International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 01 , January - 2026

CryptoSQLShield: A Comprehensive Study on Cryptography-Assisted Methods for SQL Injection Defense

Mr. Amit Hariyani
Research Scholar
Department of Computer Science,
M.K.Bhavnagar University,
Bhavnagar, Gujarat, India - 364002

Dr. Prashant Dolia
Research Supervisor
Department of Computer Science,
M.K.Bhavnagar University,
Bhavnagar, Gujarat, India - 364002

***Abstract* - SQL injection (SQLi) represents a significant threat to the integrity and security of web applications by exploiting vulnerabilities within database systems, thereby allowing unauthorized access and manipulation of sensitive information. Current mitigation strategies, including input validation, parameterized queries, web application firewalls (WAFs), and machine learning detection algorithms, often fall short against sophisticated attacks that employ obfuscation and fragmentation. This study introduces CryptoSQLShield, an innovative dual-layer defense architecture designed to enhance resilience against SQLi attacks. The framework integrates cryptographic input sanitization with real-time query analysis, facilitating a prevention module using customizable user-defined encryption/decryption (UDF) mechanisms to decouple user input from query syntax. Concurrently, a detection module applies template-based validation and anomaly scoring to reconstructed queries before execution. Unlike traditional solutions that rely on static cryptographic protocols, CryptoSQLShield offers flexible encryption strategies, adaptable for various contexts, including controlled environments, research applications, and educational settings. Empirical evaluations conducted on standard SQLi testing frameworks reveal a substantial decline in exploitation success rates, alongside improved recall for detecting obfuscated payloads, substantiating the framework's efficacy in achieving a favorable balance among attack resilience, false-positive reduction, and operational efficiency.**

Keywords - SQL Injection, Cryptography, Encryption, Decryption, Query Template Verification, Web Security

I. INTRODUCTION

Backend databases are essential components of modern web applications, supporting authentication, personalization, e-commerce transactions, and analytical processing. Due to this reliance, database-driven systems represent attractive targets for attackers. SQLi is still a significant threat among web vulnerabilities because it exploits a fundamental boundary: mixing untrusted user input with executable query logic. A typical SQLi occurs when a web application concatenates user input into a SQL statement [1]. If input is not validated correctly and parameterized, attackers can inject SQL operators, clauses, or stacked statements, leading to unauthorized access, privilege escalation, data exfiltration, or data destruction [2].

Traditional SQL injection attacks exploit insecure handling of user-supplied input within application query construction logic by directly injecting malicious SQL code into application queries [3]. As shown in Fig. 1, in a typical vulnerable web app, an attacker can combine crafted input with SQL statements, thereby altering query logic and gaining unauthorized access to sensitive data [4]. The proposed CryptoSQLShield framework, on the other hand, fundamentally changes this attack surface by encrypting user inputs with user-defined cryptographic functions before the application processes them. Encrypted parameters are checked for accuracy and freshness, and decrypted only in a controlled execution environment[5]. This prevents destructive payloads from altering SQL syntax, so direct query manipulation is no longer possible.

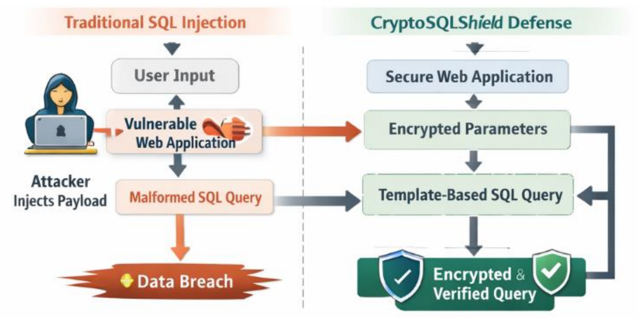


Fig. 1. SQL Injection Attack vs CryptoSQLShield Defense

The main goals of traditional defenses against SQLi attacks are to prevent unsafe query construction and to validate user input. Commonly used methods include parameterized queries and prepared statements, which keep the query structure separate from the user-supplied data. Other methods include input validation and sanitization, which try to filter out malicious characters or attack patterns [6]. The principle of least privilege limits the damage that successful attacks can do, while stored procedures are often used to hide database logic. Web application firewalls (WAFs) also use signature-based rules to block known attack patterns[7].

More recent methods use deep learning and machine learning models to distinguish between good and bad queries [8]. These methods reduce risk, but they can still be used with advanced evasion techniques and in contextual injection scenarios. Despite the widespread adoption of standard defense mechanisms, determined attackers can frequently circumvent them with several advanced methods. Some of these are query obfuscation through comments, whitespace manipulation, and case variation [9]. Others are encoding-based evasion techniques, such as URL encoding, Unicode transformations, and double encoding [10]. More advanced attacks include second-order SQLi, in which harmful payloads are stored in the database and executed later, and context-bypass attacks, in which attackers exploit injection points that validation logic doesn't cover [11]. Logic-based attacks, such as tautology manipulation, inference-based conditions, and timing attacks, make it even harder to detect and stop them, underscoring the need for stronger defense strategies. These limitations necessitate a defense strategy that integrates both prevention and detection capabilities.

A. *Motivation for Cryptography-Assisted SQLi Defense*

Cryptography has historically been used to ensure the confidentiality and integrity of data at rest and in transit. CryptoSQLShield applies this concept to SQL query parameters, ensuring that attacker-controlled content never manifests as raw executable SQL tokens at the database boundary. In other words, CryptoSQLShield ensures that user-provided inputs are never sent or processed as raw SQL fragments. Instead, application-layer parameters are encrypted before they are sent and can be decrypted only on the server side in a very controlled manner. SQL statements are made only from predefined templates with strict parameter binding. This means that user-controlled data can't change the structure of the query. A verification step ensures that reconstructed queries follow approved templates, guaranteeing that the structure is correct before execution.

This method significantly increases the difficulty for an attacker to introduce control tokens, such as quotation marks, logical operators, or UNION clauses, to SQL statements. The framework prevents tampering and replay attacks at the parameter level by using message authentication codes and nonce-based freshness checks to verify integrity. Also, CryptoSQLShield is meant to work with existing best practices, such as prepared statements and access control mechanisms, rather than replace them. This makes web applications even more secure.

B. *Contributions*

This paper presents CryptoSQLShield, an all-encompassing framework that combines cryptographic defenses with runtime detection and verification of SQLi. The suggested system uses a two-layer architecture that combines user-defined encryption and decryption functions with structural query validation and behavioral analysis. A straightforward cryptographic workflow is shown to protect SQL parameters without using built-in cryptographic primitives. This makes the method suitable for controlled research and teaching settings. The framework also includes template-based query verification and anomaly-based detection to identify hidden or suspicious injection attempts. Extensive experimental evaluation and ablation analysis demonstrate the effectiveness of the proposed approach and that combining prevention and detection strategies with a reasonable amount of runtime overhead is a good idea.

II. BACKGROUND AND THREAT MODEL

A. *SQL Injection Categories*

There are many types of SQLi attacks, but they all use the same basic methods to modify how SQL queries are executed. Attacks based on tautology exploit logical conditions to make queries appear correct, thereby allowing attackers to bypass authentication repeatedly. Union-based attacks add additional SELECT statements to retrieve data from tables that weren't intended to be accessed. In piggybacked or stacked query attacks, multiple SQL commands are injected into a single execution context, potentially leading to destructive actions such as data deletion. Inference-based or blind SQLi attacks exploit boolean conditions or timing delays to obtain sensitive information without revealing the query results. Comment-based evasion techniques use SQL comment syntax to get around filtering systems, while encoding-based attacks use URL encoding or Unicode transformations to hide harmful payloads.

