electronics

Article

Enhancing Structured Query Language Injection Detection with Trustworthy Ensemble Learning and Boosting Models Using Local Explanation Techniques [†]

Thi-Thu-Huong Le [1,*], Yeonjeong Hwang [2], Changwoo Choi [2,3], Rini Wisnu Wardhani [2], Dedy Septono Catur Putranto [4] and Howon Kim [2,*]

1 Blockchain Platform Research Center, Pusan National University, Busan 49241, Republic of Korea
2 School of Computer Science and Engineering, Pusan National University, Busan 49241, Republic of Korea; yeonjeong@islab.re.kr (Y.H.); changwoo7463@gmail.com (C.C.); rini.wisnu@pusan.ac.kr (R.W.W.)
3 SmartM2M, Busan 48058, Republic of Korea
4 IoT Research Center, Pusan National University, Busan 49241, Republic of Korea; dedy.septono@pusan.ac.kr
* Correspondence: lehuong7885@gmail.com (T.-T.-H.L.); howonkim@pusan.ac.kr (H.K.)

† This paper is an extended version of our paper published In Proceedings of the 2024 International Conference on Platform Technology and Service (PlatCon-24), Jeju, Republic of Korea, 26–28 August 2024.

Abstract: This paper presents a comparative analysis of several decision models for detecting Structured Query Language (SQL) injection attacks, which remain one of the most prevalent and serious security threats to web applications. SQL injection enables attackers to exploit databases, gain unauthorized access, and manipulate data. Traditional detection methods often struggle due to the constantly evolving nature of these attacks, the increasing complexity of modern web applications, and the lack of transparency in the decision-making processes of machine learning models. To address these challenges, we evaluated the performance of various models, including decision tree, random forest, XGBoost, AdaBoost, Gradient Boosting Decision Tree (GBDT), and Histogram Gradient Boosting Decision Tree (HGBDT), using a comprehensive SQL injection dataset. The primary motivation behind our approach is to leverage the strengths of ensemble learning and boosting techniques to enhance detection accuracy and robustness against SQL injection attacks. By systematically comparing these models, we aim to identify the most effective algorithms for SQL injection detection systems. Our experiments show that decision tree, random forest, and AdaBoost achieved the highest performance, with an accuracy of 99.50% and an F1 score of 99.33%. Additionally, we applied SHapley Additive exPlanations (SHAPs) and Local Interpretable Model-agnostic Explanations (LIMEs) for local explainability, illustrating how each model classifies normal and attack cases. This transparency enhances the trustworthiness of our approach to detecting SQL injection attacks. These findings highlight the potential of ensemble methods to provide reliable and efficient solutions for detecting SQL injection attacks, thereby improving the security of web applications.

Keywords: explained AI; SQL injection detection; decision tree; random forest; XGBoost; AdaBoost; gradient boosting decision tree; histogram gradient boosting decision tree; local explanation; SHAP; LIME

check for updates

Citation: Le, T.-T.-H.; Hwang, Y.; Choi, C.; Wardhani, R.W.; Putranto, D.S.C.; Kim, H. Enhancing Structured Query Language Injection Detection with Trustworthy Ensemble Learning and Boosting Models Using Local Explanation Techniques. *Electronics* **2024**, *13*, 4350. https://doi.org/10.3390/electronics13224350

Academic Editor: Christos J. Bouras

Received: 23 October 2024
Revised: 2 November 2024
Accepted: 5 November 2024
Published: 6 November 2024

1. Introduction

Structured Query Language injection (SQLi) remains one of the most critical security vulnerabilities affecting web applications, consistently ranking among the top security threats in reports like the OWASP Top Ten [1]. SQL injection exploits weaknesses in input validation, enabling attackers to manipulate SQL queries to access, modify, or even delete sensitive data stored in databases [2,3]. The widespread use and serious consequences of SQL injection attacks, such as data breaches, financial loss, and reputational damage to organizations, highlight the urgent need for effective detection mechanisms to safeguard web applications and their underlying databases.

Traditional approaches to mitigating SQL injection vulnerabilities, such as input validation, parameterized queries, and prepared statements, aim to sanitize user inputs and block malicious SQL commands from being executed [4–6]. However, these methods often fail in dynamically generated query environments and require rigorous implementation practices that are not always followed consistently [7]. This gap becomes more problematic as SQL injection techniques evolve and grow in sophistication, underscoring the need for more advanced and adaptable detection mechanisms capable of handling emerging threats.

Machine learning (ML) has emerged as a promising solution to improve SQL injection detection by learning historical attack patterns and generalizing to new threats [8]. Practical applications of ML in SQL injection detection include integration within web application firewalls (WAFs) and intrusion detection systems (IDSs), where they can analyze SQL query traffic in real-time to identify and respond to potential attacks. Among the various ML techniques, ensemble learning and boosting models have shown considerable potential in this domain, as they combine multiple weak learners into a stronger and more accurate classifier [9]. These models not only improve detection accuracy but also increase robustness against sophisticated SQL injection attacks, making them suitable for dynamic and complex web environments.

Ensemble learning methods, such as decision trees and random forests, take advantage of the collective wisdom of multiple models to achieve better performance than individual classifiers [10,11]. Boosting algorithms, including AdaBoost, Gradient Boosting Decision Trees (GBDTs), and Histogram Gradient Boosting Decision Trees (HGBDTs), further enhance this capability by sequentially focusing on misclassified instances, thereby improving the model's precision and recall [12–14]. These approaches offer a nuanced understanding of attack patterns, enabling them to adapt to new and sophisticated SQL injection techniques.

However, there remains a significant gap in the application of explainable machine learning models for SQL injection detection. Although numerous studies have focused on explaining the results of the model in the context of SQL injection detection [15,16], much of the research on explainable AI (XAI) has focused on other security domains, such as intrusion detection systems (IDSs) [10,13,17]. Techniques such as SHapley Additive exPlanations (SHAPs) [18] and Local Interpretable Model-agnostic Explanations (LIMEs) [19] have been widely applied in those fields to clarify model decisions and enhance transparency.

Despite this progress in other domains, explainable AI models have not yet been sufficiently explored or adapted for SQL injection detection. Given the critical importance of SQL injection attacks and the complexity of machine learning models to detect them, extending these XAI techniques to this domain is essential. By incorporating SHAP and LIME into SQL injection detection, we can improve the trustworthiness and interpretability of the model's decision-making process, thereby increasing confidence in its real-world deployment.

In this study, we focus on evaluating the performance of several ensemble learning and boosting models for SQL injection detection, specifically decision tree, random forest, XGBoost, AdaBoost, GBDT, and HGBDT. These models were selected due to their proven success in other security domains and their ability to handle large, complex datasets efficiently. Our objective is to enhance SQL injection detection systems by improving precision, recall, and overall robustness in identifying malicious SQL queries. To further increase trust and explain the proposed model decision, we incorporate well-known explainability techniques, such as SHAP and LIME, to provide transparent insights into the decision-making process of the models.

The primary contributions of this paper are as follows.

- We conduct a comparative analysis of six decision models, including decision tree, random forest, XGBoost, AdaBoost, GBDT, and HGBDT, using a comprehensive SQL injection dataset.
- We evaluate the performance of these models based on key metrics such as precision, recall, and F1 score, identifying the most effective machine learning techniques for SQL injection detection.

