International Journal of Environmental Sciences
ISSN: 2229-7359
Vol. 11 No. 8, 2025
https://theaspd.com/index.php

Evaluating Input Validation Techniques For SQL Injection Defense

Jiho Choi [1], Taek Lee[2*], and Hoon Ko[3*]
[1,2,3]Department of Computer Science and Engineering, Sunmoon University, 70, Sunmoon-ro 221 beon-gil, Tangjeong-myeon, Asan 31460, Republic of Korea
Emails: choe31904@naver.com[1], comtaek76@sunmoon.ac.kr[2], hoonko21@sumnoon.ac.kr[3]

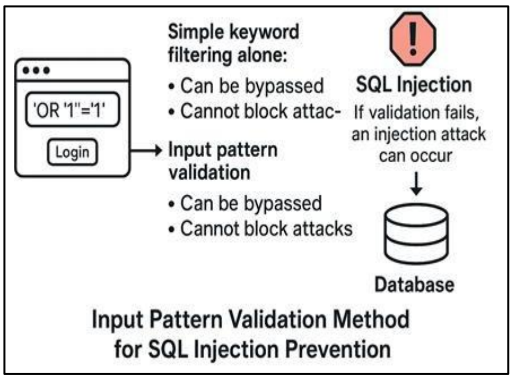
Abstract: *This study compares four SQL Injection defense techniques: input normalization, blacklist and whitelist filtering, and FSM-based context validation. Experiments using identical attack payloads show that normalization improves overall filtering accuracy, while blacklist filtering is simple to implement but vulnerable to evasion. In contrast, whitelist and FSM-based methods provide strong defensive performance but require greater implementation effort and maintenance. Overall, no single technique is sufficient on its own; instead, a multi-layered defense strategy that integrates normalization, filtering, and context validation is shown to be the most effective approach.*
Keyword: *SQL Injection, Input Normalization, Blacklist/Whitelist Filtering, FSM-based Validation, Multi-layered Defense*

1. INTRODUCTION

Web applications rely heavily on user-provided input, and insufficient validation of such input can lead to critical security vulnerabilities such as SQL Injection (SQLi) [1][2]. SQL Injection is a widely known attack technique in which an adversary injects malicious SQL statements into application input fields to access or manipulate database contents. Despite the development of numerous defensive solutions, SQLi remains one of the most frequently reported vulnerabilities across modern web systems [3]. Successful exploitation can result in personal data leakage, account compromise, unauthorized data modification, and other severe security incidents [4]. Moreover, as SQLi techniques continue to evolve through obfuscation, encoding, and structural manipulation, simple string-based filtering is increasingly ineffective at preventing sophisticated attack vectors. To address these challenges, this study focuses on input pattern validation, a core component of SQLi defense. The objective is to systematically compare and analyze the effectiveness of three major defensive strategies under a unified experimental environment [5][6][7].
First, an input normalization pipeline—comprising HTML unescaping, URL decoding, NFKC transformation, and Unicode homoglyph handling—is applied to assess how preprocessing influences the detection of obfuscated payloads.
Second, regular-expression-based Blacklist and Whitelist filtering techniques are implemented to evaluate detection accuracy, false-positive rates, bypass resistance, and inherent structural limitations.
Third, a context-aware FSM (Finite State Machine)–based validation mechanism is constructed, in which input is tokenized and examined according to syntactic and contextual state transitions. This enables identification of structural anomalies that simple regex-based filters cannot detect.
The study employs a test set consisting of 500 normal inputs and 500 SQLi payloads, allowing the quantitative comparison of these techniques with respect to detection rate, false positives, and bypass success. Through this evaluation, we demonstrate the limitations of traditional filtering approaches and highlight the defensive advantages of normalization and context-based analysis. Furthermore, we derive key design considerations for constructing a robust, multi-layered SQLi defense framework.
The remainder of this paper is organized as follows. Section 2 reviews SQL Injection attack techniques and discusses the limitations of conventional validation mechanisms. Section 3 presents the structure of input patterns and outlines the defensive models implemented in this study. Section 4 provides the experimental setup, results, and comparative analysis of normalization, regex-based filtering, and FSM-based validation methods, and Section 5 summarizes key findings and proposes future research directions.