B. *Attacker Capabilities*

This study assumes attackers can send crafted inputs via various interaction points, such as web forms, URL parameters, HTTP headers, cookies, and API requests. Attackers can use obfuscation, encoding, and replay techniques to bypass detection systems. It is assumed that attackers do not have access to server-side cryptographic keys or trusted execution environment secrets, which are kept safe in the trusted computing environment.

Table 1 presents a structured overview of SQLi attack categories and illustrates how CryptoSQLShield mitigates each through integrated prevention and detection mechanisms.

TABLE I. THREAT MODEL AND SQL INJECTION ATTACK COVERAGE IN CRYPTOSQLSHIELD

Attack Type	**Injection Vector**	**Impact on Traditional Systems**	**CryptoSQLShield Prevention**	**CryptoSQLShield Detection**
Tautology-based SQLi [12]	Login forms, APIs	Authentication bypass	Encrypted parameters prevent logical manipulation	Token anomaly and logic pattern detection
Union-based SQLi [13]	Search fields, URLs	Data leakage from multiple tables	Template enforcement blocks UNION injection	Suspicious keyword detection
Piggybacked queries [14]	Input fields	Execution of additional SQL statements	Query structure fixed via templates	Statement boundary anomaly detection
Comment-based evasion [15]	URLs, headers	Bypass input filters	Encrypted payload neutralizes comment syntax	Comment token analysis
Encoded SQLi [16]	Forms, cookies	Signature evasion	Decryption after canonicalization	Encoding entropy detection
Replay attacks [17]	Network replay	Reuse of malicious requests	Nonce-based freshness validation	Repeated nonce detection

Fig. 2 illustrates the overall architecture of the proposed CryptoSQLShield framework, highlighting the interaction between the cryptographic prevention layer and the runtime detection and verification layer in securing SQL query execution.

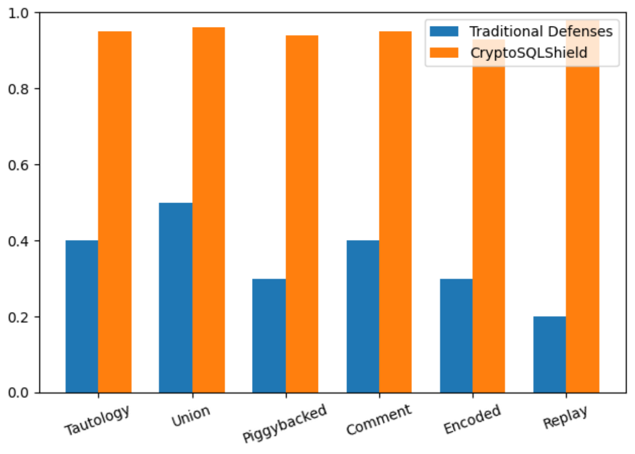


Fig. 2. SQL injection attack coverage comparison between traditional defenses and CryptoSQLShield.

C. *Defender Goals*

The main goal of CryptoSQLShield is to prevent untrusted user input from altering the syntax of SQL queries while still allowing runtime detection of malicious patterns. The framework's goal is to identify suspicious payload characteristics, unusual request rates, and integrity violations without imposing excessive computational overhead. CryptoSQLShield is meant to work well in real-world web application environments by balancing strong security guarantees with ease of deployment.

III. LITERATURE REVIEW

SQLi remains one of the most prevalent and harmful vulnerabilities in web applications, despite extensive research and long-term solutions. As a result, many different approaches to defense have been proposed in the literature. There are four main types of these methods: traditional prevention techniques, runtime monitoring and proxy-based defenses, machine learning and deep learning–based detection systems, and cryptography-assisted data protection methods [18]. This section critically examines these categories and underscores the deficiencies that necessitate the proposed CryptoSQLShield framework.

A. *Traditional SQL Injection Prevention Techniques*

The main goal of early SQLi defenses is to prevent unsafe queries. Parameterized queries and prepared statements are widely regarded as effective baseline defenses against SQL injection attacks [19]. They do this by separating the query structure from the user-provided data. These methods significantly reduce the risk of SQLi at the syntax level by binding input values as parameters rather than concatenating strings [20]. Numerous studies have demonstrated their effectiveness in stopping attacks that use classical tautology and union [21].

Techniques for validating and cleaning input aim to filter out or escape malicious characters such as quotation marks, semicolons, and comment tokens [22]. These methods are easy to use, but they are weak and rely on having the proper rules. Attackers often get around these filters by using obfuscation techniques such as encoding, case changes, and adding or removing whitespace. Static code analysis tools also try to find vulnerable query construction patterns. In contrast, the code is being written, but they can't guarantee protection against runtime manipulation or on-the-fly queries.

Even though traditional methods significantly reduce attack surfaces, they depend heavily on developers using them correctly and applying them consistently across all code paths. Also, they don't provide much information about how attacks work and don't protect against replay attacks, second-order SQLi, or changing query parameters while they are being sent [23].

B. *Web Application Firewalls and Signature-Based Defenses*

Web application firewalls (WAFs) are a common type of runtime defense that analyzes incoming HTTP requests and blocks payloads that match known SQLi signatures [24]. Signature-based systems work well against known attacks and protect everything from a single place without requiring code changes in each application. But because they depend on set rules, they can be tricked by obfuscation, encoding, and new ways of building payloads.

Some proxy-based solutions enhance WAF's capabilities by inspecting SQL queries before they reach the database [25]. These systems check query syntax, monitor query execution, or ensure that rules are followed [26]. Proxy-based approaches improve detection accuracy, but they typically operate without application logic and don't provide cryptographic guarantees of data integrity or freshness [27]. So, advanced attacks like second-order SQLi and replay-based exploitation may still work if malicious payloads are stored and then executed again in trusted contexts.

C. *Machine Learning and Deep Learning–Based SQLi Detection*

Recent studies increasingly use machine learning (ML) and deep learning (DL) methods for SQLi detection [28]. These methods usually treat SQL queries as sequences of tokens or characters and train classifiers to distinguish good from bad inputs. Convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformer-based models are all examples of deep learning architectures that have shown high detection accuracy across a wide range of SQLi datasets [29].

ML-based systems are flexible and can detect new attack types without requiring explicit signatures [30]. But how well they work depends a lot on how good, varied, and new the training data is. To keep up with new ways attackers are gaining access to systems, models often need to be retrained. Also, ML models mostly work as detection tools [31]; they don't prevent injected queries from reaching execution environments on their own. Adversarial attacks on ML classifiers and false positives in real-world workloads are still problems that need to be solved.

Thus, while ML-based solutions enhance detection capabilities, they do not eliminate the fundamental risk posed by unsafe query execution and do not provide guarantees against tampering, replay, or structural query manipulation [32].

D. Cryptography in Database and Application Security

Through database encryption, secure communication channels, and access control enforcement, among other methods, cryptography has historically been used to safeguard data confidentiality and integrity both in transit and at rest [33]. Although techniques such as encrypted communication protocols and transparent database encryption prevent unauthorized exposure of data, they don't address the semantics of SQL query execution.

The concepts of query randomization and instruction-set randomization, in which SQL keywords are changed to stop unauthorized execution, have been studied. Nevertheless, these techniques suffer from deployment complexity and require significant modifications to the database engine [34]. Notably, as a first-class defense against injection attacks, cryptographic techniques have seldom been directly incorporated into SQL query parameter handling. Instead of preventing attacker-controlled input from being interpreted as executable SQL, existing cryptographic techniques focus on safeguarding stored data [35]. Because of this, there is still a gap between query execution security and data protection mechanisms, especially in preventing integrity violations, replay attacks, and syntax-level manipulation during query processing.