- We apply SHAP and LIME to explain the decision-making processes of each model, improving transparency and trustworthiness in SQL injection detection.
- We provide insights into the strengths and limitations of ensemble learning and boosting models for practical deployment in real-world SQL detection systems.

The remainder of this paper is structured as follows. Section 2 reviews related work on SQL injection detection using machine learning techniques. Section 3 outlines the methodology and experimental setup used for this study. Section 4 presents the experimental results, followed by a discussion in Section 5. Finally, Section 6 concludes this paper and suggests directions for future research.

2. Related Work

Over the past two decades, the detection and prevention of SQL injection (SQLi) attacks have been the focus of extensive research. This section reviews significant contributions to the field, particularly those employing machine learning (ML) techniques, to highlight the progress and challenges in SQLi detection.

2.1. Machine Learning Techniques for SQLi Detection

Initial efforts to mitigate SQL injection attacks primarily relied on static analysis and heuristic-based methods. These approaches involve code reviews and the implementation of best practices, such as input validation, parameterized queries, and prepared statements [2,4]. While effective in reducing vulnerabilities, these methods are often labor-intensive and prone to human error, especially in complex or dynamically generated queries.

The advent of machine learning has provided a more dynamic and adaptive approach to SQLi detection. ML models can learn from historical attack data to identify and mitigate new and evolving threats [20]. Several studies have explored the use of various ML algorithms for SQLi detection with promising results. Support Vector Machines (SVMs) and neural networks, for instance, have been employed to classify SQL queries as malicious or benign based on features extracted from query structures and patterns. Valeur et al. [21] proposed an anomaly-based intrusion detection system leveraging SVMs to detect SQL injection and other web attacks. Similarly, Gao et al. [22] used a neural network model trained on labeled datasets of SQL queries to achieve high detection accuracy.

In addition to these, decision trees and random forests have been widely used due to their interpretability and robustness. Xu et al. [23] demonstrated that decision tree classifiers could effectively distinguish between malicious and benign SQL queries by learning the characteristic patterns of SQL injection attacks. Random forests further enhance this capability by aggregating the predictions of multiple decision trees, improving accuracy and reducing overfitting [11].

Ensemble learning methods, which combine multiple weak learners to form a strong classifier, have shown significant potential in SQLi detection. AdaBoost and Gradient Boosting Decision Trees (GBDTs) are notable examples of boosting algorithms successfully applied in this domain. These models focus on instances that are difficult to classify, iteratively improving the classifier's performance. Pan et al. [24] used AdaBoost to enhance SQLi attack detection accuracy by adapting to new attack patterns. Le et al. [25] demonstrated the effectiveness of GBDT in handling complex datasets with imbalanced class distributions, achieving high-precision detection of SQLi attacks.

XGBoost and Histogram-based Gradient Boosting Decision Trees (HGBDTs) have further advanced the state of the art in SQLi detection. These models optimize the boosting process by efficiently handling large datasets and improving generalization capabilities. Chen and Guestrin [26] introduced XGBoost, a popular algorithm for SQLi detection due to its scalability and performance. Ke et al. [27] extended this approach with HGBDT, offering improved computational efficiency and accuracy for high-dimensional data.

Several comparative studies have evaluated the performance of different ML models for SQLi detection. These studies typically compare accuracy, precision, recall, and F1 score metrics to identify the most effective models. Nguyen et al. [28] comprehensively evaluated various ML algorithms, including decision trees, SVMs, and neural networks, for detecting SQL injection attacks, highlighting the superior performance of ensemble learning and boosting models.

Despite significant advancements, several challenges remain in detecting SQL injection attacks using ML techniques. One major challenge is the imbalanced nature of datasets, where non-malicious queries vastly outnumber malicious ones [29]. This imbalance can lead to biased models that favor the majority class. Techniques such as oversampling, undersampling, and synthetic data generation have been proposed to address this issue [30].

Another challenge is the evolving nature of SQL injection attacks. Attackers continuously develop new techniques to bypass existing detection mechanisms, requiring models that can adapt to these changes [31]. Continuous learning and real-time monitoring systems are potential solutions to enhance the adaptability and effectiveness of ML-based detection systems [8].

2.2. Challenges with Explainable AI for SQL Injection Detection

While machine learning models have shown great promise in detecting SQLi attacks, the challenge of interpretability and trustworthiness remains a key barrier to their widespread adoption. SQL injection detection, especially in security-critical web applications, requires not only high-performance models, but also clear explanations of how these models make decisions. Explainable AI (XAI) has emerged as a solution to this problem, offering tools to interpret and visualize machine learning models. However, applying XAI in the context of SQL injection detection brings specific challenges.

One significant challenge in applying explainable AI models like SHapley Additive exPlanations (SHAPs) [18] and Local Interpretable Model-agnostic Explanations (LIMEs) [19] to SQLi detection lies in the inherent complexity and variability of SQLi attacks. SQLi attacks often involve obfuscated, multi-layered queries, making it difficult to isolate the specific features or patterns that contribute to a detection decision. This obfuscation can hinder the creation of meaningful explanations that security analysts can easily interpret.

Moreover, one of the main objectives of applying XAI in SQLi detection is to increase trust in the predictions of the model. In real-world deployments, security teams need to understand not only which features contributed to a prediction but also why the model classified a specific SQL query as malicious. Current XAI methods provide local explanations, which provide insight for individual queries but may not explain broader patterns or attack vectors. For example, explaining why one SQL query was flagged as malicious does not always provide insights into other, potentially related attacks. This limitation underscores the challenge of developing XAI methods that not only provide accurate local explanations but also offer global insights into SQLi attack trends.

In summary, while traditional methods provide a foundational layer of security, integrating machine learning techniques, particularly ensemble learning and boosting models, offers a more robust and dynamic approach to SQLi detection. The current research landscape highlights the effectiveness of these models in identifying and mitigating SQL injection attacks, paving the way for further advancements in this critical area of cybersecurity. However, the application of XAI in this domain remains a challenge due to the complex nature of SQLi attacks, which requires future research to address interpretability issues and improve trust in AI-driven security systems.

3. Methodology

This section presents our methodology for detecting SQL injection using ensemble learning and boosting models, depicted in Figure 1. The process begins with data preprocessing, where we encode string data into a numerical format using a Label Encoder and split the dataset into training and testing sets. We train six models, decision tree, random forest, XGBoost, AdaBoost, Gradient Boosting Decision Tree (GBDT), and Histogram Gradient Boosting Decision Tree (HGBDT) on the training dataset. These models are then saved as pretrained models for inference on the testing dataset. Finally, we evaluate the performance of our approach using key metrics such as confusion matrix, Receiver Operating Characteristic (ROC) curve, precision, recall, accuracy, and F1 score on both the training and testing datasets.