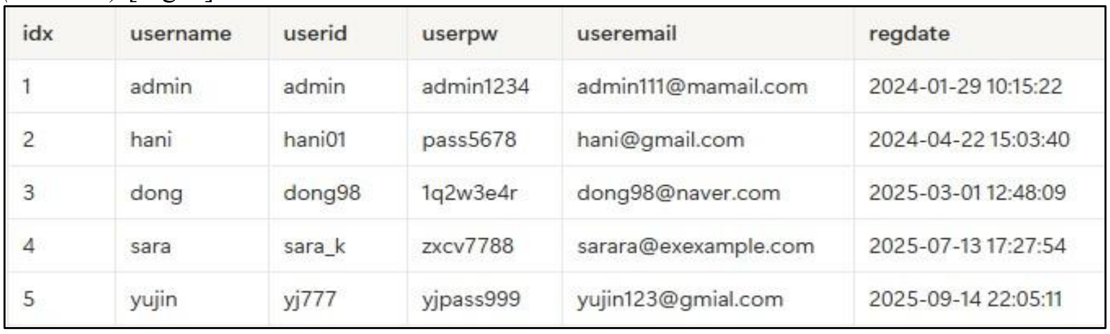
2. INPUT PATTERN VALIDATION PROBLEM FOR SQL INJECTION DEFENSE

Web applications often embed user-provided input directly into database queries. If such input is not sufficiently validated, SQL Injection vulnerabilities may arise. In this case, an attacker can inject manipulated strings such as ' OR '1'='1 into input fields to bypass authentication or to exfiltrate and modify data [Fig. 1].


'OR '1''='1'
Login
Simple keyword filtering alone:
• Can be bypassed
• Cannot block attac-
Input pattern validation
• Can be bypassed
• Cannot block attacks
SQL Injection
If validation fails, an injection attack can occur
Database
Input Pattern Validation Method for SQL Injection Prevention

[Fig. 1] Pattern Validation Problem

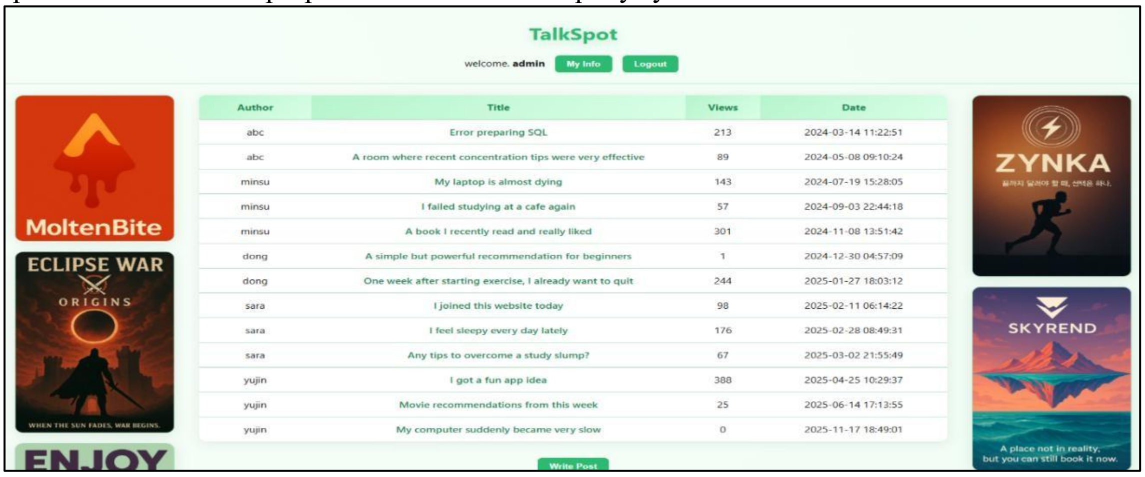
Fig. 1 conceptually illustrates how such vulnerabilities arise. An attacker injects a malicious SQL pattern into an input field—such as a login form—and if the system fails to properly validate the input, the injected string is incorporated directly into the query, causing the database to execute unintended commands [8]. Systems that rely solely on simple integrity checks cannot detect obfuscated SQL payloads and are therefore highly susceptible to bypass attempts; insufficient filtering further increases the likelihood of a successful SQL Injection attack. Accordingly, effective SQL Injection defense requires analyzing the structural patterns of input values and blocking malicious input before it reaches the database. Input pattern validation techniques are thus essential components of a robust defense mechanism [9]. Motivated by this perspective, the present study compares and experimentally evaluates several pattern-based defense strategies, including input filtering, normalization, and context-aware validation. In the experiment, a representative attack method—Union-based SQL Injection—was employed. The target system was a PHP-based bulletin board with no protective measures applied. The SQL payload used for the attack is next *'$sql = "SELECT * FROM users WHERE usersid = '$id"*. The database used for testing consisted of the following fields: *idx (int), username (varchar), usersid (varchar), userspw (varchar), usersemail (varchar), and regdate (datetime)* [Fig. 2].



idx	username	userid	userpw	useremail	regdate
1	admin	admin	admin1234	admin111@mamail.com	2024-01-29 10:15:22
2	hani	hani01	pass5678	hani@gmail.com	2024-04-22 15:03:40
3	dong	dong98	1q2w3e4r	dong98@naver.com	2025-03-01 12:48:09
4	sara	sara_k	zxcv7788	sarara@exexample.com	2025-07-13 17:27:54
5	yujin	yj777	yjpass999	yujin123@gmial.com	2025-09-14 22:05:11

[Fig. 2] Users records in the member database table

To perform the attack on the constructed database, a payload such as ' OR '1'='1 -- was used, with a trailing space added to ensure proper termination of the query syntax.



