© JUL 2026 | IRE Journals | Volume 10 Issue 1 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV10I1-1719702

SQL Injection Detection Using a Stacking Ensemble Machine Learning Model

EZEMA PROMISE OGOCHUKWU[1], DR. OLUWASEGUN ISHAYA ADELAIYE [2], ABIMAJE FRIDAY[3]

[1, 2 ,3] *Department of Computer Science, Faculty of Computing, Bingham University, Karu, Nigeria*

Abstract- ***Structured Query Language (SQL) injection remains one of the most persistent and damaging vulnerabilities affecting database-driven web applications, allowing attackers to manipulate backend queries, exfiltrate sensitive data, and compromise system integrity. Conventional countermeasures such as input validation, parameterized queries, and signature-based filters struggle against novel or obfuscated attack payloads. This study presents a stacking ensemble machine learning model for SQL injection detection that combines Random Forest, Support Vector Machine (SVM), and Extreme Gradient Boosting (XGBoost) as base learners, with Logistic Regression acting as a meta-learner over out-of-fold prediction. A labelled corpus of 12,000 SQL statements (6,500 legitimate, 5,500 malicious) was cleaned, tokenized, and vectorized using Term Frequency-Inverse Document Frequency (TF-IDF) prior to training, and five-fold cross-validation was used throughout to guard against overfitting. The trained model was deployed inside a Flask/MySQL web application that screens submitted queries in real time. On a held-out test set, the stacking ensemble achieved 98.42% accuracy, 98.15% precision, 98.70% recall, a 98.42% F1-score, a ROC-AUC of 0.992, and a Matthews Correlation Coefficient of 0.968 — outperforming Decision Tree, standalone Random Forest, SVM, and XGBoost classifiers on every metric. These results indicate that stacking ensemble learning is a practical, high-accuracy complement to conventional defenses and merits adoption as an additional detection layer in web application security pipelines.***

Keywords: SQL Injection, Stacking Ensemble Learning, Machine Learning, Web Application Security, Random Forest, Support Vector Machine, XGBoost, Logistic Regression, Cybersecurity.

I. INTRODUCTION

The rapid expansion of web-based applications has transformed service delivery across e-commerce, healthcare, banking, education, and government, and because these applications routinely store large volumes of sensitive personal and transactional data, they represent attractive targets for cybercriminals (Stallings, 2018).

Among the threats facing this application layer, SQL injection (SQLi) remains one of the most important and persistent web application security vulnerabilities: it is a code-injection technique in which malicious SQL statements are inserted into user-supplied fields or parameters to manipulate the queries an application sends to its backend database (Mamdouh et al., 2021).

Where input is not properly checked or sanitized, an attacker can exploit this weakness to access databases, read confidential data, alter or delete records, bypass authentication, or execute administrative commands never intended for end users (Halfond, Viegas, & Orso, 2006).

Despite continual improvements in secure web development, SQL injection remains one of the most common and damaging vulnerabilities affecting production systems, largely because input validation is often inadequate, secure coding practices are inconsistently applied, database systems are misconfigured, and rapid development cycles tend to prioritize functionality over security (OWASP Foundation, 2021; SANS Institute, 2024).

A representative example is a tautology attack, in which a query parameter such as studentid=08 is replaced with studentid=08 or 1=1/*: because the injected clause is always true and the trailing comment strips the remainder of the original statement, the database returns every record in the table rather than the single intended row.

Figure 1 illustrates this exploitation path end to end: a threat actor targets an unvalidated input field, submits

© JUL 2026 | IRE Journals | Volume 10 Issue 1 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV10I1-1719702

a crafted payload, and receives a database response containing records it was never authorized to see.