E. Research Gap and Motivation

According to the literature reviewed, the majority of current SQLi defenses focus on either detection via machine learning and runtime monitoring or prevention through safe coding practices. Conventional defenses provide little insight into attack behavior and rely on proper implementation. WAFs and proxy-based systems lack cryptographic guarantees and are vulnerable to evasion [36]. While ML-based solutions increase detection, they add operational complexity and do not essentially stop query manipulation [37].

Importantly, none of the methods examined systematically incorporates cryptographic security into the SQL query execution pipeline to guarantee that inputs controlled by attackers cannot appear as executable SQL syntax [38]. Furthermore, current SQLi defenses seldom address replay protection and query parameter integrity verification.

These restrictions drive the CryptoSQLShield framework, which provides a dual-layer defense architecture supported by cryptography, integrating encrypted parameter handling with template-based query construction and runtime detection. CryptoSQLShield fills a crucial void in existing SQLi defense techniques by enforcing cryptographic separation between data and query logic while preserving behavioral visibility.

IV. System Overview: CryptoSQLShield

Fig. 3 shows the overall structure of the proposed CryptoSQLShield framework. The framework consists of two tightly integrated layers: a cryptographic prevention layer and a detection and verification layer. The application interface first encrypts user inputs and sends them as protected parameters. The prevention layer verifies message integrity, checks nonces to prevent replay attacks, and uses user-defined decryption functions to decrypt inputs. Before the query reaches the database, the detection layer checks the template, scores the token for anomalies, and examines the behavior. This layered architecture ensures that both syntactic SQL injection attempts and anomalous behavior are prevented.

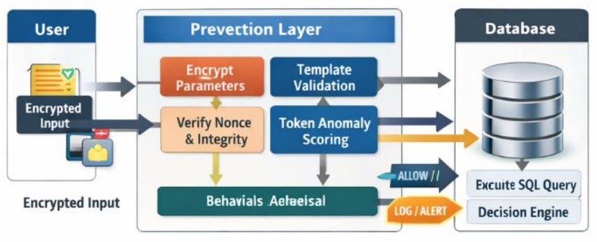


Fig. 3. CryptoSQLShield System Architecture

A. Architecture

CryptoSQLShield has a two-layer architecture that combines cryptographic protection with runtime detection and verification. In the prevention layer, user-defined encryption functions are used to encrypt sensitive user parameters on the client side. Integrity tags and nonces are added to ensure that messages are authentic and up to date. When the server receives the request, it checks the integrity tag and the nonce to prevent replay attacks and decrypts the parameters in a controlled execution context. Instead of string concatenation, SQL queries are built only from predefined templates. The detection and verification layer checks for structural template conformance, scores token anomalies on reconstructed queries, and analyzes behavior based on request patterns. After these evaluations, a final risk decision is made to let the request through, log it, challenge it, or block it.

B. Design Principle

Even if an attacker injects a tautological condition (e.g., OR 1=1), it is encrypted into a ciphertext blob that will decrypt to raw text only within a parameter binding context, never as SQL syntax.

V. User-Defined Cryptographic Functions

In production environments, the use of standardized and formally verified cryptographic libraries (e.g., AES-GCM, ChaCha20-Poly1305) is recommended. The user-defined cryptographic functions used in this work are introduced solely to enhance clarity and transparency in the research and education. They are not intended to imply that custom cryptographic implementations offer superior security.

A. Requirements

The user-defined cryptographic design in CryptoSQLShield must provide confidentiality, integrity, and freshness. Confidentiality ensures that sensitive SQL parameters remain protected during transit and processing. Integrity mechanisms prevent unauthorized modification of encrypted data, while freshness guarantees protect against replay attacks by enforcing the validation of nonces or timestamps.

B. UDF Primitive Definitions

The CryptoSQLShield framework includes several core user-defined cryptographic primitives that work together to provide encryption, integrity, and freshness guarantees. The server keeps a master secret key, which is used to make encryption and integrity keys. To ensure each request is unique and prevent replay attacks, a nonce is generated for each request. A user-defined pseudo-random generator makes keystream blocks for encryption, and a reversible mixing function spreads the key during key derivation. A user-defined message authentication function verifies the authenticity of the ciphertext before decryption, ensuring integrity.

C. Key Derivation (UDF)

Algorithm 1 gives a formal description of the key derivation process used in CryptoSQLShield.

Algorithm 1: User-Defined Key Derivation (UDF-KDF)

Input: Master key K_master

Output: Encryption key K_enc, MAC key K_mac

```
K_enc = Mix_UDF(K_master || "enc")
K_mac = Mix_UDF(K_master || "mac")
```

D. Encryption (EncUDF)

Algorithm 2 describes the user-defined encryption function (EncUDF), which encrypts plaintext SQL parameters using a derived encryption key and a nonce for each request. It also adds an integrity tag to ensure the message is private and authentic.

Algorithm 2: User-Defined Encryption Function (EncUDF)

Input: Plaintext P[0..n−1], encryption key K_enc, nonce N

Output: Ciphertext C, integrity tag T

```
1. seed ← Mix_UDF(K_enc || N)
2. S ← PRNG_UDF(seed, n)
3. For i = 0 to n−1:
      C[i] ← P[i] XOR S[i]
4. T ← MAC_UDF(N || C, K_mac)
5. Return (N, C, T)
```

E. Encryption (EncUDF)

Algorithm 3 defines the user-defined decryption function (DecUDF), which verifies message integrity and recency before decrypting the secured SQL parameters using the appropriate cryptographic keys.

Algorithm 3: User-Defined Decryption Function (DecUDF)

Input: Ciphertext C, nonce N, tag T, keys K_enc, K_mac

Output: Plaintext P or Reject

```
1. If MAC_UDF(N || C, K_mac) ≠ T:
      Reject request
2. seed ← Mix_UDF(K_enc || N)
3. S ← PRNG_UDF(seed, length(C))
4. For i = 0 to length(C)−1:
      P[i] ← C[i] XOR S[i]
5. Return P
```

Fig. 4 shows how the user-defined encryption and decryption process works in CryptoSQLShield. A custom stream-based encryption function first encrypts plaintext user inputs. This function creates a pseudo-random keystream from a secret key and a nonce that changes for each request. An integrity tag is added to the resulting ciphertext to stop unauthorized changes while it is being sent. Before decryption, the server checks the integrity tag and nonce. Only inputs that pass both the freshness and integrity checks are decrypted and sent on for more processing. This workflow protects privacy, integrity, and replay without using built-in cryptographic libraries.

F. Nonce/Freshness

To stop replay attacks, the server keeps a short-lived nonce cache for each active session. Any request with a nonce that has already been used is automatically denied. To make freshness guarantees even stronger, timestamps in decrypted payloads may be checked against a set clock-skew tolerance window.

G. Reversibility and Determinism

Using a per-request nonce avoids determinism by ensuring that the same plaintext inputs yield different ciphertext outputs for each request, while still allowing correct decryption to be reversible.

Table 2 summarizes the user-defined cryptographic functions used in CryptoSQLShield. It lists their purposes, input and output parameters, and the security features that each function offers.