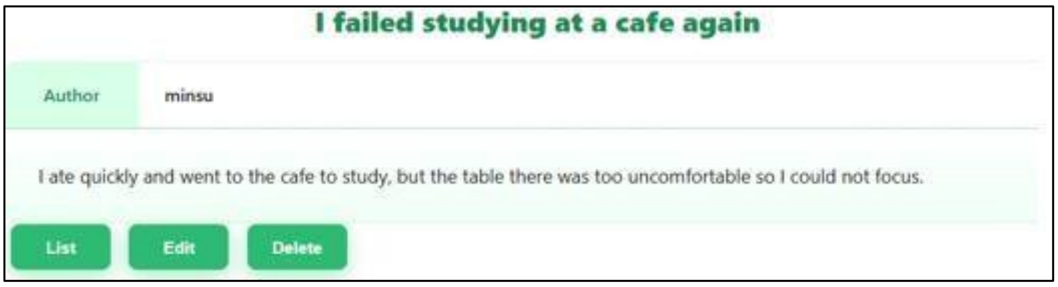
TalkSpot

welcome. **admin**

Author	Title	Views	Date
abc	Error preparing SQL	213	2024-03-14 11:22:51
abc	A room where recent concentration tips were very effective	89	2024-05-08 09:10:24
minsu	My laptop is almost dying	143	2024-07-19 15:28:05
minsu	I failed studying at a cafe again	57	2024-09-03 22:44:18
minsu	A book I recently read and really liked	301	2024-11-08 13:51:42
dong	A simple but powerful recommendation for beginners	1	2024-12-30 04:57:09
dong	One week after starting exercise, I already want to quit	244	2025-01-27 18:03:12
sara	I joined this website today	98	2025-02-11 06:14:22
sara	I feel sleepy every day lately	176	2025-02-28 08:49:31
sara	Any tips to overcome a study slump?	67	2025-03-02 21:55:49
yujin	I got a fun app idea	388	2025-04-25 10:29:37
yujin	Movie recommendations from this week	25	2025-06-14 17:13:55
yujin	My computer suddenly became very slow	0	2025-11-17 18:49:01

My Info
Logout
Write Post
MoltenBite
ECLIPSE WAR
ORIGINS
WHEN THE SUN FADES, WAR BEGINS.
ENJOY
ZYNKA
SKYREND
A place not in reality, but you can still book it now.

[Fig. 3] Compromised Bulletin Board


I failed studying at a cafe again

Author	minsu

I ate quickly and went to the cafe to study, but the table there was too uncomfortable so I could not focus.

[Fig. 4] Board Contents

When this payload is inserted into the ID input field and any arbitrary value is entered in the Password field, the query condition evaluates to True, resulting in an authentication bypass that logs in the attacker as the second row's admin account. Once the attack succeeds, the attacker gains admin-level access, enabling the creation, modification, and deletion of posts, as well as viewing posts created by other users [Fig. 3][Fig. 4].

3. INPUT PATTERN-BASED DEFENSE STRATEGY

To effectively evaluate SQL Injection defense mechanisms, it is essential to systematically compare the performance of each protection stage that processes user input. In this study, four defensive techniques—extending beyond simple string-based filtering to include input normalization and context-aware structural validation—were implemented, and their detection rates, false-positive rates, and bypass success rates were quantitatively measured [9].

First, regular-expression-based *blacklist* and Whitelist filtering methods were implemented to compare the efficiency of pattern-blocking and *allowlist*-based approaches. Although both techniques are widely used, they exhibit fundamental structural limitations that make them vulnerable to various evasion attacks. In this study, obfuscated SQL payloads were applied to evaluate the detection performance of each method.

Second, an input normalization pipeline—composed of HTML unescaping, URL decoding, NFKC normalization, and Unicode homoglyph mapping—was constructed to analyze how normalization affects the detection accuracy of filtering-based methods. The performance before and after normalization was compared to assess its impact.

Third, an FSM (Finite State Machine)–based context validation technique was implemented, in which input strings are tokenized and evaluated according to their contextual state transitions. This method was tested to determine whether it can more effectively detect evasion techniques—such as nested structures, quote breaking, and insertion of benign-looking strings—that are difficult to identify using simple regular expressions.