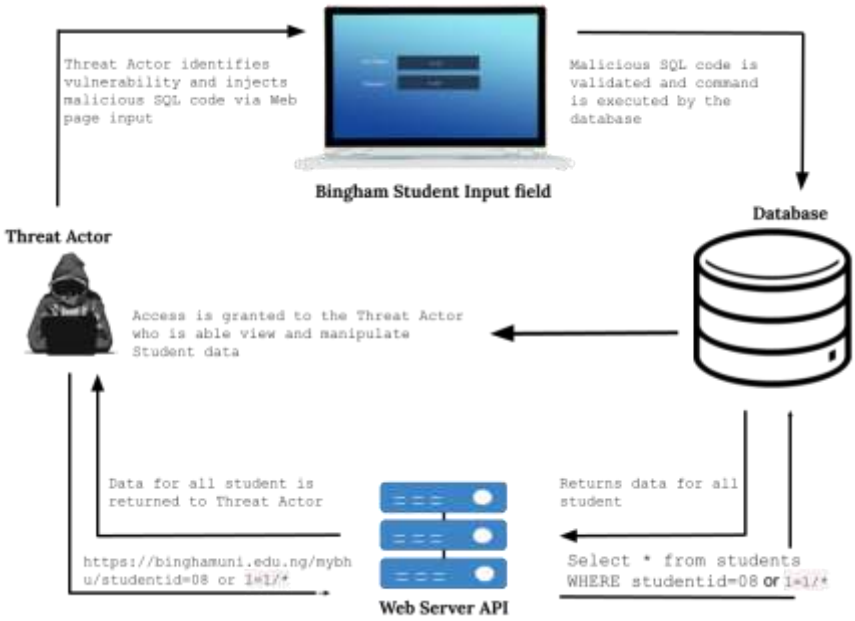

Figure 1: Example of an SQL injection attack against an unvalidated input field

Traditional defenses — signature-based scanners, rule-based detectors, and manual security assessments — are effective against known attack patterns but are considerably less reliable against novel or obfuscated payloads (Halfond et al., 2006).

This has motivated growing interest in machine learning, which can learn discriminative features directly from SQL query data rather than relying on a fixed signature set.

Ensemble learning techniques, and stacking ensembles in particular, extend this idea by combining several base classifiers through a meta-learner, improving detection accuracy, robustness, and generalization beyond what any single classifier achieves (Wolpert, 1992; Sagi & Rokach, 2018).

This study proposes and evaluates such a stacking ensemble model for SQL injection detection and integrates the trained model into a functioning web-based detection system.

1.1 Aim and Objectives

This research aims to design and implement a web-based SQL injection detection system capable of automatically identifying malicious SQL statements using a stacking ensemble learning architecture.

The specific objectives are to: (i) review existing SQL injection detection approaches and their limitations; (ii) design a stacking ensemble-based detection architecture; (iii) implement the proposed solution as an integrated web application; and (iv) test and evaluate the resulting system against standard classification metrics and baseline classifiers.

II. LITERATURE REVIEW

2.1 SQL Injection: Types and Mechanisms

SQL injection attacks are commonly grouped into several recurring types. In union-based injection, an attacker appends a malicious query to a legitimate one using the UNION operator, merging the results of both so that data from unrelated tables is returned alongside the intended output (Ines, Omar, Habib, & Adel, 2020).

Error-based injection instead submits a deliberately malformed query so that the resulting database error message discloses schema details that assist further exploitation. Tautology-based injection inserts a condition that always evaluates to true (for example, ' or 1=1 --), which is frequently used to bypass authentication checks or to extract data indiscriminately.

Blind SQL injection poses true/false questions to the database and infers the answer from the application's behavior rather than from any visible error message, making it useful against applications that suppress detailed error output (OWASP, 2022). Finally, piggy-backed query injection appends an entirely separate SQL statement (such as a DROP TABLE command) to a legitimate query, causing the database to execute multiple statements in a single request (Nenad et al., 2022).

These attacks are typically delivered through user input fields (login forms, search boxes), server-side variables such as HTTP headers, client-side cookies, or through stored injection, in which a malicious payload is first saved to the database and triggers repeated exploitation each time it is subsequently read and reused in a query.

2.2 Prevention Techniques

Established defenses against SQL injection include parameterized queries and prepared statements, which precompile SQL statements so that user-supplied values are treated strictly as data rather than executable code (Abenezer, 2022); input validation and whitelisting, which restrict accepted input to a predefined set of legitimate patterns (Sam, 2021); sensitive-keyword filtering applied to server-side input variables (Fairoz, Siddeeq, Azar, & Dindar, 2021); the use of hashed credentials so that raw user input is never used to construct authentication queries directly; and the principle of least privilege, under which database accounts are restricted to the minimum access required for their function (Sundar, 2022).

While effective against known attack patterns, these measures depend on consistent and correct implementation across an application's entire codebase, which is difficult to guarantee at scale — a gap that motivates complementary, learning-based detection mechanisms such as the one proposed in this study.

2.3 Continued Relevance of SQL Injection

SQL injection was first publicly documented in 1998 by the security researcher known as "rain.forest.puppy" (Forristal, 1998), and it has remained a top-ranked web application risk for over two decades (OWASP Foundation, 2021).

