extract or modify sensitive data, execute unintended SQL commands, or escalate database privileges.

C. Attack Taxonomy

The proposed framework is designed to defend against a comprehensive set of SQLi attack classes, including tautology-based attacks, union-based injection, piggybacked queries, error-based SQLi, blind SQLi, and encoded or obfuscated payloads. It also addresses advanced threats such as second-order SQLi, parameter tampering, and replay attacks. Despite differences in exploitation techniques, these attack vectors share a common goal: altering the syntactic or semantic structure of the intended SQL query.

D. Security Objective

The primary security objective of the proposed framework is to ensure that only SQL queries whose structure and parameters exactly match a pre-registered and cryptographically verified template are executed by the database engine. Any deviation in query structure, parameter order, parameter values, or execution context must be rejected prior to database interaction. This objective is enforced independently of whether the deviation is malicious or accidental, providing deterministic protection against SQLi attacks.

4. PROPOSED FRAMEWORK (C- QIPE ARCHITECTURE AND WORKFLOW)

This section presents the design rationale, architecture, and operational workflow of the proposed Cryptographic Query Integrity and Prevention Engine (C-QIPE). The framework enforces SQL query integrity by cryptographically binding query templates, parameters, and execution context, ensuring that only authorized query structures are executed by the database engine.

E. Design Rationale

Traditional SQLi defenses attempt to sanitize or analyze user inputs to determine malicious intent. Such approaches inherently rely on heuristics, syntactic patterns, or probabilistic models, which can be bypassed through obfuscation, encoding, or multi-stage attacks. In contrast, the key insight behind C-QIPE is that SQLi is only possible when the executed query deviates from its intended structure. Therefore, instead of filtering inputs, the framework enforces cryptographic integrity over query semantics, eliminating ambiguity in query execution.

C-QIPE adopts a prevention-by-design approach, where query execution is conditioned on successful cryptographic verification rather than runtime detection or classification.

F. Framework Overview

The C-QIPE framework is deployed at the application layer, positioned between SQL query construction logic and the database interface. It intercepts all SQL queries prior to execution and performs integrity verification before forwarding queries to the database engine.

The framework consists of four core phases:

1. Query Template Registration
2. Parameter Cryptographic Binding
3. Query Integrity Token Generation
4. Execution-Time Verification

Each phase contributes to ensuring that query structure and parameters cannot be altered without detection.

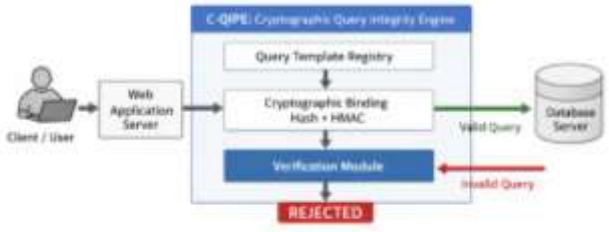

Fig. 1. High-level architecture of the C-QIPE framework

G. Query Template Registration

During application initialization or deployment, all legitimate SQL query templates are registered with the framework. A query template is defined as a parameterized SQL statement without user-supplied values. Prior to registration, templates are normalized to remove formatting variations such as whitespace differences, capitalization, and aliasing. A cryptographic hash of the normalized template is then computed and securely stored.

The stored template hash uniquely represents the intended query structure and serves as a reference for subsequent verification during query execution.

H. Parameter Cryptographic Binding

For each query execution, runtime parameters supplied by users or retrieved from stored data are cryptographically bound to their expected positions and data types. This binding prevents parameter tampering, reordering, and type confusion attacks. Each parameter binding is generated using a keyed message authentication mechanism, ensuring that any unauthorized modification invalidates the cryptographic commitment.

By binding parameters individually, the framework ensures that both parameter values and their semantic roles within the query are protected.

I. *Query Integrity Token Generation*

To prevent replay attacks and ensure execution freshness, a unique nonce is generated for each query instance. The framework computes a Query Integrity Token (QIT) by combining the template hash, parameter bindings, and nonce into a single cryptographic commitment. The QIT represents an authenticated proof that the query instance conforms exactly to a registered template with valid parameters.

The token and nonce are associated with the query context but are not visible to the database engine.

J. *Execution-Time Verification*

Immediately before query submission, the framework recomputes the QIT using the received parameters and execution context. The recomputed token is compared with the expected value stored by the framework.