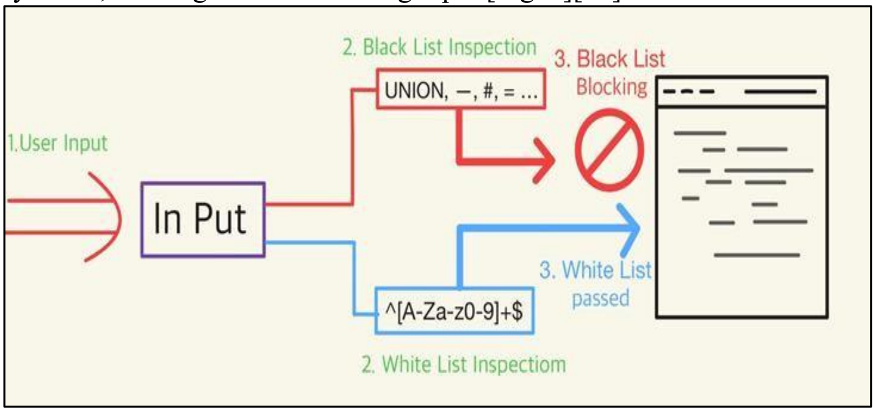
Finally, all three techniques were evaluated using the same attack dataset, and detection rates, false positives, and bypass results were measured. Based on these outcomes, the structural differences and practical defensive effectiveness of the methods were comprehensively compared and analyzed.

3-1 INPUT VALUE NORMALIZATION

The second method, input normalization, refers to the process of restoring attacker-modified payloads to a standardized form before filtering is applied. For example, when attempting to block the word "복숭아," an attacker may evade the filter by transforming the string into variants such as "복숭ㅇr," "Peach," or "복ㅅㅜㅇ아." Input normalization converts such obfuscated representations back into their canonical form, "복숭아," prior to inspection. The normalization process includes several steps—HTML entity decoding, URL decoding, Unicode homoglyph normalization, and case folding—ensuring that the semantic meaning of the input remains consistent. This preprocessing phase is effective in detecting evasion techniques such as mixed encoding, insertion of special characters, and string fragmentation, thereby compensating for the inherent limitations of simple string-based filtering [10].

3-2 REGULAR-EXPRESSION-BASED BLACKLIST & WHITELIST

The first method applies regular-expression-based Blacklist and Whitelist filtering techniques. As illustrated in the figure, both approaches examine the input at an early stage, but they differ in their operational principles. The Blacklist approach blocks any input containing prohibited words or patterns, whereas the Whitelist approach permits only predefined characters or patterns and rejects all others. In the Blacklist method, commonly used SQL Injection components—such as UNION, --, and #—are detected and blocked either on the client side or the server side. In contrast, the Whitelist method only accepts characters included in an approved set, such as numbers, alphabetic characters, and selected special symbols, filtering out all remaining input [Fig. 5][11].



[Fig. 5] BlackList·WhiteList Procedure

3-3 FSM (FINITE STATE MACHINE)

The Finite State Machine (FSM) is a context-based technique used to track how an attacker's input alters the structure of an SQL query, and it can be regarded as a form of SQL syntax validator. The FSM processes an SQL query character by character and transitions between states accordingly. The initial state is 0, and when a single quotation mark (') or double quotation mark (") is encountered, the machine transitions into a string state (state 1). In this state, keywords such as OR or AND are treated purely as text and therefore are not considered attack indicators. When comment patterns such as -- or /* are detected, the FSM transitions into a comment state (state 2), in which all subsequent characters are treated as comments until a newline or the closing */ is reached. Additional states are defined to handle other syntactic elements such as escape characters, parentheses, and operators. By leveraging this contextual information, the FSM determines whether user input modifies the structural semantics of the query. This enables effective detection of various evasion techniques—such as quote breaking, comment-based obfuscation, and keyword insertion—that cannot be reliably captured through simple filtering alone [12].

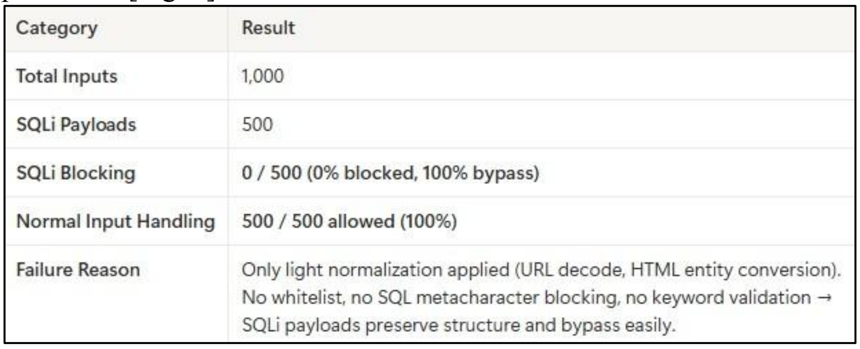
4. EXPERIMENT & ANALYSIS

In this experiment, a total of 1,000 inputs were used, consisting of 500 SQL Injection payloads and 500 normal inputs, such as valid login attempts, incorrect passwords, and nonexistent account queries. This configuration simulates a realistic login environment in which legitimate and malicious requests coexist, enabling a balanced evaluation of attack-blocking performance and false-positive occurrences.

4-1 Filtering Technique Based on Input Normalization

(1) Basic Normalization

In the first experiment, only basic normalization techniques—such as simple URL decoding, HTML entity conversion, and whitespace/character cleanup—were applied. As a result, all 500 SQLi payloads successfully bypassed the defense, indicating that the basic normalization alone failed to provide effective protection [Fig. 6].