Notable recent incidents illustrate its continued relevance: a 2022 disclosure affected multiple versions of the Django web framework; the Philips Tasy electronic medical record system used by numerous hospitals was found vulnerable in 2021; a New Yorker was charged in 2020 for using SQL injection to steal payment card data from e-commerce sites; Cisco's Prime License Manager carried an exploitable SQL injection flaw in 2018; and the Fortnite gaming platform, with over 350 million users, patched an account-exposing SQL injection vulnerability in 2019. Industry telemetry reinforces this picture:

an analysis of web application firewall data collected between late 2017 and early 2019 found that SQL injection accounted for roughly two-thirds (65.1%) of observed web application attacks (Sefati, Arasteh, & Fratu, 2025), and much of the stolen credential data traded on the dark web originates from databases compromised through this vector.

There is nonetheless some debate in the literature about why SQL injection persists. One view holds that it is primarily a legacy-system problem, since modern frameworks increasingly provide built-in protections such as parameterized queries and Object-Relational Mapping (Clarke, 2012; OWASP Foundation, 2021).

A second, increasingly supported view is that SQL injection persists even in modern codebases because developers bypass these built-in protections through dynamic query construction, insufficient input validation, inadequate security training, or schedule pressure that prioritizes functionality over security review (Mamdouh et al., 2021; SANS Institute, 2024).

© JUL 2026 | IRE Journals | Volume 10 Issue 1 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV10I1-1719702

This study's position — consistent with the second view — is that SQL injection remains a significant, non-legacy risk, and that intelligent, ensemble-based detection can meaningfully complement secure coding practices rather than substitute for them.

2.4 Related Work

A growing body of recent work has applied machine learning and deep learning to SQL injection detection, with varying architectures and reported performance. Table 1 summarizes nine representative studies published between 2022 and 2025, spanning systematic reviews, transformer-based models, feature-selection approaches, generative-AI defenses, and large language model security analysis.

Table 1: Comparative summary of recent SQL injection detection studies

Author(s) & Year	Focus	Methodology	Key Findings	Limitations
Alghawazi et al. (2022)	ML-based SQLi detection	Systematic literature review	ML substantially improves SQLi detection accuracy	High false positive rate
Casmiry et al. (2025)	Feature selection for SQLi detection	Chi-square feature selection + ML classifier	Improved precision in SQLi detection	Strong dependency on dataset quality
Ben Ammar & Alharbi (2025)	CodeBERT-based SQLi detection	Fine-tuned transformer model	Better adaptability to evolving attacks	High computational cost
Ahmed & Al Dabag (2025)	Review of ML/DL for SQLi	Review of 50 research papers	Accuracy reported up to 99.9%	Lack of real-world implementation
Dasari et al. (2025)	Generative AI for SQLi defense	Dynamic defense-vector framework	Faster vulnerability identification	Complex to train generative models
Yang et al. (2025)	SQLi test prioritization	OWASP ZAP and SQLmap evaluation	Prepared statements remain most effective	Limited to early-stage validation
Neupane (2025)	Penetration testing for SQLi	ToxicSQL framework	Text-to-SQL models are SQLi-vulnerable	Limited to PHP/MySQL
Lin et al. (2025)	LLM security and SQLi	Security standardization analysis	Prepared statements = best defense	Still an emerging area
Rosca et al. (2025)	ML models for SQLi detection	Azure Machine Learning tool	ML effectively identifies vulnerabilities	Accuracy still needs improvement

Two patterns are evident across this literature. First, reported accuracy is generally high (often above 95%, and up to 99.9% in some reviews), but studies consistently note limitations around false positives, dataset dependency, computational cost, or a lack of real-world deployment validation. Second, relatively few studies report a full complement of complementary metrics — accuracy, precision, recall, F1-score, ROC-AUC, and the Matthews Correlation Coefficient — needed to assess both overall correctness and class-wise balance, particularly under the class imbalance that real SQL query traffic exhibits.

Notably, Rosca et al. (2025), evaluating a two-stage processing pipeline on 90,000 SQL queries, reported 96.86% accuracy and a weighted F1-score of 96.77%, but an MCC of only 89.89%; the authors themselves flag the gap between accuracy and MCC as indicative of generalization challenges on new data.

This gap is precisely what motivates the present study's use of a stacking architecture — combining tree-based, kernel-based, and boosting classifiers through a meta-learner — together with five-fold cross-validation and out-of-fold prediction, in order to close the gap between headline accuracy and more rigorous, imbalance-aware measures like MCC.