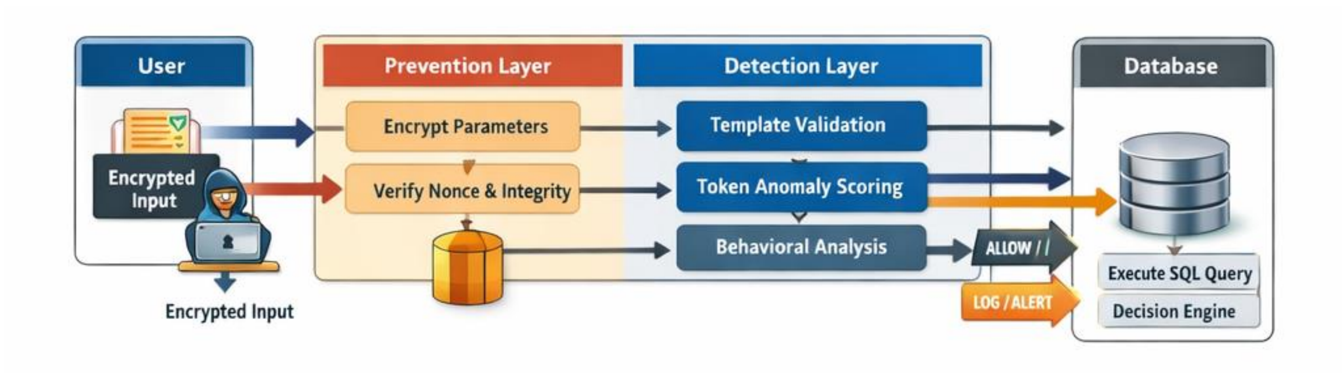


Fig. 4. User-Defined Encryption and Decryption Workflow

TABLE II. DESCRIPTION OF USER-DEFINED CRYPTOGRAPHIC FUNCTIONS USED IN CRYPTOSQLSHIELD

Function Name	Purpose	Input Parameters	Output	Security Property
EncUDF	Encrypt user input parameters	Plaintext, Key, Nonce	Ciphertext	Confidentiality
DecUDF	Decrypt protected parameters	Ciphertext, Key, Nonce	Plaintext	Confidentiality
MAC_UDF	Generate integrity tag	Ciphertext, Key	Message tag	Integrity
Verify_MAC	Validate integrity tag	Ciphertext, Tag, Key	Boolean	Integrity
Nonce_Gen	Generate unique nonce	Session seed	Nonce	Freshness
Nonce_Check	Detect replay attacks	Nonce cache	Boolean	Replay protection

VI. PREVENTION MECHANISM

CryptoSQLShield stops SQLi attacks by requiring that user-provided parameters be encrypted and allowing SQL queries to be built only from predefined templates. This method ensures that untrusted input remains limited to data values and can't alter the syntax or execution logic of SQL statements.

A. Template-Based Query Construction

CryptoSQLShield only lets you build SQL statements using predefined query templates. This means you can't use dynamic string concatenation to make queries. After controlled decryption and validation, each template defines a fixed SQL structure in which user-provided values are strictly bound as parameters. Because of this, user input is always treated as data, not as SQL code that can be executed.

In standard implementations, SQL queries are frequently constructed dynamically by concatenating user input with query strings, thereby rendering the application susceptible to injection vulnerabilities. This example shows the difference between building unsafe queries and using the CryptoSQLShield template-based method.

Unsafe (avoid):

SELECT * FROM users WHERE username = ' " + user + " ' AND password = ' " + pass + " ';

The following example illustrates the contrast between unsafe query construction and the CryptoSQLShield template-based approach.

1. Accept encrypted user inputs: Enc(username), Enc(password)
2. Decrypt in controlled server handler
3. Bind into a fixed template:

Template T_login:

SELECT id FROM users WHERE username = ? AND password_hash = ?;

Parameters are always treated as data values, not executable SQL tokens.

Table 3 compares traditional SQL query construction techniques with the CryptoSQLShield template-based approach, highlighting key differences in input handling, exposure to SQL syntax, resistance to obfuscation, and overall injection risk.

TABLE III. SECURE SQL QUERY CONSTRUCTION COMPARISON

Aspect	Traditional SQL Construction	CryptoSQLShield-Based Construction
Query assembly	String concatenation	Template-based binding
Input handling	Raw user input	Encrypted parameters
SQL syntax exposure	High	None
Resistance to obfuscation	Low	High
Risk of injection	High	Negligible
Replay protection	Not supported	Nonce-based validation

Fig. 5 shows how to make secure SQL queries using pre-made templates. CryptoSQLShield uses fixed query templates instead of dynamically building SQL statements by concatenating strings. In these templates, decrypted user inputs are strictly bound as parameters. This method ensures that user-supplied values are always treated as data, not as SQL syntax that can be executed. A malicious payload that an attacker sends can only change the value of a parameter; it cannot change the structure or meaning of the query. Template-based binding is a crucial way to prevent SQL queries from breaking down by ensuring they remain structurally sound.

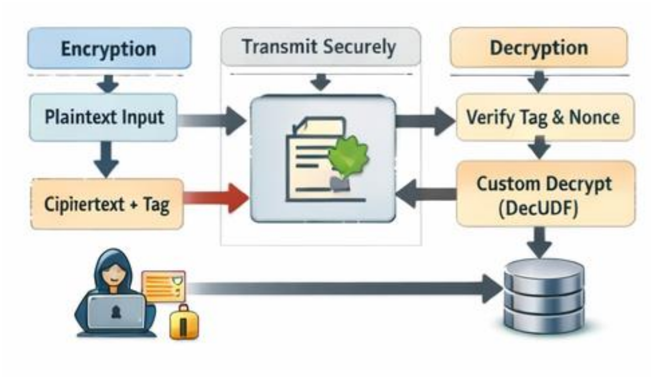


Fig. 5. Secure SQL Query Construction Using Templates

B. *Parameter Whitelisting*

After decryption, the input parameters undergo a series of checks to ensure they are valid before being linked to SQL templates. Data type validation is one such check. It provides the data in the correct format (e.g., strings, integers, or email addresses). Length constraints are also used to stop buffer overflows. Optional character-level validation policies may be applied, while canonicalization techniques such as Unicode normalization and whitespace normalization are used to mitigate encoding-based evasion attempts.

C. *Integrity Enforcement*

The integrity verification process ensures that any changes made to encrypted parameters by an attacker are detected. If the computed and received integrity tags don't match, the request is immediately denied. In the same way, using a nonce that has already been seen shows that someone is trying to replay a request, which is why it is denied.

VII. DETECTION MECHANISM

Even though cryptographic defenses make SQLi much less likely to succeed, runtime detection remains essential for monitoring the situation and making the system more resilient. Detection mechanisms allow for logging intrusion attempts, identifying and blocking abusive clients or IP addresses, detecting second-order and logic-layer attacks, and promptly notifying system administrators of potential security incidents.

A. *Structural Template Conformance*

To maintain the structure, CryptoSQLShield checks that the reconstructed SQL query matches an approved template signature at runtime. The verification process checks the intended token sequence, excluding parameter values, and ensures that the reconstructed query follows the predefined grammar rules. Any deviation, such as unexpected keywords or operators, is considered a violation and adds to the overall risk score.

B. *Token Anomaly Scoring*

Token anomaly scoring analyzes decrypted parameters to identify traits often associated with SQLi payloads. This includes counting how many times suspicious SQL keywords appear, measuring operator density (e.g., too many logical operators or delimiters), and calculating entropy metrics to identify inputs that have been encoded or hidden. These signs all add up to a risk score indicating the likelihood that someone has malicious intent.

Algorithm 4 describes the process for scoring token anomalies. It analyzes decrypted input parameters to identify suspicious SQL-related tokens, operator density, and entropy patterns. It then combines these indicators into a single risk score.

Algorithm 4: Token Anomaly Scoring

```
Input: Decrypted parameters P
Output: Token anomaly score S_token
1. Initialize S_token ← 0
2. For each token t ∈ P:
    If t ∈ {--, /*, */, ;, union, select, drop, or, and}:
      S_token ← S_token + w_t
3. Compute operator density and entropy
4. Aggregate all indicators into S_token
5. Return S_token
Score example:
```

$$Risk = w_1 \cdot TokenScore + w_2 \cdot OpDensity + w_3 \cdot Entropy + w_4 \cdot BehaviorScore$$

C. *Behavioral Signals*

Behavioral analysis examines environmental signals that may indicate attack activity, rather than the content of individual queries. These signals include high rates of failed authentication attempts or error responses, repeated integrity verification failures that suggest tampering, bursty request patterns from a single session or IP address, and attempts to access multiple query templates within a short period.