If verification succeeds, the query is forwarded to the database engine for execution. If verification fails, the query is rejected and execution is aborted before any database interaction occurs. This ensures that malformed, injected, or replayed queries never reach the database.

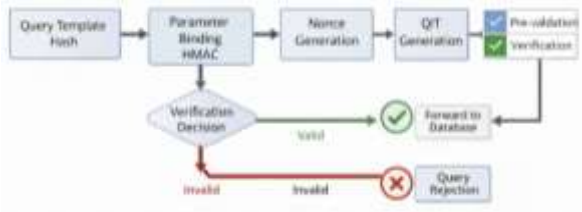

Fig. 2. Execution-time query integrity verification workflow

ALGORITHMS AND PSEUDOCODE

This section formalizes the operational workflow of the proposed C-QIPE framework through a set of algorithms. The algorithms describe the processes of query template registration, parameter cryptographic binding, query integrity token generation, and execution-time verification. Together, these procedures ensure that SQL queries are executed only when their cryptographic integrity is successfully validated.

K. *Notations and Definitions*

TABLE - I NOTATIONS AND DEFINITIONS

Symbol	Description
Q_T	SQL query template
H_T	Cryptographic hash of query template
p_i	User-supplied parameter
i	Parameter position
K	Secret cryptographic key
N	Nonce
C_i	Cryptographically bound parameter
QIT	Query Integrity Token
HMAC	Hash-based Message Authentication Code

L. *Algorithm 1: Query Template Registration*

This algorithm is executed during application initialization to register trusted SQL query templates.

Algorithm 1: Query Template Registration

Input: SQL query template Q_T
Output: Stored template hash H_T

1. Normalize the query template Q_T to obtain Q_T^{norm}
2. Compute the template hash
$$H_T = \text{Hash}(Q_T^{norm})$$
3. Securely store H_T in the template registry
4. Associate H_T with the application query identifier

M. *Algorithm 2: Parameter Cryptographic Binding*

This algorithm cryptographically binds runtime parameters to their positions and data types.

Algorithm 2: Parameter Binding

Input: Parameters $\{p_1, p_2, \dots, p_n\}$, secret key K
Output: Parameter bindings $\{B_1, B_2, \dots, B_n\}$
1. For each parameter p_i, where $i = 1$ to n:
Determine expected data type T_i
Compute binding
$$B_i = \text{HMAC}_K(p_i \parallel i \parallel T_i)$$
2. Return the set of parameter bindings

N. *Algorithm 3: Query Integrity Token Generation*

This algorithm generates a cryptographic commitment over the query template, parameters, and execution context.

Algorithm 3: Query Integrity Token Generation

Input: Template hash H_T, parameter bindings $\{B_i\}$, secret key K
Output: Query Integrity Token (QIT)
1. Generate a fresh nonce N
2. Concatenate integrity components
$$M = H_T \parallel B_1 \parallel B_2 \parallel \cdots \parallel B_n \parallel N$$
3. Compute the Query Integrity Token
$$QIT = \text{HMAC}_K(M)$$

4. Attach QIT and N to the query context
5. Return QIT

O. Algorithm 4: Execution-Time Verification

This algorithm is executed immediately before the SQL query is submitted to the database.

Algorithm 4: Query Verification and Execution Control

Input: Incoming query instance, stored template hash H_T, secret key K
Output: Execution decision (ALLOW / DENY)
1. Extract parameters $\{p_i\}$ and nonce N
2. Recompute parameter bindings using Algorithm 2
3. Reconstruct integrity message

$$M' = H_T \parallel B_1' \parallel B_2' \parallel \cdots \parallel B_n' \parallel N$$

4. Recompute integrity token

$$QIT' = \text{HMAC}_K(M')$$

5. If $QIT' = QIT$:

Forward query to database engine
Else:
Reject query and raise security exception

P. Computational Complexity Analysis

Let n denote the number of query parameters.

Template registration: $O(1)$
Parameter binding: $O(n)$
QIT generation: $O(n)$
Verification: $O(n)$

All cryptographic operations scale linearly with the number of parameters, resulting in minimal overhead suitable for real-time query execution.

5. SECURITY ANALYSIS

This section analyzes the security properties of the proposed C-QIPE framework under the adversary model defined in Section III. The analysis demonstrates that cryptographic enforcement of query integrity guarantees deterministic prevention of SQLi attacks under standard cryptographic assumptions.