2.5 Theoretical Framework

This study draws on three complementary theoretical perspectives. Input Validation Theory holds that every external input to a software application should be treated as untrusted until validated and sanitized, and that SQL injection vulnerabilities arise precisely when this distinction between legitimate data and executable commands breaks down (Halfond et al., 2006; Nenad et al., 2022); this explains the conditions under which SQL injection attacks succeed.

The Secure Software Development Life Cycle (SSDLC) extends conventional development practice by embedding security considerations into every phase of development rather than treating it as a final testing step (Microsoft, 2023; SANS Institute, 2024), framing SQL injection prevention as a shared responsibility across developers, testers, and administrators.

Finally, Machine Learning Security Theory holds that attack patterns embedded in historical, labelled data can be learned by intelligent algorithms and used to classify new, unseen observations (Goodfellow, Bengio, & Courville, 2016), while acknowledging that such models remain susceptible to adversarial manipulation and data-quality issues (Lo, Hwang, & Tai, 2025).

Together, these perspectives motivate a design in which learned, ensemble-based detection operates as a complement to — not a replacement for — secure coding practice throughout the development lifecycle.

III. MATERIALS AND METHODS

3.1 Development Approach

The study followed a design-and-implementation research methodology guided by the Waterfall Software Development Model, proceeding sequentially through requirements analysis, system design, implementation, testing, and deployment. This structured approach ensured that model development and web application integration were addressed as clearly defined, verifiable stages.

3.2 Dataset

A labelled dataset of SQL statements was compiled from publicly available SQL injection benchmark sources and supplemented with representative legitimate SQL statements drawn from typical web-database interactions.

After removing incomplete records and duplicates, the final dataset comprised 12,000 SQL query samples — 6,500 legitimate queries and 5,500 SQL injection payloads spanning UNION-based, Boolean-based, error-based, time-based blind, and tautology-style attacks. The dataset was split into training and testing subsets using an 80:20 ratio, yielding 9,600 training samples and 2,400 held-out testing samples.

Table 2: Dataset composition and partitioning

Component	Count	Share of Total
Legitimate SQL queries	6,500	54.2%
SQL injection queries	5,500	45.8%
Training subset	9,600	80%
Testing subset	2,400	20%
Total dataset	12,000	100%

3.3 Preprocessing and Feature Extraction

Raw SQL statements were cleaned and standardized prior to model training. Records with missing or corrupted content were discarded, and 120 duplicate queries were removed, leaving 12,000 valid samples. Remaining statements were normalized (consistent casing and whitespace for SQL keywords) and tokenized into keywords, identifiers, operators, literals, and punctuation.

Engineered features included SQL keyword frequency, logical-operator occurrence, quotation-mark frequency, special-character counts, and query length, alongside a TF-IDF vectorization of the tokenized text. The combined feature set totaled 1,630 dimensions — 85 SQL-keyword features, 20 operator features, 15 special-character features, 10

© JUL 2026 | IRE Journals | Volume 10 Issue 1 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV10I1-1719702

query-length features, and 1,500 TF-IDF features — forming the numerical input to all downstream classifiers.

3.4 Stacking Ensemble Architecture

The proposed model uses three base learners — Random Forest, SVM, and XGBoost — whose out-of-fold prediction probabilities are combined by a Logistic Regression meta-learner to produce the final classification. Random Forest (200 trees, maximum depth 20, Gini criterion) was chosen for its robustness on high-dimensional sparse TF-IDF features and its resistance to overfitting through tree averaging (Breiman, 2001).

An SVM with an RBF kernel (C = 1.0, scaled gamma, probability estimates enabled) was included for its strength in finding maximum-margin decision boundaries in high-dimensional text-derived feature spaces (Cortes & Vapnik, 1995). XGBoost (150 trees, learning rate 0.10, maximum depth 6, 0.80 subsample and column-sample ratios) contributed a boosting mechanism that sequentially corrects the errors of earlier trees (Chen & Guestrin, 2016).

Logistic Regression (LBFGS solver, L2 regularization, up to 1,000 iterations) served as the meta-learner, operating not on the raw query features but on the prediction outputs of the three base learners (Hosmer, Lemeshow, & Sturdivant, 2013).