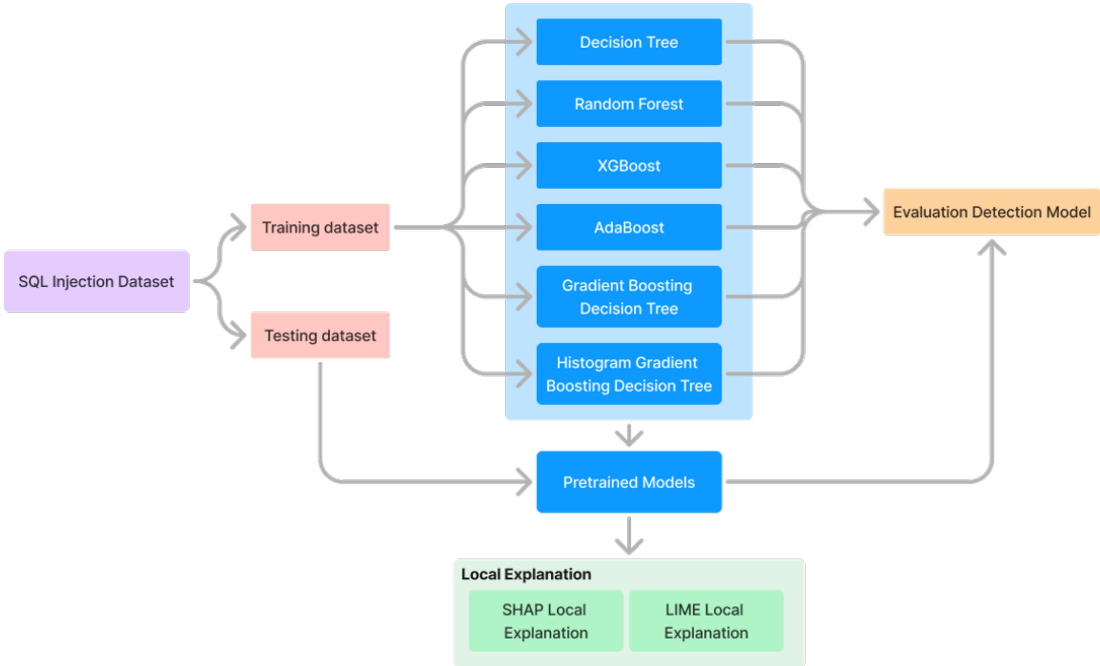


Figure 1. The proposed method for detecting SQL injection from ensemble learning and boosting models.

3.1. Data Description

This dataset is sourced from the SQL injection attack dataset [32]. It contains raw SQL query strings labeled with integers in the label column, where a value of 0 indicates a non-malicious query, and a value of 1 denotes a malicious query.

Figure 2 illustrates the distribution of sample data labeled as attacks and non-attacks in the SQL injection dataset. The dataset comprises approximately 12,000 attack samples and around 20,000 non-attack samples. To evaluate the performance of our models, we partitioned the dataset into training and testing sets in a ratio of 70:30. Furthermore, we used a 5-fold cross-validation strategy during model training to enhance the robustness of our evaluation. This approach involves splitting the training set into five subsets, allowing each model to be trained and validated on different combinations of these subsets. By averaging the performance metrics across all folds, we ensured a more reliable estimate of the model's effectiveness in detecting SQL injection attacks.

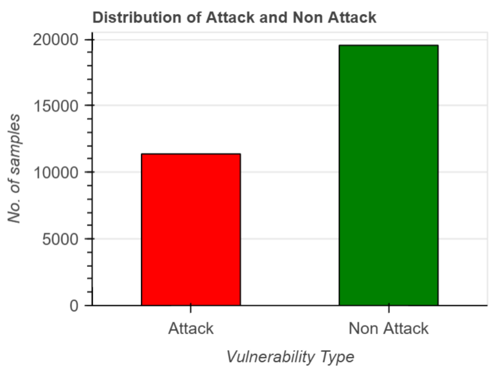


Figure 2. Distribution sample data.

3.2. SQL Vulnerability-Detection-Based Ensemble and Bagging Models

The rationale for selecting these models lies in their diverse strengths and complementary capabilities. Decision trees provide straightforward interpretation and quick inference, making them useful for initial insights and rapid prototyping. random forests, an ensemble of decision trees, enhance performance by reducing overfitting and improving generalization. Boosting models like XGBoost and AdaBoost iteratively focus on misclassified instances, increasing overall model accuracy and robustness. GBDT and HGBDT extend boosting methods by efficiently handling large datasets and high-dimensional feature spaces, making them suitable for complex real-world applications.

- Decision tree: This model has hierarchical structures where each node represents a feature, and each branch represents a decision rule based on that feature. The decision tree recursively partitions the feature space into regions that minimize a splitting criterion (e.g., Gini impurity or entropy).
- Random forest: This is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes (classification) or average prediction (regression) of the individual trees. Let T denote the number of trees in the forest. For classification, the random forest combines the predictions $\hat{y}_{(t)}$ of each tree t by voting: $\hat{y} = mode(\hat{y}_{(1)}, \ldots, \hat{y}_{(t)})$.
- XGBoost (Extreme Gradient Boosting): XGBoost is an optimized gradient-boosting library known for its speed and performance. It sequentially builds trees, where each subsequent tree corrects errors made by the previous one. XGBoost minimizes the loss function $L(\phi)$ iteratively by adding weak learners h_t to the model, $\hat{y} = \phi(x) = \sum_{t=1}^{T} h_t(x)$, where T is the number of boosting rounds.
- AdaBoost: AdaBoost is another ensemble learning method that combines multiple weak classifiers to create a strong classifier. It adjusts the weights of incorrectly classified instances so that subsequent classifiers focus more on difficult cases. At each iteration t, AdaBoost updates the weights $D_t(i)$ of training instances and computes the model weight α_t based on the classification error. The final classifier is a weighted sum: $\hat{y} = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$.
- GBDT: GBDT builds trees sequentially, where each tree attempts to correct errors made by the previous one. Unlike AdaBoost, which adjusts instance weights, GBDT fits each new tree to the residual errors of the current model predictions. GBDT minimizes the loss function by adding new trees that approximate the negative gradient of the loss function for the ensemble model, $\hat{y} = \sum_{t=1}^{T} h_t(x)$, where h_t is the t-th decision tree.
- HGBDT: HGBDT is an optimized version of GBDT that uses histograms to discretize continuous features, reducing memory usage and speeding up training. Similar to GBDT, HGBDT constructs an ensemble model by iteratively adding decision trees that minimize the loss function, $\hat{y} = \sum_{t=1}^{T} h_t(x)$, where each tree h_t is trained on histogram-based feature representations.

3.3. Local Explanation for SQL Injection Model Decision Based on SHAP and LIME

In the context of SQL injection detection, understanding why a machine learning model classifies an SQL query as normal or malicious is crucial to ensuring the reliability and robustness of the system. Local explanation techniques, such as SHAP and LIME, are widely used to explain individual predictions. These methods help to elucidate the decision-making process of the model by providing feature attributions for a specific instance. This section details how SHAP and LIME can be used to explain the behavior of machine learning models to detect SQL injection attacks.

3.3.1. SHAP: SHapley Additive exPlanations

SHAP is a method based on cooperative game theory, specifically Shapley values, which provides a theoretically grounded approach to explain the prediction of a model. Shapley values calculate the marginal contribution of each feature to the prediction of the model by considering all possible combinations of features.

Mathematically, the Shapley value for a feature i is defined as

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} (f(S \cup \{i\}) - f(S)) \quad (1)$$

where N is the set of all features, $S \subseteq N \setminus \{i\}$ is a subset of features, excluding feature i, $f(S)$ is the model's prediction when only the features in S are considered, $f(S \cup \{i\})$ is the model's prediction when feature i is added to S, and ϕ_i represents the contribution of feature i to the model's prediction.