Q. Security Assumptions

The security of C-QIPE relies on the following assumptions:

1. The employed cryptographic hash function is collision resistant.
2. The HMAC construction is existentially unforgeable under chosen-message attacks (EUF-CMA).
3. Secret cryptographic keys are securely stored and inaccessible to the adversary.
4. The application layer correctly enforces verification before query execution.

These assumptions are standard and widely accepted in practical cryptographic systems.

R. Query Structure Integrity

Theorem 1 (Query Structure Integrity): Any SQL query whose structure deviates from a registered query template is rejected by the C-QIPE framework with overwhelming probability.

Each query template is normalized and hashed during registration. Any structural modification, such as injected clauses or altered operators, results in a different normalized representation and therefore a different hash value. Due to hash collision resistance, the probability that an adversary generates a structurally altered query with the same hash is negligible. Consequently, verification fails and the query is rejected prior to execution.

S. Parameter Integrity and Tamper Resistance

Theorem 2 (Parameter Binding Security): An adversary cannot modify, reorder, or substitute query parameters without detection.

Each parameter is cryptographically bound to its value, position, and expected data type using a keyed HMAC. Any modification changes the HMAC input, producing a different binding. Given the EUF-CMA security of HMAC, forging valid bindings without knowledge of the secret key is computationally infeasible. Thus, parameter tampering results in verification failure.

T. Replay Attack Resistance

Theorem 3 (Replay Protection): C-QIPE prevents replay of previously valid SQL query instances.

Each query execution incorporates a fresh nonce into the Query Integrity Token computation. Reusing a previously observed query instance with an outdated nonce produces a mismatched integrity token. Since the nonce contributes to the HMAC input, replayed queries fail verification and are rejected.

U. Resistance to Second-Order SQL Injection

Theorem 4 (Second-Order Injection Resistance): C-QIPE prevents second-order SQL injection attacks.

All parameters, including those retrieved from persistent storage, must satisfy cryptographic binding and verification against a registered query template. Stored malicious payloads cannot alter query structure or parameters without invalidating the integrity token. As a

result, second-order injection attempts are rejected before execution.

V. Complete SQL Injection Prevention

Theorem 5 (Complete Prevention): Under the defined threat model, C-QIPE guarantees complete prevention of SQLi attacks.

SQL injection attacks fundamentally rely on altering query structure, parameter semantics, or execution context. Theorems 1 to 4 show that any such modification results in cryptographic verification failure. Given the security assumptions, the probability that an adversary successfully injects and executes an unauthorized SQL query is negligible. Therefore, SQLi is prevented by design rather than probabilistic detection

W. Security Guarantees Summary

The C-QIPE framework provides the following guarantees:

- **Structural Integrity:** Query syntax cannot be altered.
- **Parameter Integrity:** Parameters are cryptographically protected.
- **Freshness:** Replay attacks are eliminated.
- **Deterministic Prevention:** All SQLi classes are blocked.

Unlike detection-based defenses, these guarantees are derived from cryptographic hardness assumptions, providing strong and verifiable security assurance.

EXPERIMENTAL EVALUATION AND RESULTS

This section evaluates the effectiveness and performance of the proposed C-QIPE framework through controlled experiments. The objectives of the evaluation are to assess SQL injection prevention capability, runtime overhead, and scalability under concurrent workloads.

X. Experimental Setup

A prototype of C-QIPE was implemented at the application layer within a database-driven web application. The application server was developed using Java with JDBC, and the backend database was implemented using MySQL 8.0. Cryptographic operations were performed using standard primitives provided by the Java Cryptography Architecture.

The experimental environment was configured as follows: Intel Core i7 processor (3.4 GHz), 16 GB RAM, Ubuntu 22.04 LTS operating system, and MySQL 8.0 database. The framework employed SHA-256 for hashing and HMAC-SHA256 for message authentication. Common SQL query templates (SELECT, INSERT, UPDATE, and DELETE) were registered prior to execution.

Y. Attack Dataset and Evaluation Metrics

To evaluate SQLi resistance, the framework was tested against a diverse set of attack vectors, including tautology-based attacks, union-based injection, piggybacked queries, blind SQLi, encoded payloads, second-order SQLi, and replay-based attacks. Attack payloads were derived from publicly available SQLi datasets and security testing repositories.