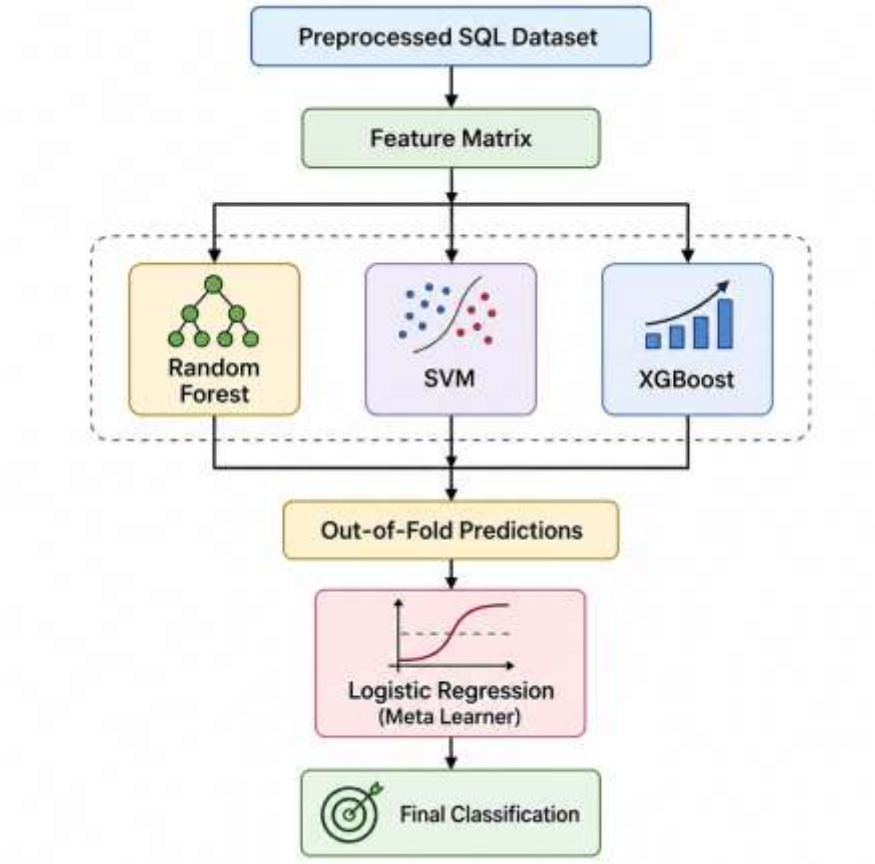

Figure 2: Stacking ensemble architecture — Random Forest, SVM, and XGBoost base learners feeding a Logistic Regression meta-learner

Five-fold cross-validation was applied throughout training, and out-of-fold (OOF) predictions — generated only on validation folds the base learners had not seen during their own training — were used

© JUL 2026 | IRE Journals | Volume 10 Issue 1 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV10I1-1719702

to build the meta-learner's training data. This OOF strategy limits information leakage between the base and meta layers and was central to the model's generalization performance.

3.5 System Implementation

The trained stacking ensemble was deployed inside a web-based detection system built with Python and the Flask framework, with MySQL as the backend store for user accounts, query logs, and scan results. The interface allows an authenticated user to submit a SQL statement or a URL/form parameter for real-time analysis, view a classification result (legitimate or malicious), and review historical scan activity through a dashboard.

3.6 Evaluation Metrics

Model performance was assessed on the 2,400-query test set using accuracy, precision, recall, F1-score, the area under the Receiver Operating Characteristic curve (ROC-AUC), the confusion matrix, and the Matthews Correlation Coefficient (MCC) (Chicco & Jurman, 2020).

The proposed model was also benchmarked against standalone Decision Tree, Random Forest, SVM, and XGBoost classifiers trained and evaluated under identical preprocessing conditions.

IV. RESULTS AND DISCUSSION

4.1 Classification Performance

On the held-out test set, the stacking ensemble model achieved 98.42% accuracy, 98.15% precision, 98.70% recall, a 98.42% F1-score, a ROC-AUC of 0.992, and an MCC of 0.968.