For our SQL injection detection model, SHAP values can be computed for each feature in the SQL query, such as SQL keywords (e.g., 'SELECT', 'UNION'), query length, and query structure. The SHAP values indicate how each feature contributes to classifying the query as either normal or malicious.

To visualize these attributions, we use SHAP's `force plot`, which provides an intuitive representation of the forces that push the model's prediction towards a certain class. The force plot for two cases—one for a normal query and one for an SQL injection attack—illustrates how specific features (e.g., keywords or anomalous patterns) drive the decision.

3.3.2. LIME: Local Interpretable Model-Agnostic Explanations

LIME provides local explanations by approximating the decision boundary of the original model with an interpretable surrogate model, such as a linear regression or a decision tree, in the neighborhood of the instance to be explained. LIME works by generating perturbed samples around the instance of interest and fitting a local model to explain how the original model predictions change in response to those perturbations.

The prediction for an instance x is explained by solving the following minimization problem:

$$\xi(x) = \arg\min_{g \in G} \mathcal{L}(f, g, \pi_x) + \Omega(g) \quad (2)$$

where f is the original model, g is the interpretable surrogate model from the class of models G (e.g., a linear model), $\mathcal{L}(f, g, \pi_x)$ is the loss function that measures how well the surrogate model g approximates the original model f in the local neighborhood defined by π_x, and $\Omega(g)$ is a complexity term that ensures the surrogate model remains interpretable.

For SQL injection detection, we apply `LimeTabularExplainer` to both normal and malicious SQL queries. `LimeTabularExplainer` generates new data points by randomly perturbing the original query, and then examines how the model responds to these variations. Build a local interpretable model to approximate the decision-making process for that specific query.

4. Experiment

Our models are developed in Python, utilizing widely used machine learning libraries such as Scikit-learn. The experiments are performed on a computer with the following specifications: an Intel Core i7-10700K CPU @ 3.80 GHz processor, 64 GB RAM, and a Windows 10 operating system.

4.1. Experiment Setting

Table 1 presents a comprehensive overview of the hyperparameters utilized across several prominent ensemble learning and boosting models, including the XGBClassifier, RandomForestClassifier, DecisionTreeClassifier, GradientBoostingClassifier, HistGradientBoostingClassifier, and AdaBoostClassifier. Understanding these hyperparameters is crucial, as they directly influence the learning dynamics and predictive capabilities of each model.

Table 1. Hyperparameters for ensemble learning and boosting models.

Model Name	Hyperparameter	Value
XGBoost	`enable_categorical`	False
	`n_estimators`	100
Random Forest	`n_estimators`	100
	`criterion`	gini
	`max_depth`	None
	`min_samples_split`	2
	`min_samples_leaf`	1
	`max_features`	auto
Decision Tree	`criterion`	gini
	`splitter`	best
	`max_depth`	None
	`min_samples_split`	2
	`random_state`	42
GBDT	`loss`	deviance
	`learning_rate`	0.1
	`n_estimators`	100
	`subsample`	1.0
	`criterion`	friedman_mse
	`min_samples_split`	2
	`min_samples_leaf`	1
	`max_depth`	3
	`max_features`	None
	`random_state`	42
HGBDT	`loss`	auto
	`learning_rate`	0.1
	`max_iterations`	100
	`max_leaf_nodes`	31
	`max_depth`	None
	`min_samples_leaf`	20
	`l2_regularization`	0.0
	`max_bins`	255
	`early_stopping`	auto
	`scoring`	loss
	`random_state`	42
AdaBoost	`n_estimators`	50
	`learning_rate`	1.0
	`algorithm`	SAMME.R
	`base_estimator`	DecisionTreeClassifier
	`random_state`	None
	`estimator_params`	()

The key hyperparameters detailed in this table encompass the number of estimators, which governs the ensemble size; the learning rate, which controls the step size during optimization; and various structural parameters, such as the maximum depth of the trees, which can help prevent overfitting.

- Number of Estimators: This determines the ensemble size in models like XGBClassifier and RandomForestClassifier. While more estimators can enhance accuracy by aggregating predictions, they also increase computational demands and may lead to overfitting.
- Learning Rate: Specified in GBDT and HGBDT, the learning rate controls how much each estimator contributes to the final prediction. A lower learning rate generally improves robustness but requires more estimators, whereas a higher rate can speed up convergence but risks instability.
- Max Depth:Limiting the maximum depth of trees in models such as decision tree and random forest helps to prevent overfitting. Deeper trees capture complex patterns but may lead to overfitting if not regulated.
- Min Samples Split and Min Samples Leaf: These parameters help to prevent overfitting by setting thresholds for the minimum number of samples needed to create a new split or leaf node.
- Criterion: The choice of evaluation metric (e.g., 'gini' or 'friedman_mse') affects how splits are made in tree-based models, impacting their ability to differentiate between classes.
- Loss Function: The specified loss function in models like GBDT defines the optimization target, influencing convergence and performance.
- Regularization: Parameters like `l2_regularization` in HGBDT help mitigate overfitting by penalizing large coefficients, enhancing generalization.

4.2. Evaluation Metrics

To assess the performance of our binary classifier, we use several evaluation metrics, including the confusion matrix, precision, recall, F1 score, accuracy, and Receiver Operating Characteristic (ROC) curve. Each of these metrics provides valuable insights into the model's performance and its ability to classify instances correctly.

The confusion matrix is a table that summarizes the performance of a classification algorithm. It displays the counts of true positive (TP), false positive (FP), true negative (TN), and false negative (FN) predictions as follows:

The precision known as the positive predictive value measures the precision of the positive predictions. It is calculated as

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3}$$

Recall, also known as sensitivity or true positive rate, measures the ability of the model to identify positive instances. It is defined as

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4}$$

The F1 score is the harmonic mean of precision and recall, providing a single metric that balances both concerns. It is computed as

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5}$$

Accuracy measures the overall correctness of the model by calculating the ratio of correctly predicted instances to the total instances:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{6}$$

Receiver Operating Characteristic (ROC) Curve: The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The TPR and FPR are defined as

$$\text{TPR} = \frac{TP}{TP + FN} \tag{7}$$

$$\text{FPR} = \frac{FP}{FP + TN} \tag{8}$$

The area under the ROC curve (AUC-ROC) is a single scalar value that summarizes the performance of the classifier across all thresholds, where a higher AUC indicates better performance.

These evaluation metrics collectively provide a comprehensive view of the performance of the model, allowing us to make informed decisions regarding its performance classification task at hand.