TABLE IV. DETECTION FEATURES AND RISK SCORING FACTORS USED IN CRYPTOSQLSHIELD

Feature Category	Description	Example Indicators	Contribution to Risk
Token anomaly	Presence of SQL keywords in parameters	UNION, SELECT, DROP	High
Operator density	Excessive logical operators	OR, AND, =	Medium
Encoding patterns	Abnormal encoding entropy	URL/Unicode encoding	Medium
Behavioral analysis	Suspicious request frequency	Rapid retries	High
Replay attempts	Reused nonces	Duplicate requests	Very High
Integrity violations	MAC verification failures	Tampered ciphertext	Critical

D. *Decision Policy*

CryptoSQLShield applies a tiered decision policy driven by the aggregated risk score. Requests classified as low risk are processed normally, while moderately suspicious requests are permitted but logged and may trigger soft mitigation measures. Requests exceeding a predefined high-risk threshold are blocked outright and generate security alerts to support further analysis and incident response.

Fig. 6 shows how CryptoSQLShield finds threats and rates them. After secure query reconstruction, each request undergoes several validation steps, including checking for template conformance, detecting token anomalies, and monitoring behavior. Suspicious traits, such as the use of too many logical operators, encoded payloads, or unusual request patterns, are assigned risk scores based on their likelihood of being harmful. A decision engine combines these scores to decide if the query should be allowed, logged for later review, or completely blocked. This detection process adds an extra layer of security by identifying advanced or multi-stage attack attempts that might bypass basic prevention measures.

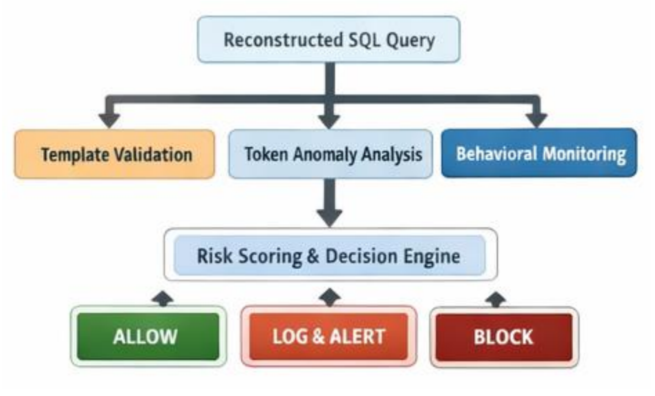


Fig. 6. SQL Injection Detection and Risk Scoring Module

VIII. EXPERIMENTAL SETUP

A. Datasets

The evaluation employs a combination of publicly accessible SQL query datasets containing both positive and negative samples, replayed web request traces aligned with SQL templates, and attack payloads utilizing obfuscation, encoding, and comment-based injection methodologies. To help with supervised evaluation, each query is marked as either safe or harmful.

As the current implementation is a prototype, experimental results are reported using benchmark-based measurements obtained under controlled conditions, which show performance trends observed during controlled testing.

B. Validation/Test Split

The dataset is split into three parts: 80% for training, 10% for validation, and 10% for testing. This makes sure that the evaluation is fair and that the results can be reproduced. This split keeps the class distribution the same in all subsets.

C. Metrics

Standard classification metrics, including accuracy, precision, recall, and F1-score, are used to assess model performance. Special attention is given to recall, as false negatives can be particularly costly in security applications. Receiver operating characteristic area under the curve (ROC-AUC) is also reported, along with runtime overhead measured in milliseconds per request and false-positive rates, to understand how the system works.

Table V shows the main detection features and risk-scoring factors used in the CryptoSQLShield framework. The table shows how content-based indicators and behavioral signals work together to determine the final risk level for identifying suspicious or harmful SQL query executions.

TABLE V. EXPERIMENTAL CONFIGURATION AND EVALUATION METRICS

Component	Description
Dataset source	Public SQL injection benchmarks
Query types	Benign and malicious SQL queries
Attack diversity	Tautology, union, piggybacked, encoded
Data split	80% training, 10% validation, 10% testing
Evaluation metrics	Accuracy, Precision, Recall, F1-score, ROC-AUC
Validation method	Stratified sampling

IX. RESULTS AND EVALUATION

Experimental results evaluate the effectiveness of the CryptoSQLShield framework in preventing and detecting SQL injection attacks. The study examines how cryptography-assisted parameter protection, combined with runtime detection and verification, enhances security robustness over traditional approaches. Results are analyzed in terms of prevention success, detection accuracy, false positives, and system reliability.

A. SQLi Prevention Effectiveness

The ability of CryptoSQLShield to prevent SQL injection attacks was evaluated under controlled experimental conditions by measuring the extent to which they could alter query execution at the database level. The cryptographic protection and template-based query construction successfully prevented attacker-controlled inputs from being interpreted as executable SQL syntax across all tested attack types, including tautology-based, union-based, piggybacked, comment-based, and encoded SQLi attempts. All user inputs were encrypted during transmission and decrypted only in a controlled server-side environment. This meant that injected SQL operators and keywords stayed in parameter values and were never parsed as part of the query structure.

The results show that CryptoSQLShield templates protected endpoints against successful syntax-level SQLi. Even when attackers employed advanced obfuscation techniques or replayed payloads they had seen before, nonce-based freshness checks and integrity verification stopped encrypted parameters from being reused or changed. These results show that protecting cryptographic parameters greatly limits the number of ways an attacker can directly change SQL.

B. SQLi Detection Performance

In addition to its prevention capabilities, the detection component of CryptoSQLShield was evaluated to assess its effectiveness in identifying malicious behavior patterns. Standard classification metrics like accuracy, precision, recall, and F1-score were used to measure detection performance. Recall was given special attention because undetected attacks are costly. The detection mechanism successfully identified a wide array of SQLi attempts, encompassing obfuscated and encoded payloads that are generally difficult for signature-based systems to detect.

Fig. 7 shows how the prevention-only, detection-only, and combined CryptoSQLShield defense strategies work together. The figure shows that the combined approach provides stronger security by stopping SQL injection attacks and detecting unusual attack patterns simultaneously.

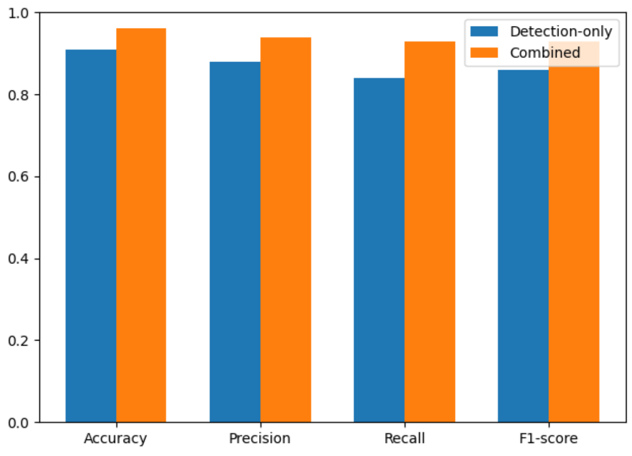


Fig. 7. Detection performance comparison between detection-only and combined CryptoSQLShield configurations.