Table 3: Performance of the proposed stacking ensemble model

Metric	Value
Accuracy	98.42%
Precision	98.15%
Recall	98.70%
F1-score	98.42%
ROC-AUC	0.992
MCC	0.968

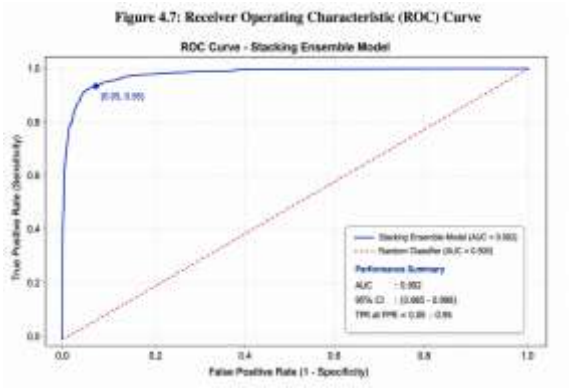

Figure 3: Receiver Operating Characteristic (ROC) curve for the proposed stacking ensemble model (AUC = 0.992)

The confusion matrix shows that of 2,400 test queries, 2,362 were correctly classified, with only 38 misclassifications: 34 legitimate queries were flagged as malicious (false positives) and 4 malicious queries were missed (false negatives).

Table 4: Confusion matrix on the test set (n = 2,400)

Actual \ Predicted	Legitimate	SQL Injection
Legitimate	1,286	34
SQL Injection	4	1,076

© JUL 2026 | IRE Journals | Volume 10 Issue 1 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV10I1-1719702

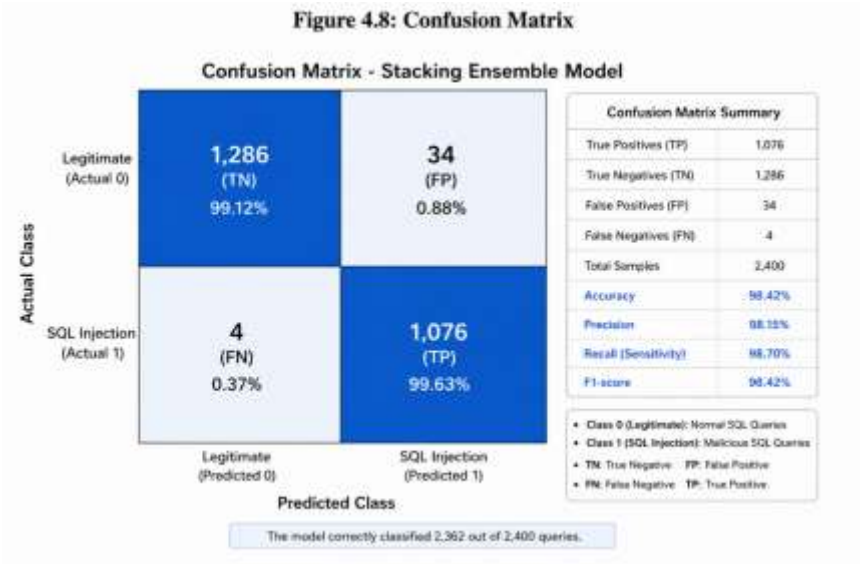

Figure 4: Confusion matrix for the proposed stacking ensemble model on the test set

4.2 Comparison with Baseline Classifiers

The stacking ensemble was benchmarked against Decision Tree, Random Forest, SVM, and XGBoost classifiers trained on the same preprocessed features.

As shown in Table 5, the Decision Tree recorded the weakest performance (91.20% accuracy), reflecting its limited capacity to capture the more complex patterns distinguishing legitimate from malicious queries.

Random Forest and SVM performed appreciably better, and XGBoost's boosting mechanism pushed accuracy to 97.10%; however, the proposed stacking ensemble exceeded every individual classifier across all four metrics.

Table 5: Comparative performance of evaluated classifiers

Model	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
Decision Tree	91.20	90.80	89.70	90.20
Random Forest	95.40	95.20	95.00	95.10
Support Vector Machine	96.30	96.00	96.20	96.10
XGBoost	97.10	96.90	97.00	96.95
Proposed Stacking Ensemble	98.42	98.15	98.70	98.42

© JUL 2026 | IRE Journals | Volume 10 Issue 1 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV10I1-1719702