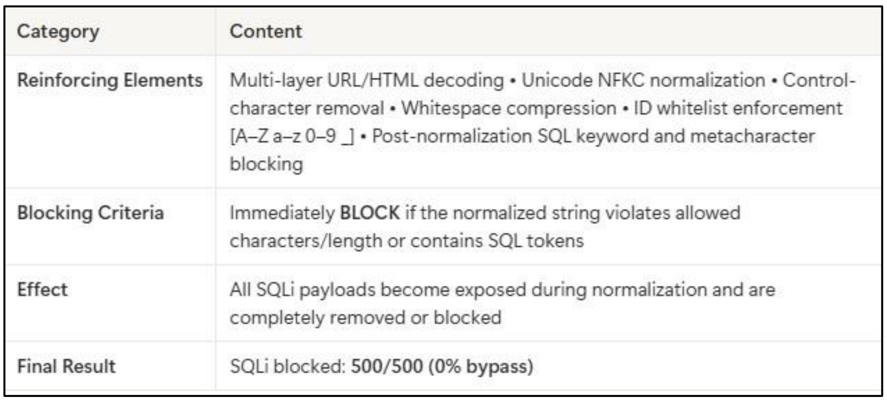


Category	Result
Total Inputs	1,000
SQLi Payloads	500
SQLi Blocking	0 / 500 (0% blocked, 100% bypass)
Normal Input Handling	500 / 500 allowed (100%)
Failure Reason	Only light normalization applied (URL decode, HTML entity conversion). No whitelist, no SQL metacharacter blocking, no keyword validation → SQLi payloads preserve structure and bypass easily.

[Fig. 6] Basic Normalization

(2) Enhanced Normalization

In the second experiment, additional defensive steps were introduced in the sequence of normalization → reduction → whitelist filtering → SQL token-level blocking. As a result, all SQLi payloads were successfully prevented, achieving a 0% bypass rate [Fig. 7].



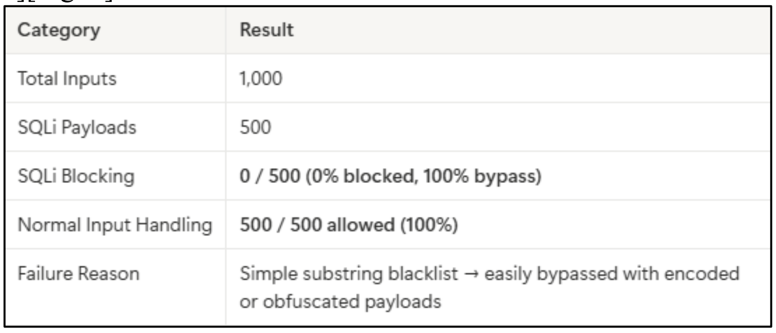
Category	Content
Reinforcing Elements	Multi-layer URL/HTML decoding • Unicode NFKC normalization • Control-character removal • Whitespace compression • ID whitelist enforcement [A–Z a–z 0–9 _] • Post-normalization SQL keyword and metacharacter blocking
Blocking Criteria	Immediately **BLOCK** if the normalized string violates allowed characters/length or contains SQL tokens
Effect	All SQLi payloads become exposed during normalization and are completely removed or blocked
Final Result	SQLi blocked: **500/500 (0% bypass)**

[Fig. 7] Enhanced Normalization

4-2 Regular-Expression-Based Blacklist and Whitelist Filtering

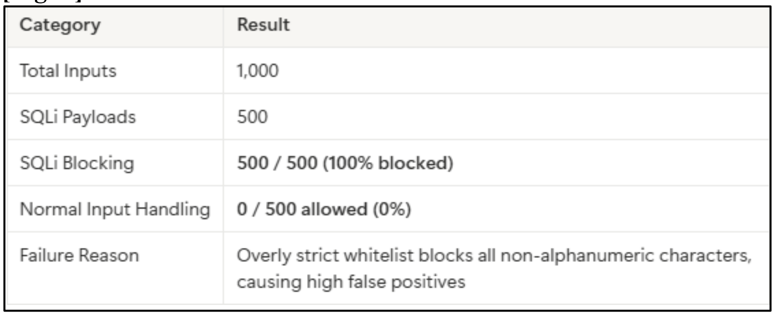
(1) Basic Blacklist-Whitelist

The first experiment applied a simple Blacklist and Whitelist approach, and both methods were completely bypassed (100% bypass rate), resulting in a failure to defend against SQL Injection attacks [Fig. 8][Fig. 9].