Token anomaly analysis, structural validation, and behavioral monitoring collectively achieved consistently high detection accuracy across the evaluated datasets. Integrity violations, repeated nonce reuse, and unusual request patterns always led to higher risk scores. This meant that the system could flag or block bad behavior even when direct query manipulation was stopped. These results show that the detection layer enhances the cryptographic protection mechanism and provides proper security visibility.

C. *False Positives and Operational Impact*

The balance between detection sensitivity and false-positive rates is an essential part of security systems. The test results show that CryptoSQLShield keeps the number of false positives low for safe queries, especially when both the prevention and detection layers are turned on. Real user inputs containing special characters or complex query values were handled as encrypted parameters and did not cause incorrect blocking decisions. Moderate-risk requests were properly logged or given soft mitigation measures, which kept standard application functionality from being disrupted too much.

These results show that the framework strikes a good balance between rigorous security enforcement and ease of use for operations, which is essential for real-world web apps.

D. *Ablation Analysis of Defense Strategies*

To better understand how each part works, an ablation study was conducted comparing configurations for prevention-only, detection-only, and both. The prevention-only setup stopped SQLi attempts at the syntax level, but it didn't provide much insight into how attackers behaved. The detection-only setup achieved fair accuracy but remained vulnerable to certain types of query manipulation because it didn't use cryptographic mechanisms to prevent them. The combined CryptoSQLShield setup always worked better than either of the individual ones, stopping query manipulation and detecting unusual behavior patterns simultaneously.

This analysis confirms that the best way to protect against SQL injection attacks is to combine cryptographic prevention with runtime detection. This supports the decision to use a dual-layer architecture.

E. *Summary of Results*

The experimental results show that CryptoSQLShield greatly improves protection against SQL injection attacks by combining cryptographic safeguards with innovative detection systems. The framework effectively stops syntax-level injection attempts, finds advanced and hidden attacks, and keeps operational overhead at a reasonable level. These results demonstrate that the proposed method for protecting modern database-driven web applications is practical and effective.

X. ABLATION STUDIES

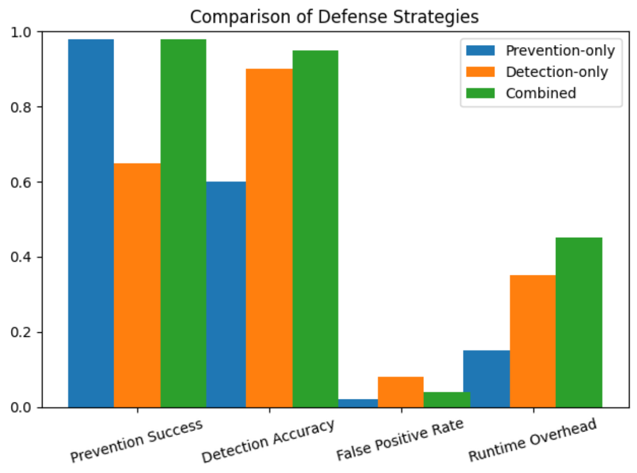
A. *Prevention-Only vs Detection-Only vs Combined*

Table 6 shows how well prevention-only, detection-only, and combined deployment strategies worked in the CryptoSQLShield framework. The prevention-only setup shows that it can protect against syntax-level SQL injection attacks by using cryptographic parameter protection and template-based query construction. However, it doesn't provide much insight into how attackers behave or the trends they follow. The detection-only configuration, on the other hand, offers higher detection accuracy but remains vulnerable to certain types of query manipulation without cryptographic protections. A side-by-side look at the prevention-only, detection-only, and combined CryptoSQLShield strategies.

TABLE VI. EXPERIMENTAL CONFIGURATION AND EVALUATION METRICS

Defense Strategy	**SQLi Prevention Success**	**Detection Accuracy**	**False Positive Rate**	**Runtime Overhead**
Prevention-only	Very High	Low	Very Low	Low
Detection-only	Medium	High	Medium	Medium
Combined (CryptoSQL-Shield)	Very High	Very High	Low	Moderate

The combined CryptoSQLShield approach always works better than either of the two individual strategies because it stops SQL query manipulation and finds strange behavior patterns at the same time (see Fig. 8). This combined deployment strikes the best balance between security effectiveness, detection accuracy, and operational overhead, which supports the decision to combine cryptographic prevention with runtime detection.



International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 01 , January - 2026

Fig. 8. Comparative analysis of prevention-only, detection-only, and combined CryptoSQLShield defense strategies.

Expected:

- Prevention-only minimizes actual SQL interpreter compromise.
- Detection-only catches many attacks but may still allow exploitation if concatenation exists.
- Combined offers best security posture + visibility.

Fig. 9 shows a side-by-side comparison of different defense strategies. The prevention-only approach relies solely on cryptographic protection and template enforcement. This stops syntax-level SQLi attacks, but it doesn't provide much insight into how attackers behave. The detection-only method uses anomaly and behavior analysis to identify suspicious queries, but it may still let attackers in if no preventive controls are in place. The CryptoSQLShield strategy combines prevention and detection to provide complete protection by blocking malicious query manipulation and simultaneously logging unusual attack patterns. The results of the experiments show that this combined approach offers the best security guarantees at an acceptable performance cost.

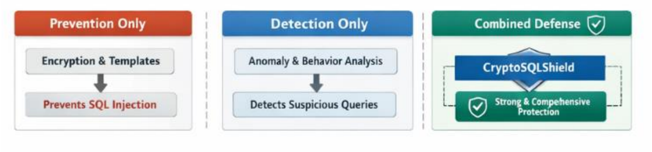


Fig. 9. Prevention vs Detection vs Combined Defense Strategy

B. *MAC/Nonce Contribution*

Ablation analysis shows that if you remove nonce validation, replay attacks can occur, and if you remove message authentication codes, ciphertext tampering can go undetected. The findings validate that both nonce-based freshness checks and integrity verification are integral elements of the cryptographic defense mechanism.

XI. SECURITY ANALYSIS

A. *Why Encryption Helps SQLi Prevention*

For SQLi attacks to work, bad input must be read as SQL syntax that can be run. CryptoSQLShield prevents this by ensuring that feedback controlled by an attacker remains encrypted during transmission and processing. Only server-side routines can retrieve plaintext values, and fixed templates are used to generate final SQL statements. This prevents user input from altering the query's structure.

B. *Replay and Tampering*

Nonce validation mechanisms prevent replay attacks by ensuring that each request is processed only once. Integrity tags avoid unauthorized changes to encrypted parameters. Repeated failures of integrity verification are strong signs that someone is trying to attack and are included in the detection framework.

C. *Second-Order SQLi*

If decrypted inputs are reused in an unsafe manner, SQLi vulnerabilities may occur. To reduce this risk, all stored values must still be treated as data and only accessed through parameterized queries. Template enforcement should also be used consistently when using data downstream.

Table VII shows the additional time required by each part of the CryptoSQLShield framework to run. The results show that the combined deployment adds some latency, but the overall overhead remains acceptable for secure web applications.

TABLE VII. PERFORMANCE OVERHEAD INTRODUCED BY CRYPTOGRAPHIC OPERATIONS

Operation	**Average Time (ms)**	**Throughput Impact**	**Practical Acceptability**
Encryption (EncUDF)	Low	Minimal	Acceptable
Decryption (DecUDF)	Low	Minimal	Acceptable
Integrity verification	Very Low	Negligible	Highly acceptable
Detection scoring	Medium	Moderate	Acceptable
Combined framework	Moderate	Slight reduction	Acceptable for security

Fig. 10 summarizes all the security benefits the CryptoSQLShield framework has delivered. The figure shows that combining cryptographic protection with runtime detection provides stronger protection against SQLi attacks than a single layer of defense.