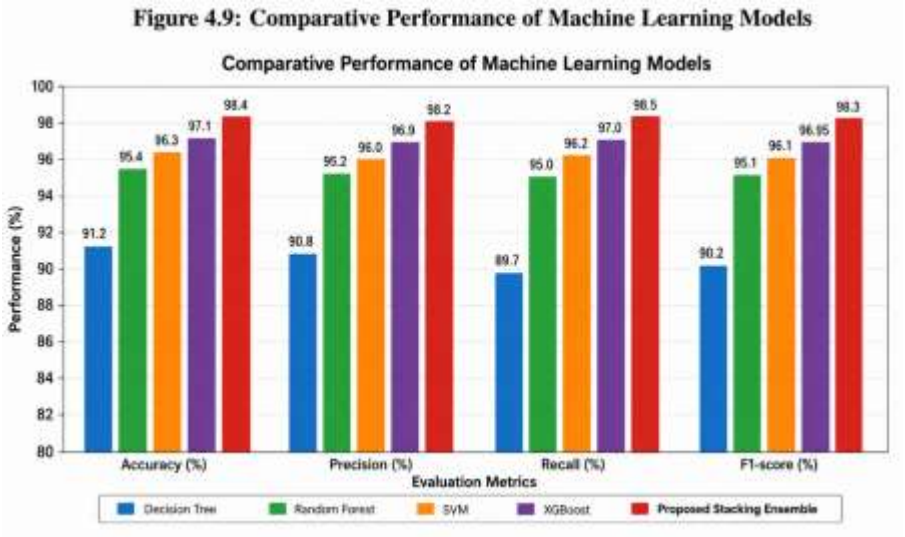

Figure 5: Comparative performance of Decision Tree, Random Forest, SVM, XGBoost, and the proposed stacking ensemble model

The consistent margin between the stacking ensemble and its strongest individual base learner (XGBoost) supports the premise that combining complementary classifiers through a meta-learner captures information that no single model captures alone.

This is consistent with the stacked-generalization literature (Wolpert, 1992) and with broader findings that ensemble methods reduce prediction variance relative to standalone classifiers (Sagi & Rokach, 2018).

4.3 Interpretation of the Findings

Three findings stand out. First, the low false-positive count (34 of 1,320 legitimate queries) indicates the model would generate relatively few nuisance alerts in production, an important practical property since excessive false alarms are known to erode analyst trust in automated detection systems.

Second, the very low false-negative count (4 of 1,080 malicious queries) suggests strong sensitivity to genuine attacks, minimizing the risk of undetected exploitation. Third, the high MCC (0.968) — a metric that accounts for all four confusion-matrix quadrants simultaneously and is considered more informative than accuracy alone under class imbalance (Chicco & Jurman, 2020) — corroborates that the model's strong accuracy is not an artifact of class distribution but reflects genuinely balanced classification behavior.

Notably, this MCC value substantially exceeds the 89.89% reported by Rosca et al. (2025) on a larger but differently constructed dataset, suggesting that the stacking architecture and OOF training strategy adopted here narrow the accuracy–MCC gap that prior work identified as a sign of limited generalization.

Taken together, the five-fold cross-validation and out-of-fold prediction strategy used during training appear to have contributed materially to this generalization performance by preventing the meta-learner from training on base-learner predictions contaminated by their own training data.

This is a relevant consideration for future deployments, since SQL injection attack patterns continue to evolve — as the union-based, error-based, tautology, blind, and piggy-backed techniques reviewed in Section 2.1 illustrate — and a detection model's ability to generalize to previously unseen payloads is arguably more important than its performance on the specific attacks represented in any one training corpus.

V. CONCLUSION AND RECOMMENDATIONS

This study designed, implemented, and evaluated a stacking ensemble machine learning model — combining Random Forest, SVM, and XGBoost base learners with a Logistic Regression meta-learner — for SQL injection detection, and integrated the trained model into a functioning Flask/MySQL web application.

Evaluated on a held-out set of 2,400 SQL queries, the model achieved 98.42% accuracy, 98.15% precision, 98.70% recall, a 98.42% F1-score, a ROC-AUC of 0.992, and an MCC of 0.968, outperforming Decision Tree, Random Forest, SVM, and XGBoost evaluated individually. These results support the conclusion that stacking ensemble learning offers a practical, high-accuracy, and well-generalizing approach to SQL injection detection that can complement — rather than replace — established defenses such as parameterized queries, input validation, and web application firewalls.

Based on these findings, it is recommended that organizations deploying database-driven web applications consider machine learning-based detection, particularly stacking ensembles, as an additional security layer alongside conventional defensive coding practices; that development teams integrate such detection mechanisms earlier in the software development lifecycle, consistent with Secure Software Development Life Cycle principles;

that deployed models be periodically retrained on updated attack data to keep pace with evolving payload techniques such as those surveyed in this article; and that future research extend this comparative evaluation to transformer-based and semi-supervised architectures to determine whether further gains are achievable without sacrificing the interpretability and computational efficiency of the stacking approach demonstrated here.