Category	Result
Total Inputs	1,000
SQLi Payloads	500
SQLi Blocking	**0 / 500 (0% blocked, 100% bypass)**
Normal Input Handling	**500 / 500 allowed (100%)**
Failure Reason	Simple substring blacklist → easily bypassed with encoded or obfuscated payloads

[Fig. 8] Basic Blacklist

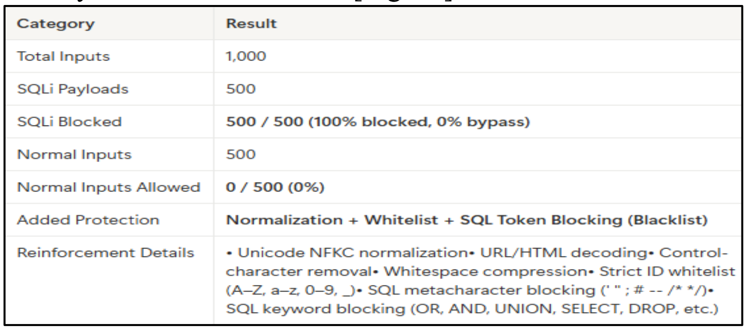


Category	Result
Total Inputs	1,000
SQLi Payloads	500
SQLi Blocking	**500 / 500 (100% blocked)**
Normal Input Handling	**0 / 500 allowed (0%)**
Failure Reason	Overly strict whitelist blocks all non-alphanumeric characters, causing high false positives

[Fig. 9] Basic Whitelist

(2) Hybrid Filtering

Hybrid filtering is an enhanced technique that combines normalization, whitelist enforcement, SQL keyword blocking, and meta-character restrictions. In the experiment, this method successfully blocked 100% of SQL Injection payloads, with no bypass attempts observed. However, the input constraints were excessively strict, resulting in the unintended rejection of some legitimate user inputs. Thus, although the security effectiveness is very high, the method presents usability limitations that make it difficult to deploy directly in real-world services [Fig. 10].



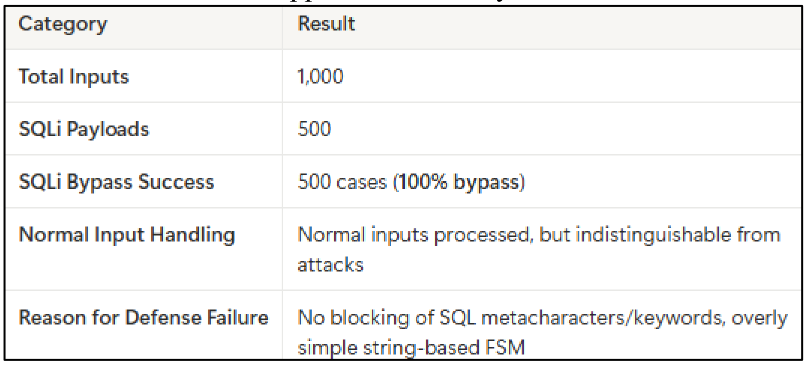
Category	Result
Total Inputs	1,000
SQLi Payloads	500
SQLi Blocked	**500 / 500 (100% blocked, 0% bypass)**
Normal Inputs	500
Normal Inputs Allowed	**0 / 500 (0%)**
Added Protection	**Normalization + Whitelist + SQL Token Blocking (Blacklist)**
Reinforcement Details	• Unicode NFKC normalization• URL/HTML decoding• Control-character removal• Whitespace compression• Strict ID whitelist (A–Z, a–z, 0–9, _)• SQL metacharacter blocking (' " ; # -- /* */)• SQL keyword blocking (OR, AND, UNION, SELECT, DROP, etc.)

[Fig. 10] Hybrid Filtering

4-3 FSM (Finite State Machine) based Context Validation Method

(1) Basic FSM

The first experiment employed a basic FSM that relied solely on simple string inspection, without state-based blocking for SQL meta-characters (e.g., OR, --, '). As a result, most payloads were able to pass as long as their structural form appeared minimally valid.



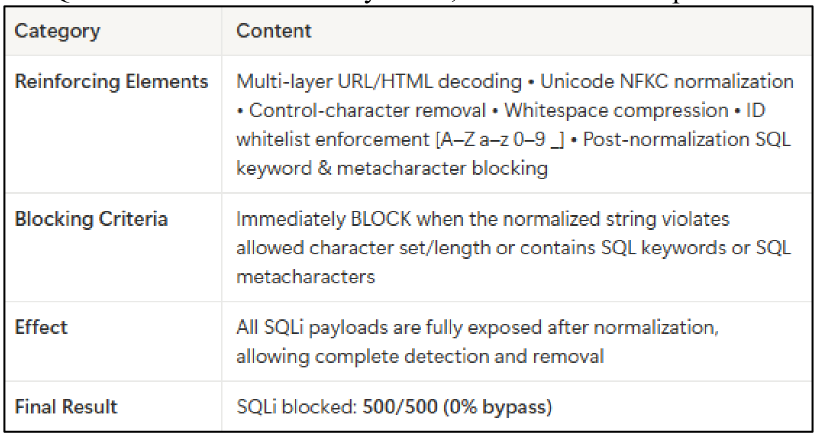
Category	Result
Total Inputs	1,000
SQLi Payloads	500
SQLi Bypass Success	500 cases (**100% bypass**)
Normal Input Handling	Normal inputs processed, but indistinguishable from attacks
Reason for Defense Failure	No blocking of SQL metacharacters/keywords, overly simple string-based FSM

[Fig. 11] Basic FSM Result

As a result, because additional protection mechanisms such as input normalization and character-level whitelisting were not applied, all 500 SQLi payloads successfully bypassed the defense, leading to a complete failure in protection [Fig. 11].