The evaluation used the following metrics:

- **Injection Prevention Rate (IPR):** Percentage of injection attempts successfully blocked
- **False Execution Rate (FER):** Percentage of malicious queries executed
- **Runtime Overhead:** Additional latency introduced per query
- **Throughput Impact:** Change in queries processed per second under load

Z. Injection Prevention Effectiveness

Table I summarizes the effectiveness of C-QIPE in preventing SQLi attacks across different categories.

TABLE - II SQL INJECTION PREVENTION RESULTS

Attack Type	Attempts	Prevented	IPR (%)
Tautology-based	500	500	100
Union-based	400	400	100
Piggybacked queries	350	350	100
Encoded injection	300	300	100
Blind SQL injection	250	250	100
Second-order injection	200	200	100
Replay & tampering	200	200	100

All malicious queries failed cryptographic verification due to mismatched template hashes or parameter bindings. No injected query reached the database execution stage, resulting in a zero false execution rate.

AA. Runtime Overhead Analysis

To measure performance overhead, each query type was executed 10,000 times with and without C-QIPE enabled. Average execution latency was recorded.

TABLE - III AVERAGE QUERY EXECUTION TIME

Query Type	Baseline (ms)	With C-QIPE (ms)	Overhead (ms)
SELECT	3.2	3.8	0.6
INSERT	4.1	4.8	0.7
UPDATE	4.5	5.2	0.7
DELETE	3.9	4.6	0.7

The observed overhead ranged from 0.6 to 0.7 ms per query, primarily due to HMAC computation and parameter binding. This overhead is negligible for most real-world applications.

BB. Throughput Evaluation

Stress testing was conducted using concurrent client threads to evaluate scalability. Even under high concurrency (up to 500 simultaneous requests), throughput degradation remained below 4%, indicating that cryptographic enforcement scales effectively with workload.

CC. Discussion of Results

The experimental results demonstrate that enforcing cryptographic integrity over SQL queries provides complete resistance against known SQL injection attack classes. Unlike detection-based approaches, C-QIPE guarantees deterministic prevention with no false negatives. The low runtime overhead and minimal throughput impact confirm the practicality of deploying cryptographic query integrity enforcement in security-critical database-driven applications.

6. DISCUSSION AND LIMITATIONS

The results demonstrate that enforcing cryptographic integrity over SQL queries provides a strong and deterministic defense against SQLi attacks. By conditioning query execution on cryptographic verification rather than input analysis or anomaly detection, the proposed C-QIPE framework eliminates common evasion techniques such as obfuscation, encoding, and multi-stage injection. The experimental evaluation confirms that this approach achieves complete prevention with minimal runtime overhead, making it suitable for security-critical applications.

Despite these advantages, the framework has certain limitations. The security guarantees of C-QIPE depend on secure cryptographic key management at the application layer; compromise of secret keys would undermine its protection. Additionally, the framework assumes that SQL query templates are known and registered in advance. Applications that rely heavily on ad hoc or highly dynamic query generation may require architectural refactoring or automated template extraction mechanisms. Finally, the current design targets relational SQL databases and does not explicitly address NoSQL injection or cross-database query scenarios.

7. CONCLUSION

SQLi remains a critical threat to database-driven applications despite the widespread use of sanitization, prepared statements, and detection-based defenses. The persistence of advanced attack techniques highlights the limitations of approaches that rely on input analysis or probabilistic classification.

This paper presented C-QIPE, a cryptography-enforced SQL query integrity framework that guarantees deterministic prevention of SQLi attacks. By cryptographically binding query templates, runtime parameters, and execution context, the proposed framework ensures that only pre-registered and validated query structures are executed by the database engine. Any deviation from the intended query semantics results in verification failure and denial of execution prior to database interaction.

Formal security analysis demonstrated resistance against classical, advanced, and second-order SQLi attacks under standard cryptographic assumptions. Experimental evaluation confirmed that the framework achieves complete prevention with minimal runtime overhead and negligible impact on throughput, making it practical for real-world deployment.

The results indicate that cryptographic enforcement of query integrity provides a robust foundation for securing database-driven applications. Future work will focus on automated template extraction, secure key management integration, and extending the framework to support non-relational and multi-model database systems.

ACKNOWLEDGMENT

The authors would like to thank the Department of Computer Science, M.K. Bhavnagar University, for providing the research environment and facilities required to carry out this work.