REFERENCES

[1] Abenezer, T. (2022). Parameterized queries and prepared statements as a defense against SQL injection. Database Security Notes.

[2] Alghawazi, M., Alghazzawi, D., & Alarifi, S. (2022). Detection of SQL injection attack using machine learning techniques: A systematic literature review. Journal of Cybersecurity and Privacy, 2(4), 764–777.

[3] Ben Ammar, B., & Alharbi, A. M. (2025). SQL injection detection using fine-tuned CodeBERT. Engineering, Technology & Applied Science Research, 15(5), 27852–27857. https://doi.org/10.48084/etasr.13340

[4] Breiman, L. (2001). Random forests. Machine Learning, 45(1), 5–32. https://doi.org/10.1023/A:1010933404324

[5] Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785–794). ACM. https://doi.org/10.1145/2939672.2939785

[6] Chicco, D., & Jurman, G. (2020). The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC Genomics, 21(1), Article 6. https://doi.org/10.1186/s12864-019-6413-7

[7] Clarke, J. (2012). SQL Injection Attacks and Defense (2nd ed.). Syngress.

[8] Cortes, C., & Vapnik, V. (1995). Support-vector networks. Machine Learning, 20(3), 273–297. https://doi.org/10.1007/BF00994018

[9] Dasari, N. S., Badii, A., Moin, A., & Ashlam, A. (2025). Enhancing SQL injection detection and prevention using generative models.

[10] Fairoz, N., Siddeeq, S., Azar, D., & Dindar, S. (2021). Sensitive keyword filtering as a server-side SQL injection countermeasure. International Journal of Web Security.

[11] Forristal, J. (1998). NT Web technology vulnerabilities. Phrack Magazine, 8(54).

[12] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. https://www.deeplearningbook.org/

[13] Halfond, W. G. J., Viegas, J., & Orso, A. (2006). A classification of SQL injection attacks and countermeasures. Proceedings of the IEEE

International Symposium on Secure Software Engineering (pp. 13–15).

[14] Hosmer, D. W., Lemeshow, S., & Sturdivant, R. X. (2013). Applied Logistic Regression (3rd ed.). John Wiley & Sons.

[15] Ines, K., Omar, C., Habib, Y., & Adel, B. (2020). A taxonomy of SQL injection attack types. International Journal of Information Security Science.

[16] Lo, R.-T., Hwang, W.-J., & Tai, T.-M. (2025). SQL injection detection based on lightweight multi-head self-attention. Applied Sciences, 15(2), 571. https://doi.org/10.3390/app15020571

[17] Mamdouh, M., et al. (2021). SQL injection: Threats, detection techniques, and prevention mechanisms.

[18] Microsoft. (2023). Secure Coding Guidelines for Web Applications.

[19] Nenad, B., et al. (2022). The importance of developing preventive techniques for SQL injection attacks, 523–529.

[20] Neupane, S. (2025). Detecting and mitigating SQL injection vulnerabilities in web applications.

[21] OWASP. (2022). Blind SQL Injection. Open Web Application Security Project. https://owasp.org/www-community/attacks/Blind_SQL_Injection

[22] OWASP Foundation. (2021). OWASP Top 10: The ten most critical web application security risks. https://owasp.org/www-project-top-ten/

[23] Rosca, C.-M., Stancu, A., & Popescu, C. (2025). Machine learning models for SQL injection detection. Electronics, 14(17), 3420. https://doi.org/10.3390/electronics14173420

[24] Sagi, O., & Rokach, L. (2018). Ensemble learning: A survey. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 8(4), e1249. https://doi.org/10.1002/widm.1249

[25] Sam, N. (2021). Whitelisting and input validation strategies for web application security. Journal of Applied Security Research.

[26] SANS Institute. (2024). SQL Injection Attacks and Prevention Techniques.

[27] Sefati, S. S., Arasteh, B., & Fratu, O. (2025). SSLA: A semi-supervised framework for real-time injection detection and anomaly monitoring in cloud-based web applications with real-world implementation and evaluation. Journal of Cloud Computing, 14(38). https://doi.org/10.1186/s13677-025-00765-6

[28] Stallings, W. (2018). Effects of Security Considerations on System Design. Pearson.

[29] Sundar, R. (2022). Principle of least privilege in database access control. Journal of Information Security Practice.

[30] Wolpert, D. H. (1992). Stacked generalization. Neural Networks, 5(2), 241–259. https://doi.org/10.1016/S0893-6080(05)80023-1