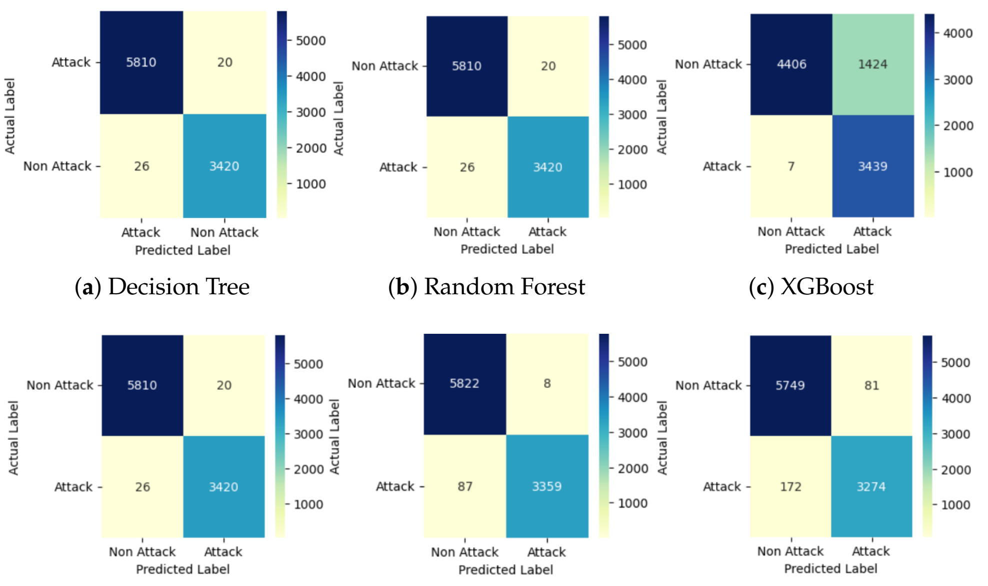
4.3. Threshold Levels for Metric Calculations

To ensure consistent baseline comparisons across models, we calculated the metrics (accuracy, precision, recall, and F1 score) using a decision threshold of 0.5, which is commonly applied in binary classification tasks. This threshold means that any instance with a predicted probability $\geq$0.5 was classified as a positive case (attack), while probabilities $<$0.5 were classified as negative cases (normal).

For metrics sensitive to threshold adjustments, such as the ROC-AUC, we assessed the model's performance across varying thresholds to capture the trade-off between the true positive rate (TPR) and false positive rate (FPR). This approach allows us to evaluate the robustness of each model under different conditions, providing a comprehensive understanding of the model's capability to detect SQL injection attacks.

4.4. Confusion Matrix Results

This section presents the confusion matrix results for our models in classifying SQL injection attacks and non-attacks. The decision tree, random forest, and AdaBoost models exhibit the best detection performance, closely followed by the GBDT model. In contrast, XGBoost demonstrates the least accuracy, misclassifying 1424 non-attack samples as attacks (Figure 3c).



(**a**) Decision Tree (**b**) Random Forest (**c**) XGBoost

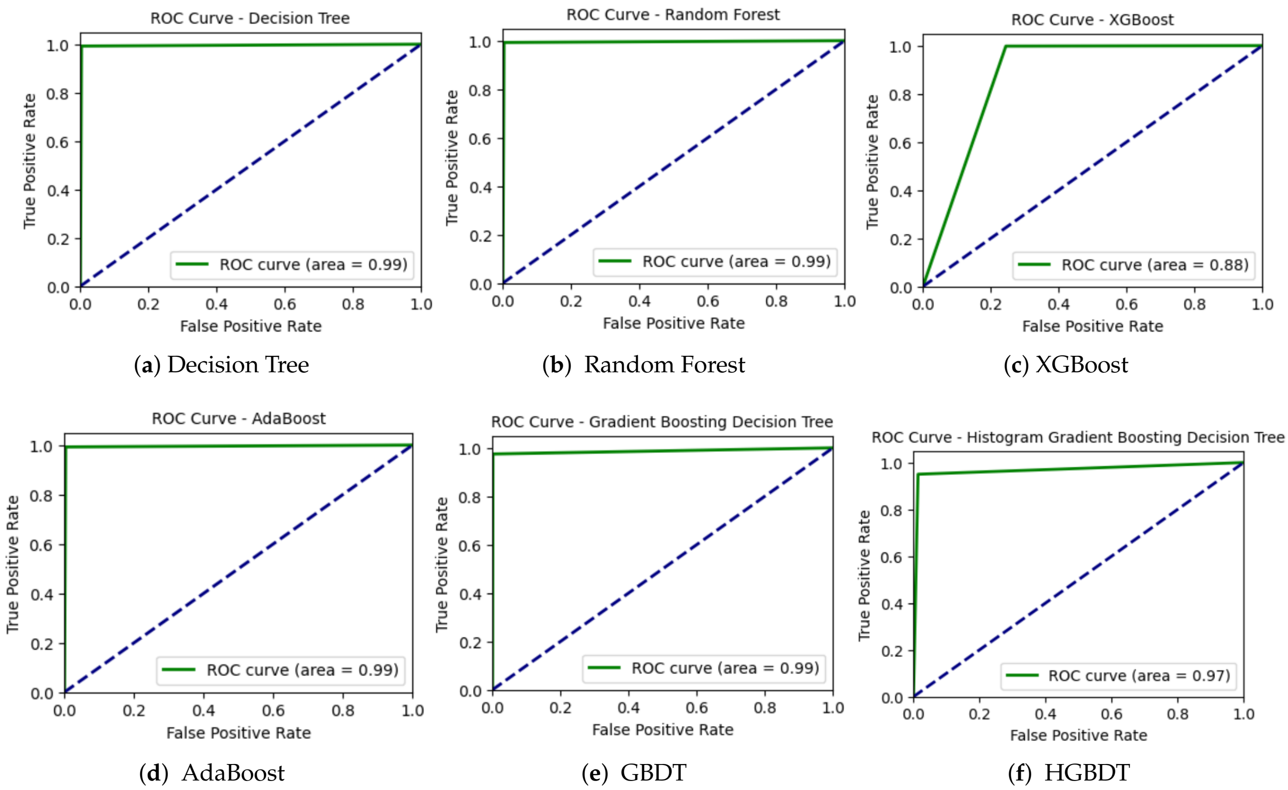
(**d**) AdaBoost (**e**) GBDT (**f**) HGBDT

Figure 3. Confusion matrix results of ensemble learning and boosting models.

4.5. ROC Curve Results

We analyze the diagnostic performance of a binary classifier system by varying its discrimination threshold. The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) across various threshold settings. The area under the ROC curve (AUC-ROC) serves as a concise metric of the predictive accuracy, where a higher AUC-ROC signifies the superior performance of the model.

Figure 4 presents the ROC measurements for six models. The results indicate that XGBoost achieves an AUC-ROC of 88%, while the remaining models achieve robust AUC-ROC values exceeding 97%.



(**a**) Decision Tree (**b**) Random Forest (**c**) XGBoost

(**d**) AdaBoost (**e**) GBDT (**f**) HGBDT

Figure 4. ROC curve results of ensemble learning and boosting models.

4.6. Other Performance Evaluation Matrix Results

This section comprehensively analyzes the performance of our models using several evaluation metrics, including precision, recall, accuracy, and the F1 score. The evaluation results, summarized in Table 2, illustrate the performance classification of each model in detecting SQL injection attacks.

Table 2. Detailed classification evaluation results.

Model	Precision	Recall	Accuracy	F1	AUC ROC
Decision Tree	0.9942	0.9925	0.9950	0.9933	0.99
Random Forest	0.9942	0.9925	0.9950	0.9933	0.99
XGBoost	0.7072	0.9980	0.8457	0.8278	0.88
AdaBoost	0.9942	0.9925	0.9950	0.9933	0.99
GBDT	0.9976	0.9748	0.9898	0.9861	0.99
HGBDT	0.9759	0.9501	0.9727	0.9628	0.97

The decision tree, random forest, and AdaBoost models demonstrate exceptional performance, each achieving a precision of 99.42%, a recall of 99.25%, an accuracy of 99.50%, and an F1 score of 99.33%. These results indicate that these models are highly effective in identifying SQL injection attacks while maintaining a low rate of false positives.