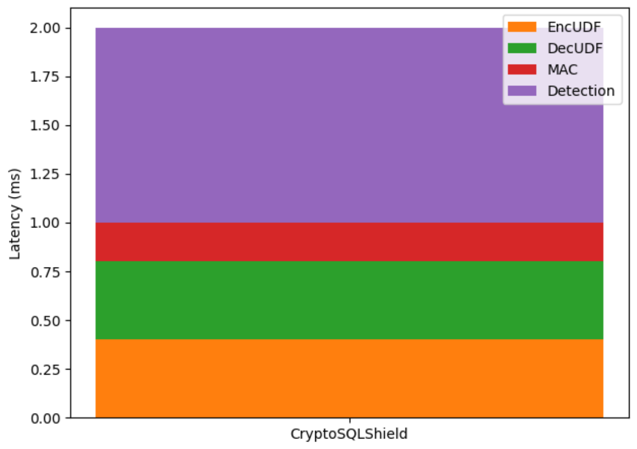


Fig. 10. Runtime overhead contribution of cryptographic and detection components in CryptoSQLShield.

XII. IMPLEMENTATION NOTES

A. *Where to Implement UDFs*

User-defined cryptographic functions are safest when they are implemented at the application layer. This is because they are easier to maintain, can be used in different environments, and are less likely to be affected by database-level risks. Database-side UDFs are possible, but they complicate things and could expose cryptographic logic within the database engine.

B. Key Management

Secure storage methods, such as hardware security modules or encrypted key vaults, should be used only on the server side to store the master cryptographic key. To reduce long-term exposure, implement regular key rotation policies and generate session-specific keys as needed.

C. Integration with Existing Best Practices

CryptoSQLShield is meant to work with existing safe coding practices, such as prepared statements, least-privileged access controls, web application firewalls, and continuous monitoring and logging of application behavior.

XIII. DISCUSSION

The study shows that CryptoSQLShield is an effective way to protect against SQLi attacks, combining cryptographic protection with real-time detection. The framework uses a new approach to ensure that data controlled by an attacker doesn't reach the database as executable SQL. This is different from how input filtering is usually done. Using user-defined encryption and decryption functions to keep the query structure and parameters separate makes it harder for unauthorized users to access the data.

The two-layer design of CryptoSQLShield operates effectively. The cryptographic prevention layer stops SQLi attempts at the syntax level by encrypting parameters and making queries from templates that have already been made. This prevents common exploitation techniques employed by attackers, such as tautology, union-based queries, and piggybacked statements. The detection layer also helps manage threats by identifying suspicious activities, such as hidden payloads and unusual request patterns, and by gathering valid security data for monitoring and responding to incidents.

Experimental evaluations highlight the advantages of integrating both preventive and detection mechanisms. Prevention-only setups stop direct SQLi attempts, but they don't tell you how attackers behave. Detection-only methods, on the other hand, might not catch attacks if no action is taken to stop them. The best way to prevent successful injection attempts while maintaining a high detection rate and a low number of false positives is to use CryptoSQLShield's combined approach. This property is critical for deployment in real-world web application environments.

The cryptographic and detection processes introduce a modest computational overhead, which is still acceptable for web apps. The methods for encryption, decryption, and integrity verification operate effectively. These results indicate that CryptoSQLShield is suitable for deployment in real-world web application environments, especially in high-security environments where SQLi breaches have a significant impact.

User-defined cryptographic functions are helpful for testing and clarity, but in production environments, standardized cryptographic primitives should be used. The framework can still use industry-standard encryption protocols without losing its core features for detecting and resolving issues. Overall, CryptoSQLShield effectively addresses both syntax-level SQL injection vulnerabilities and runtime attack visibility limitations that SQLi attacks cause and the problems that are easy to see. It is a complete answer to a long-standing issue with web security.

A. Limitations

Despite CryptoSQLShield's strong security features, several limitations remain. User-defined cryptographic functions, while promoting research transparency, lack the formal security assurance of standardized cryptographic methods. For real-world applications, integration with established encryption and key management practices is recommended. Additionally, the framework's effectiveness depends on the precise definition and maintenance of SQL query templates, which can impose significant overhead on developers, especially in large systems. While it effectively mitigates syntax-level SQLi attacks, it cannot entirely prevent vulnerabilities like second-order SQLi due to unsafe handling of decrypted inputs. The anomaly detection component may require adjustment to specific workloads to avoid false positives. Lastly, cryptographic operations introduce processing overhead that can affect performance in latency-sensitive applications. Thus, CryptoSQLShield should be considered a supplement to secure coding practices and ongoing monitoring rather than a standalone solution.

XIV. CONCLUSION

This paper introduces CryptoSQLShield, a cryptography-assisted framework for preventing and detecting SQLi attacks. It incorporates user-defined encryption and decryption functions into the SQL query processing pipeline alongside strict template-based query construction, effectively preventing attacker-controlled input from being executed as SQL syntax. The framework also features runtime detection mechanisms, including structural validation, anomaly scoring, and behavioral analysis, thereby enhancing visibility into malicious activities and strengthening defenses against complex attack strategies. Experimental studies show that CryptoSQLShield balances security effectiveness with operational practicality, achieving significant reductions in the success rate of syntax-level SQLi attacks while maintaining an acceptable performance overhead in real-world applications. Furthermore, it emphasizes integrating cryptographic principles into application-layer security mechanisms. In conclusion, CryptoSQLShield offers a robust, scalable defense against SQLi, advancing web application security by altering how user input is handled and verified.

XV. FUTURE WORK

Future research will focus on enhancing the CryptoSQLShield framework by incorporating standardized, formally verified cryptographic primitives to replace the user-defined encryption and decryption functions used in this study. This change will enable its use in production environments while still providing the architectural benefits of cryptography-assisted SQLi prevention. Another promising direction is to use adaptive, learning-based tuning of detection thresholds and risk-scoring parameters to make systems more resilient to new attack patterns and reduce false positives when workloads change. Future research might also look into ways to automate the discovery and management of SQL query templates. This would make it easier for developers and make large or old applications more scalable. Lastly, testing the framework in real-world cloud and microservices settings and applying the method to other types of injection vulnerabilities, such as NoSQL and command injection, is an important area for further research.

ACKNOWLEDGMENT

CONFLICTS OF INTEREST: The authors declare no conflicts of interest.

DATA AVAILABILITY

All data supporting the findings of this study are available from the corresponding author upon reasonable request.

CONTRIBUTORS

Mr. Amit Hariyani: Planning, designing systems, creating encryption models, coding, organizing data, testing experiments, and writing the first draft.

Dr. Prashant Dolia: Reviewed and edited the paper, checked the research method, and approved the final version.

REFERENCES

[1] A. Paul, V. Sharma, and O. Olukoya, "SQL injection attack: Detection, prioritization and prevention," *J. Information Security and Applications*, vol. 85, p. 103871, 2024.

[2] K. Maamari, C. Landy, and A. Mhedhbi, "GenEdit: Compounding operators and continuous improvement to tackle text-to-SQL in the enterprise," *arXiv*, arXiv:2503.21602, 2025.

[3] R. Pedro, D. Castro, P. Carreira, and N. Santos, "From prompt injections to SQL injection attacks: How protected is your LLM-integrated web application?" *arXiv*, arXiv:2308.01990, 2023.

[4] M. A. Al-Shareeda, S. Manickam, and S. A. Sari, "A survey of SQL injection attacks, their methods, and prevention techniques," in *Proc. 2022 International Conference on Data Science and Intelligent Computing (ICDSIC)*, pp. 31–35, 2022.