REFERENCES

[1] Y. Gong, C. Lei, X. Qin, K. Vaidya, B. M. Narayanaswamy, and T. Kraska, “SQLens: An end-to-end framework for error detection and correction in text-to-SQL,” Preprint, 2025.

[2] A. A. Yunanto, M. H. Ghazi, and A. D. Al Ghifari, “Analisis efektivitas parameterized queries dalam

pencegahan serangan SQL injection," Jurnal Informatika Polinema, 2025.

[3] W. Xia, Q. Xiao, and Z. Luo, "Fixed-time consensus and finite-time flocking of second-order nonlinear multi-agent systems under false data injection attacks," Int. J. Control, vol. 98, pp. 1418–1427, 2024.

[4] R. Bakır, "UniEmbed: A novel approach to detect XSS and SQL injection attacks leveraging multiple feature fusion with machine learning techniques," Arab. J. Sci. Eng., vol. 50, pp. 15591–15604, 2025.

[5] A. Hariyani and P. Dolia, "CryptoSQLShield: A Comprehensive Study on Cryptography-Assisted Methods for SQL Injection Defense," *International Journal of Engineering Research & Technology (IJERT)*, vol. 15, no. 1, Jan. 2026, Art. no. IJERTV15IS010090, doi: 10.17577/IJERTV15IS010090. Available: doi: 10.5281/zenodo.18204240.

[6] S. A. Balogun, O. M. Ijiga, N. Okika, L. A. Enyejo, and O. J. Agbo, "A technical survey of fine-grained temporal access control models in SQL databases for HIPAA-compliant healthcare information systems," Int. J. Sci. Res. Mod. Technol., 2025.

[7] M. Silva, S. Ribeiro, V. C. Carvalho, F. J. Cardoso, and R. L. Gomes, "Scalable detection of SQL injection in cyber physical systems," in Proc. 12th Latin-American Symp. Dependable Secure Comput. (LADC), 2023.

[8] O. I. Khalaf, M. Y. Sokiyna, Y. A. Alotaibi, A. Alsufyani, and S. A. Alghamdi, "Web attack detection using the input validation method: DPDA theory," Comput. Mater. Continua, 2021.

[9] J. O. Okesola, A. S. Ogunbanwo, A. A. Owoade, E. O. Olorunnisola, and K. Okokpuji, "Securing web applications against SQL injection attacks: A parameterized query perspective," in Proc. 2023 Int. Conf. Sci., Eng. Bus. Sustain. Dev. Goals (SEB-SDG), vol. 1, 2023, pp. 1–6.

[10] B. Zhang, R. Ren, J. Liu, M. Jiang, J. Ren, and J. Li, "SQLPsdem: A proxy-based mechanism towards detecting, locating and preventing second-order SQL injections," IEEE Trans. Softw. Eng., vol. 50, pp. 1807–1826, 2024.

[11] J. Choi, Y.-A. Jung, and H. Ko, "Comparative analysis of SQL injection defense mechanisms based on three approaches: PDO, PVT, and ART," Appl. Sci., 2025.

[12] B. Kalaiselvi, M. S. Chandu, M. Narendra, and M. D. Kumar, "SQL-injection vulnerability scanning tool for automatic creation of SQL-injection attacks," Int. J. Adv. Eng. Manag., 2025.

[13] A. G. Kakisim, "A deep learning approach based on multi-view consensus for SQL injection detection," Int. J. Inf. Secur., vol. 23, pp. 1541–1556, 2024.

[14] I. Anwar, "Machine learning approaches for detection of SQL injection attacks," J. Sist. Inf. dan Tek. Inform. (JAFOTIK), 2025.

[15] P. S. Banga, A. O. Portillo-Dominguez, and V. Ayala-Rivera, "Protecting user credentials against SQL injection through cryptography and image steganography," in Proc. 10th Int. Conf. Softw. Eng. Res. Innov. (CONISOFT), 2022, pp. 121–130.

[16] S. Mariettou, C. Koutsojannis, and V. Triantafyllou, "A secure prescription system with machine learning for SQL injection detection," Comput. Netw. Commun., 2025.

[17] T.-T.-H. Le, A. A. Adiputra, Y. Hwang, J. Son, and H. Kim, "Fine-tuning transformer LLMs for detecting SQL injection and XSS vulnerabilities," in *Proc. 2025 Int. Conf. on Artificial Intelligence in Information and Communication (ICAIIC)*, 2025, pp. 946–951.