In contrast, XGBoost, while achieving a high recall of 99.80%, has a lower precision of 70.72%, resulting in an overall accuracy of 84.57% and an F1 score of 82.78%. This suggests that although XGBoost is good at detecting almost all attack instances, it produces more false positives than the other models.

The GBDT model shows strong performance with a precision of 0.9976, a recall of 97.48%, an accuracy of 98.98%, and an F1 score of 98.61%. This indicates that GBDT is also a reliable model, though slightly less accurate than decision tree, random forest, and AdaBoost.

Finally, the HGBDT model, while still effective, lags slightly behind the top performers with a precision of 97.59%, a recall of 95.01%, a precision of 97.27%, and an F1 score of 96.28%. These metrics show that HGBDT is a solid model, but not as robust as the other ensemble methods evaluated.

Our analysis highlights that ensemble methods like random forest and AdaBoost, along with the decision tree model, provide the most reliable performance detecting SQL injection attacks. These findings emphasize the importance of model selection and tuning in the development of effective cybersecurity solutions.

4.7. Comparison with Existing Methods

Table 3 presents a performance comparison between our approach and several existing methods for SQL injection detection.

Table 3. Comparison of classification performance with other prior methods.

Model	Precision	Recall	Accuracy	F1	AUC ROC
Logistic Regression [33]	-	-	99.3	-	-
SQLIA [34]	97.4	99.7	98	98.5	99.9
CNN [35]	-	-	99.6	-	-
Naive Bayes [36]	94.19	-	98.33	97.00	97.71
Ours	99.42	99.25	99.50	99.33	99

Our model demonstrates competitive or superior performance across multiple metrics. With an accuracy of 99.50%, our method slightly outperforms the CNN model (99.6%) [35] and Logistic Regression (99.3%) [33]. The F1 score of our model (99.33%) surpasses that of SQLIA (98.5%) [34] and Naive Bayes (97.00%) [36]. While SQLIA reports a marginally higher ROC (99.9%) compared to our 99%, our model maintains a better overall balance across all metrics. These results underscore the performance accuracy of our approach in the context of current SQL injection detection techniques.

4.8. Local Explanation Results for Model Decision

To interpret the decisions made by our ensemble and boosting models, we apply SHAP and LIME values on two types of data samples: attack samples and normal samples. For SHAP, we utilize a force plot to display SHAP values, which highlight feature contributions to model predictions. For LIME, we visualize prediction probabilities alongside feature contributions, helping to explain the factors that influence each model's decision on sample classification.

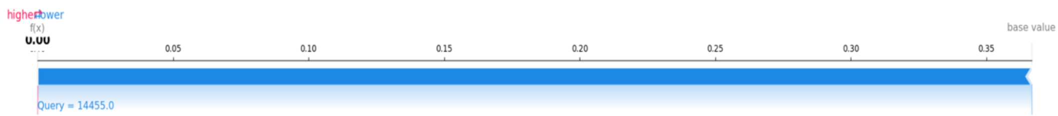
First, we examine the SHAP values for the ensemble models, including the decision tree and random forest, as shown in Figure 5. Next, we interpret the SHAP values for boosting models, including XGBoost, AdaBoost, GBDT, and HGBDT, as illustrated in Figure 6. In these SHAP results, positive SHAP values (displayed in red) indicate that the model classifies the sample as an attack, while negative SHAP values (displayed in blue) suggest a normal classification.

The decision tree model classifies a testing sample as an attack when the SHAP value reaches 1, with a query value of 18,814 (refer to Figure 5a), while the random forest model classifies a sample as an attack when the SHAP value is 1, with a query value of 11,566 (Figure 5c). Similarly, both models classify a sample as normal when the SHAP value is 0, with query values of 14,455 and 26,773 for the decision tree and random forest, respectively, as shown in Figure 5b,d. Similarly, we explain the SHAP and query values for XGBoost, AdaBoost, GBDT, and HGBDT in detail in Table 4.

Table 4. SHAP values for classification models on sample data.

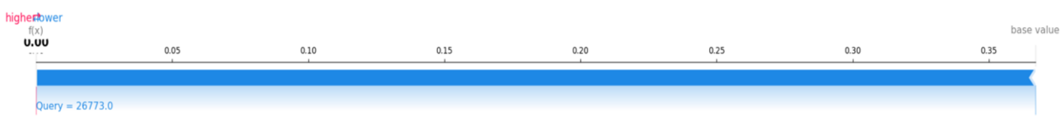
Model	Sample Data Label	SHAP Value	Query Value
Decision Tree	Attack	1	18,814
	Normal	0	14,455
Random Forest	Attack	1	11,566
	Normal	0	26,773
XGBoost	Attack	14.84	1.157×10^4
	Normal	-5.23	2.677×10^4
AdaBoost	Attack	0.52	1.157×10^4
	Normal	-8.76	1.446×10^4
GBDT	Attack	5.17	1.157×10^4
	Normal	-2.96	2.677×10^4
HGBDT	Attack	9.41	1.157×10^4
	Normal	-6.44	2.677×10^4

(**a**) Interpreting SHAP values of decision tree for attack. sample



(**b**) Interpreting SHAP values of decision tree for normal sample.

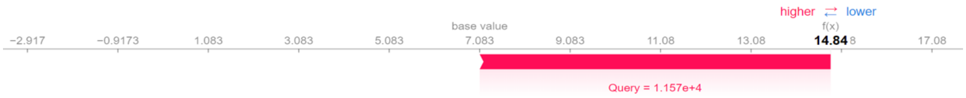
(**c**) Interpreting SHAP values of random forest for attack sample.



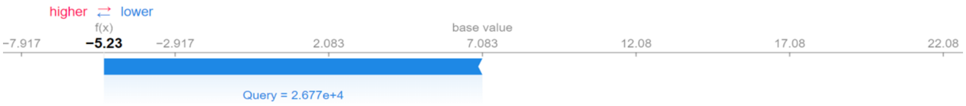
(**d**) Interpreting SHAP values of random forest for normal sample.

Figure 5. Interpreting SHAP values in ensemble models for normal and attack samples.

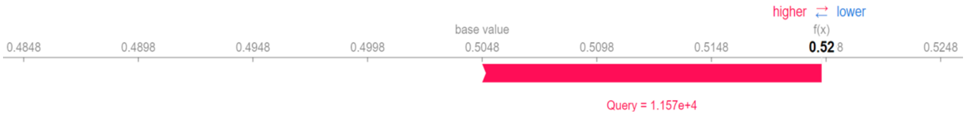
Next, we visualize the interpretation of LIME values for six models, as shown in Figures 7–12. The LIME values for each model, as shown in Table 5, illustrate how different classifiers interpret SQL queries as either attacks or normal based on query ranges and the sign of the LIME values. A positive LIME value (in green) across all models indicates an attack classification, while a negative LIME value (in red) points to a normal classification. For example, in the decision tree and random forest models, attack classifications generally occur in the query range of [7707.50–15,527.00] with LIME values up to 0.37, whereas normal classifications are associated with higher query values (>23,220.50) and negative LIME values. This pattern is consistent across models, though some, like GBDT and HGBDT, show slightly higher LIME value ranges for attacks, indicating subtle model-specific variations in interpretability.