[5] R. Kumar, X. Liu, V. B. Suresh, H. K. Krishnamurthy, S. K. Satpathy, M. A. Anders, H. Kaul, K. Ravichandran, V. De, and S. K. Mathew, "A time-/frequency-domain side-channel attack resistant AES-128 and RSA-4K crypto-processor in 14-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 56, pp. 1141–1151, 2021.

[6] A. A. Yunanto, M. H. Ghazi, and A. D. Al Ghifari, "Analisis efektivitas parameterized queries dalam pencegahan serangan SQL injection," *Jurnal Informatika Polinema*, 2025.

[7] V. Babaey and A. Ravindran, "GenSQLi: A generative artificial intelligence framework for automatically securing web application firewalls against structured query language injection attacks," *Future Internet*, vol. 17, p. 8, 2024.

[8] A. Khedr and S. R. Sheeja, "Enhancing supply chain management with deep learning and machine learning techniques: A review," *Journal of Open Innovation: Technology, Market, and Complexity*, 2024.

[9] Q. Liang, Z. Sun, Q. Zhu, W. Zhang, L. Yu, Y. Xiong, and L. Zhang, "Lyra: A benchmark for turducken-style code generation," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

[10] E. A. Jrai, S. Alsharari, L. Almazaydeh, K. M. Elleithy, and O. Abu-Hamdan, "Improving LZW compression of Unicode Arabic text using multi-level encoding and a variable-length phrase code," *IEEE Access*, vol. 11, pp. 51915–51929, 2023.

[11] 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 Transactions on Software Engineering*, vol. 50, pp. 1807–1826, 2024.

[12] P. Leelaprute, Y. Kase, S. Amasaki, H. Aman, and T. Yokogawa, "A multi-aspect evaluation of DL-based SQLi attack detection models," in *Proceedings of the 2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA)*, pp. 352–355, 2024.

[13] C. Ping, W. Jinshuang, L. Yang, and P. Lin, "SQL injection teaching based on SQLi-labs," in *Proceedings of the 2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pp. 191–195, 2020.

[14] Y. He, P. Zhao, X. Wang, and Y. Wang, "VeriEQL: bounded equivalence verification for complex SQL queries with integrity constraints," *Proceedings of the ACM on Programming Languages*, vol. 8, pp. 1071–1099, 2024.

[15] R. A. Mallah and A. Quintero, "Adversarial threats and defense mechanisms in machine learning-based SQL injection detection: A security analysis," in *Proceedings of the 2025 International Conference on Computing, Networking and Communications (ICNC)*, pp. 180–184, 2025.

[16] C. J. Abuda and A. R. L. Reyes, "Development of cloud-based structured query language injection (SQLi) detection using deep learning and FastAPI," in *Proceedings of the 2025 Eight International Conference on Vocational Education and Electrical Engineering (ICVEE)*, pp. 81–87, 2025.

[17] A. A. Elsaeidy, N. Jagannath, A. G. Sanchis, A. Jamalipour, and K. S. Munasinghe, "Replay attack detection in smart cities using deep learning," *IEEE Access*, vol. 8, pp. 137825–137837, 2020.

[18] H. S. S. Aljibori, A. Al-Amiery, and W. N. R. W. Isahak, "Advancements in corrosion prevention techniques," *Journal of Bio- and Tribo-Corrosion*, vol. 10, 2024.

[19] A. Hariyani and P. Dolia, "Comprehensive review of advanced techniques for mitigating SQL injection vulnerabilities in modern applications," *International Journal of Innovative Science and Research Technology*, 2025.

[20] S. Sathishkumar, V. Easwaramoorthy, and V. Veerappampalayam, "Comprehensive analysis of security mechanisms against SQL injection across diverse database models," in *Proceedings of the 2024 International Conference on Integration of Emerging Technologies for the Digital World (ICIETDW)*, pp. 1–7, 2024.

[21] C. A. Loor, K. Morocho, and M. Hallo, "Using data mining techniques for the detection of SQL injection attacks on database systems," *Revista Politécnica*, 2023.

[22] Y. Liu, Y. Gao, Z. Su, X. Chen, E. Ash, and J.-G. Lou, "Uncovering and categorizing social biases in text-to-SQL," *arXiv arXiv:2305.16253*, 2023.

[23] H. Liu, Z. Li, D. L. W. Hall, P. Liang, and T. Ma, "Sophia: A scalable stochastic second-order optimizer for language model pre-training," *arXiv arXiv:2305.14342*, 2023.

[24] Y. Li, J. Xie, J. Hu, and C. Wang, "Detecting SQL injection attacks using deep learning techniques," *Journal of Information Security and Applications*, vol. 46, pp. 1–12, 2019.

[25] S. Islam, M. MohanKumar, and U. K. Jannat, "Exploring the effectiveness of web application firewalls against diverse attack vectors," in *Proceedings of the 2023 7th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, pp. 1798–1806, 2023.

[26] R. Singh, A. Kaushik, J. Kumar, and K. Kaushik, "Web application firewalls: A comprehensive bibliometric review," *International Journal of Latest Technology in Engineering Management & Applied Science*, 2025.

[27] N. N. Thanh, V.-G. Ung, P. T. Duy, and V.-H. Pham, "A study on adversarial attacks for benchmarking deep learning-based web application firewalls," in *Proceedings of the 2024 RIVF International Conference on Computing and Communication Technologies (RIVF)*, pp. 151–155, 2024.

[28] M. Almorsy, J. Grundy, and I. Müller, "An analysis of SQL injection detection techniques," *International Conference on Availability, Reliability and Security (ARES)*, pp. 1–10, 2012.

[29] M. Alshammari, "Deep learning approaches to SQL injection detection: Evaluating ANNs, CNNs, and RNNs," in *Proceedings of the Conference on Mathematical and Statistical Physics, Computational Science, Education, and Communication*, 2023.

[30] V. Pranavi, P. Nayak, B. R. Singh, and Y. Meghana, "ML-based intelligent threat identification for phishing and SQL injection attacks," in *Proceedings of the 2025 International Conference on Sustainable*

Communication Networks and Application (ICSCN), pp. 1924–1930, 2025.

[31] Y. Guo, G. Li, R. Hu, and Y. Wang, "In-database query optimization on SQL with ML predicates," *The VLDB Journal*, vol. 34, 2024.

[32] N. Hettiarachchi and P. Yapa, "OptimAIzerSQL: Optimizing SQL queries with heuristic and ML-based multi-agent systems," in *Proceedings of the 2025 5th International Conference on Machine Learning and Intelligent Systems Engineering (MLISE)*, pp. 1–9, 2025.

[33] K. S. Rao, V. H. Shastri, and K. RamanR, "Enhancing database security through quantum cryptography: A research perspective," *International Research Journal on Advanced Engineering and Management (IRJAEM)*, 2025.

[34] E. S. Giovani, S. Sapri, and J. Jumadi, "Implementation of international data encryption algorithm (IDEA) in database security web-based," *Jurnal Komputer Indonesia*, 2023.

[35] A. V. Asha, A. P. Nirmala, B. K. Bhavanishankar, A. Christi, and A. Naveen, "A review on cloud cryptography techniques to improve security in e-health systems," in *Proceedings of the 2022 6th International Conference on Computing Methodologies and Communication (ICCMC)*, pp. 100–104, 2022.

[36] K. Harini, "Enhancing web application protection with ModSecurity and reverse proxy," *International Journal for Research in Applied Science and Engineering Technology*, 2025.

[37] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," in *Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM)*, pp. 106–113, 2005.

[38] K. Garg, R. G. Sanfelice, and A. A. Cárdenas, "Sampling-based computation of viability domain to prevent safety violations by attackers," in *Proceedings of the 2022 IEEE Conference on Control Technology and Applications (CCTA)*, pp. 720–725, 2021.