(2) Enhanced FSM

In the second experiment, the same CSV input dataset was used, but the FSM design was significantly reinforced. The applied techniques included input normalization, whitelist filtering, state-based blocking of SQL meta-characters and keywords, and the use of Prepared Statements in the final stage.



Category	Content
Reinforcing Elements	Multi-layer URL/HTML decoding • Unicode NFKC normalization • Control-character removal • Whitespace compression • ID whitelist enforcement [A–Z a–z 0–9 _] • Post-normalization SQL keyword & metacharacter blocking
Blocking Criteria	Immediately BLOCK when the normalized string violates allowed character set/length or contains SQL keywords or SQL metacharacters
Effect	All SQLi payloads are fully exposed after normalization, allowing complete detection and removal
Final Result	SQLi blocked: **500/500 (0% bypass)**

[Fig. 12] Enhanced FSM Result

The enhanced FSM technique successfully blocked all 500 SQL Injection payloads, while normal inputs and general error cases were processed without any issues.

4-4 Comparative Analysis of the Four Methods

This section compares and analyzes the security effectiveness and limitations of input normalization, regex-based Blacklist/Whitelist filtering, and FSM-based context validation, based on experimental results obtained using an identical test vector. The dataset consisted of 1,000 total inputs (500 normal and 500 SQL Injection payloads), and each technique was evaluated in terms of SQLi blocking rate, normal request handling, and susceptibility to bypass attacks.

(1) Comparative Evaluation Across Methods

The comparison of input normalization, Blacklist/Whitelist filtering, and FSM-based context validation under the same experimental conditions revealed distinct characteristics and security levels for each technique.

First, basic normalization applied only simple preprocessing steps such as URL decoding and HTML entity handling. As a result, the structural meaning of the SQLi payloads remained intact, allowing all 500 attack payloads to bypass detection. This confirms that normalization alone is not a standalone defense mechanism but rather an auxiliary step intended to enhance the accuracy of subsequent filtering.

Second, the basic Blacklist approach—relying on keyword-based string matching—proved vulnerable to common evasion methods such as case manipulation, whitespace insertion, and encoding changes. In the experiment, all attack payloads successfully bypassed this method, while all normal inputs were accepted.

This aligns with prior studies demonstrating the inherent limitations of purely pattern-based Blacklist filtering.

Third, basic Whitelist filtering blocked all 500 SQLi payloads due to its strict acceptance of only predefined characters. However, the approach also rejected 100% of normal inputs, highlighting significant usability concerns and its impracticality for direct deployment in real-world systems.

Fourth, the basic FSM method analyzed SQL context states—such as string, comment, and code states—rather than relying solely on string comparison. While this enabled greater resilience to evasion attempts, the limited SQL keyword set, absence of normalization, and simplified state transitions allowed several payloads to bypass detection.

(2) Security Operation Strategy Recommendations

Based on the enhanced experiments—Enhanced Normalization, Enhanced Hybrid, and Enhanced FSM—this study confirms that no single technique is sufficient to completely block SQL Injection attacks. Instead, a multilayered defense architecture composed of Normalization → Character-Based Whitelist → SQL Token/Keyword Validation → FSM or Prepared Statement provides the most effective protection. From these results, the following operational strategies are recommended for practical deployment.

a. Apply normalization to all incoming inputs

The enhanced normalization experiment applied multi-step preprocessing—multiple URL/HTML decodings, Unicode NFKC normalization, control character removal, and whitespace compression. All 500 SQLi payloads were fully exposed during normalization and subsequently blocked (0% bypass). This demonstrates that normalization serves as the primary defensive layer that determines the accuracy of all subsequent filtering. Therefore, all inputs must be normalized into a standardized form before entering the service logic.

b. Apply Whitelist filtering selectively to fixed-pattern fields

In the enhanced Hybrid experiment, applying a Whitelist (letters, digits, and underscores) solely to the user ID field resulted in a 100% blocking rate for all SQLi payloads. However, because broad Whitelist enforcement can inadvertently block legitimate inputs, it should be selectively applied to fields with predictable formats, such as user IDs, page numbers, or integer parameters. Limiting Whitelist filtering to high-risk, fixed-structure fields ensures strong protection while preserving usability.

c. Use Blacklist filtering as a supplementary detection layer

The enhanced Hybrid experiment demonstrated that adding SQL keyword (e.g., OR, UNION, SELECT) and meta-character (e.g., ‘, ”, --) blocking provided an additional layer of defense, effectively catching payloads that might otherwise bypass normalization or Whitelist filtering. Although Blacklist filtering is weak as a standalone defense, it is valuable as a supplementary signature-based layer that enhances the overall detection capability.

d. Apply FSM or Prepared Statements as the final defense layer

The enhanced FSM experiment accurately tracked SQL context states (code, string, comment) and achieved a 100% blocking rate for all 500 SQLi payloads while correctly handling all normal inputs. FSM is the most powerful final defensive layer, capable of detecting attacks that evade simple string-based filters. Prepared Statements provide a comparable level of security by fully separating query logic from input values. Therefore, a multilayered structure of Normalization → Whitelist/Blacklist → FSM or Prepared Statement constitutes the most reliable SQL Injection defense pipeline.