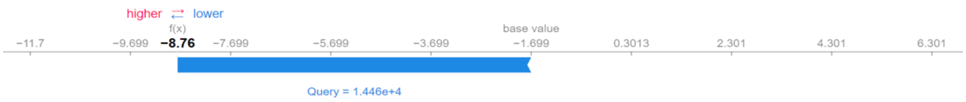
(**a**) Interpreting SHAP values of XGBoost for attack sample.



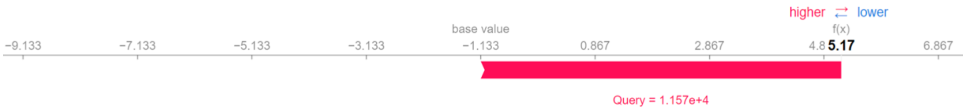
(**b**) Interpreting SHAP values of XGBoost for normal sample.



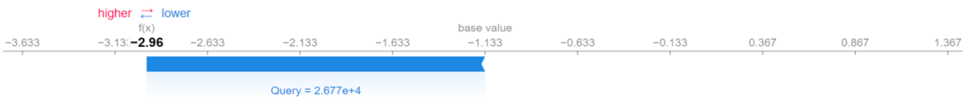
(**c**) Interpreting SHAP values of AdaBoost for attack sample.



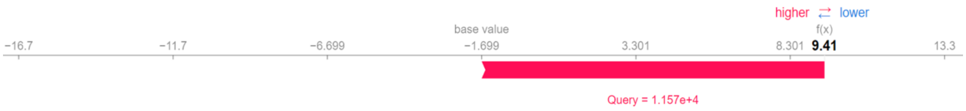
(**d**) Interpreting SHAP values of AdaBoost for normal sample.



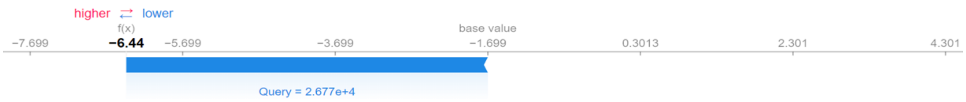
(**e**) Interpreting SHAP values of GBDT for attack sample.



(**f**) Interpreting SHAP values of GBDT for normal sample.

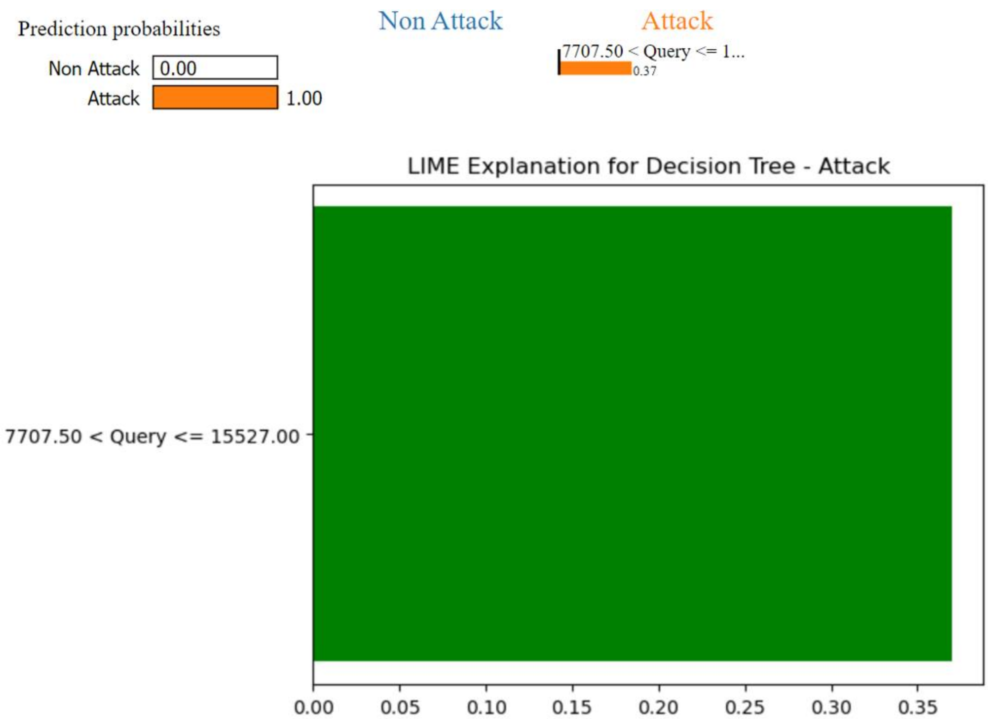


(**g**) Interpreting SHAP values of HGBDT for attack sample.

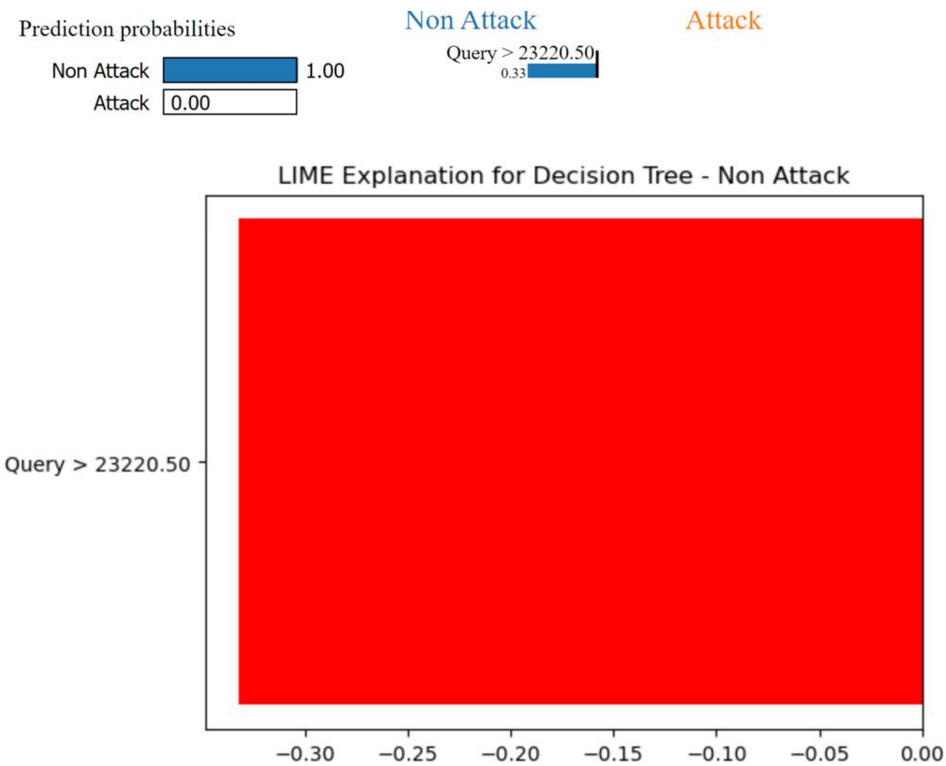


(**h**) Interpreting SHAP values of HGBDT for normal sample.

Figure 6. Interpreting SHAP values in boosting models for normal and attack samples.