5. CONCLUSION

In This study compared and analyzed four major SQL Injection defense techniques—input normalization, regex-based Blacklist/Whitelist filtering, and FSM-based context validation—under a unified experimental environment. The test vector consisted of 500 normal inputs and 500 SQLi payloads, and each technique was evaluated in both basic and enhanced versions. In the basic experiments, normalization and Blacklist filtering showed clear limitations as standalone defense mechanisms, as they were unable to block most SQLi payloads. The Whitelist method successfully blocked all SQLi attempts but also rejected all normal inputs, revealing a critical usability issue. The basic FSM achieved relatively high detection performance but failed to completely mitigate all attack payloads. These findings indicate that no single technique is sufficient to reliably defend against diverse SQLi evasion attacks. In contrast, the enhanced experiments demonstrated that a multilayered architecture—combining normalization, Whitelist filtering, SQL keyword and meta-character blocking, and FSM—achieved a 100% blocking rate across all 500 SQLi payloads, with no bypasses observed. Enhanced FSM and enhanced normalization, in particular, played a decisive role by accurately detecting payloads involving various forms of obfuscation, encoding transformations, and whitespace manipulation. These results empirically confirm that a sequential multilayered strategy—consisting of normalization, character-based restriction, keyword inspection, and context-aware validation—provides the most effective defense against SQL Injection. Therefore, the study concludes that a stepwise defense structure of Normalization → Blacklist/Whitelist → SQL token validation → FSM/Prepared Statement is essential for minimizing bypass attacks and ensuring robust protection. Future research may extend this work through machine learning–based payload classification, traffic sequence pattern analysis, and real-world service-level experiments to further validate and enhance the multilayered defense framework.

Acknowledgments

This research was supported by the MSIT(Ministry of Science ICT), Korea, under the National Program for Excellence in SW, supervised by the IITP(Institute of Information & Communications Technology Planning & Evaluation) in 2025 (No. 2024-0-00023).

REFERENCES

1. W. G. J. Halfond, J. Viegas, and A. Orso, “A Classification of SQL -Injection Attacks and Countermeasures,” IEEE International Symposium on Secure Software Engineering, pp. 13–15, 2006.
2. LIU, Anyi, et al. SQLProb: a proxy-based architecture towards preventing SQL injection attacks. In: Proceedings of the 2009 ACM symposium on Applied Computing. pp. 2054-20612009, 2009.
3. G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, “Using Parse Tree Validation to Prevent SQL Injection Attacks,” IEEE International Workshop on Software Engineering for Secure Systems, pp. 106–113, 2005.
4. BALZAROTTI, Davide, et al. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In: 2008 IEEE Symposium on Security and Privacy (sp 2008). IEEE, pp. 387-401, 2008.
5. F. Valeur, D. Mutz, and G. Vigna, “A Learning-Based Approach to the Detection of SQL Attacks,” Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), vol. 3548, pp. 123–140, 2005.
6. SHIN, Yonghee; WILLIAMS, Laurie; XIE, Tao. SQLUnitgen: Test case generation for SQL injection detection. North Carolina State University, Raleigh Technical report, NCSU CSC TR, 2006.
7. SCHOLTE, Theodoor, et al. An empirical analysis of input validation mechanisms in web applications and languages. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, pp. 1419-1426, 2012.
8. AMMAGUNTA, Sathish, et al. Defending against SQL injection: Practical application with open-source tools for improved cyber security. In: AIP Conference Proceedings. AIP Publishing LLC, pp. 020036, 2025.
9. JOSHI, Anamika; GEETHA, V. SQL Injection detection using machine learning. In: 2014 international conference on control, instrumentation, communication and computational technologies (ICCICCT). IEEE, pp. 1111-1115, 2014.
10. ELIA, Ivano Alessandro; FONSECA, Jose; VIEIRA, Marco. Comparing SQL injection detection tools using attack injection: An experimental study. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering. IEEE, pp. 289-298, 2010.
11. Z. Su and G. Wassermann, “The Essence of Command Injection Attacks in Web Applications,” ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), pp. 372–382, 2006.
12. NGUYEN-TUONG, Anh, et al. Automatically hardening web applications using precise tainting. In: IFIP International Information Security Conference. Boston, MA: Springer US, pp. 295-307, 2005.