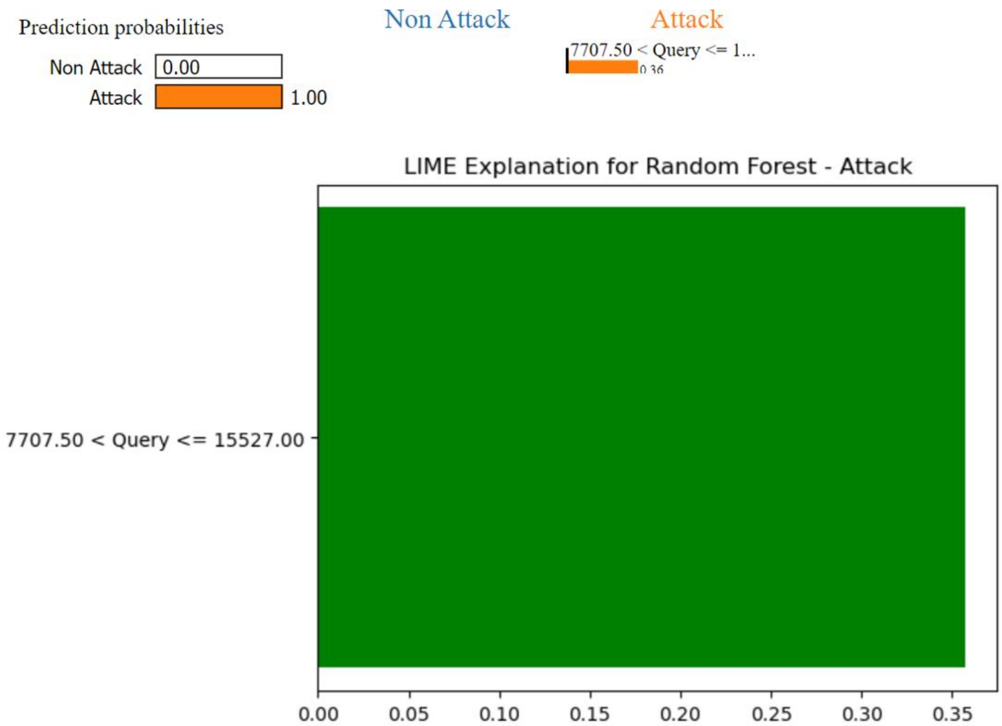


(**a**) Interpreting LIME values of decision tree for attack sample.

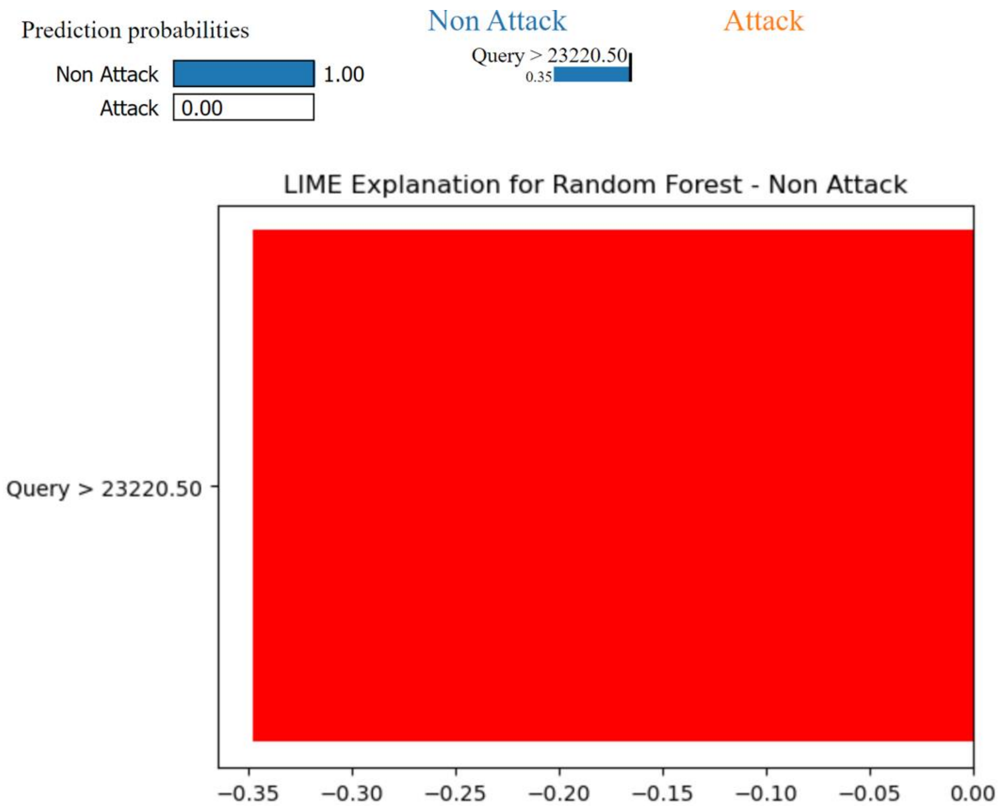


(**b**) Interpreting LIME values of decision tree for normal sample.

Figure 7. Interpreting SHAP values in decision tree model for normal and attack samples.

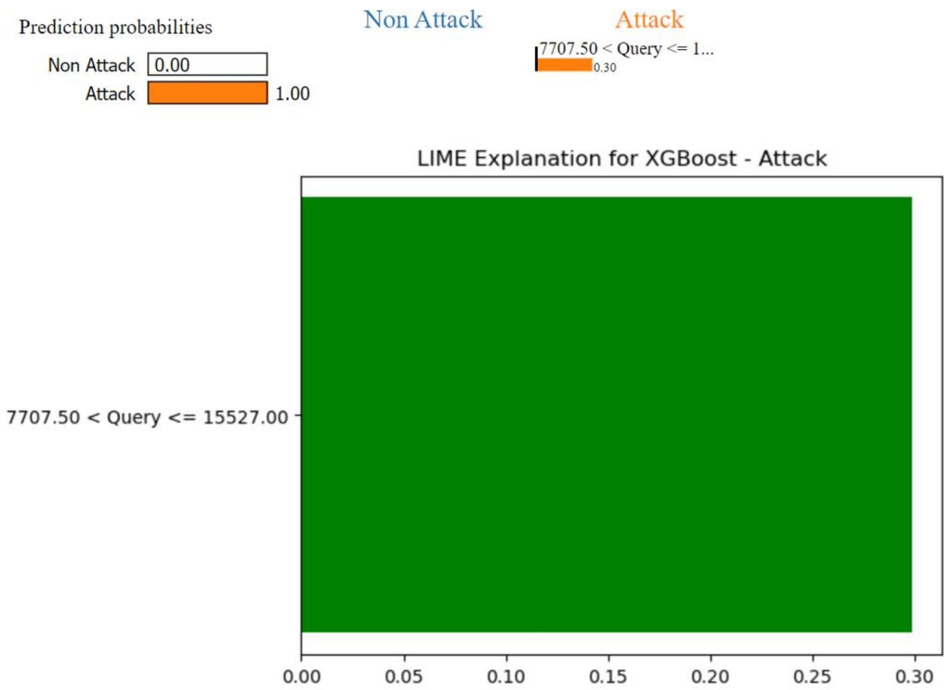


(**a**) Interpreting LIME values of random forest for attack sample.

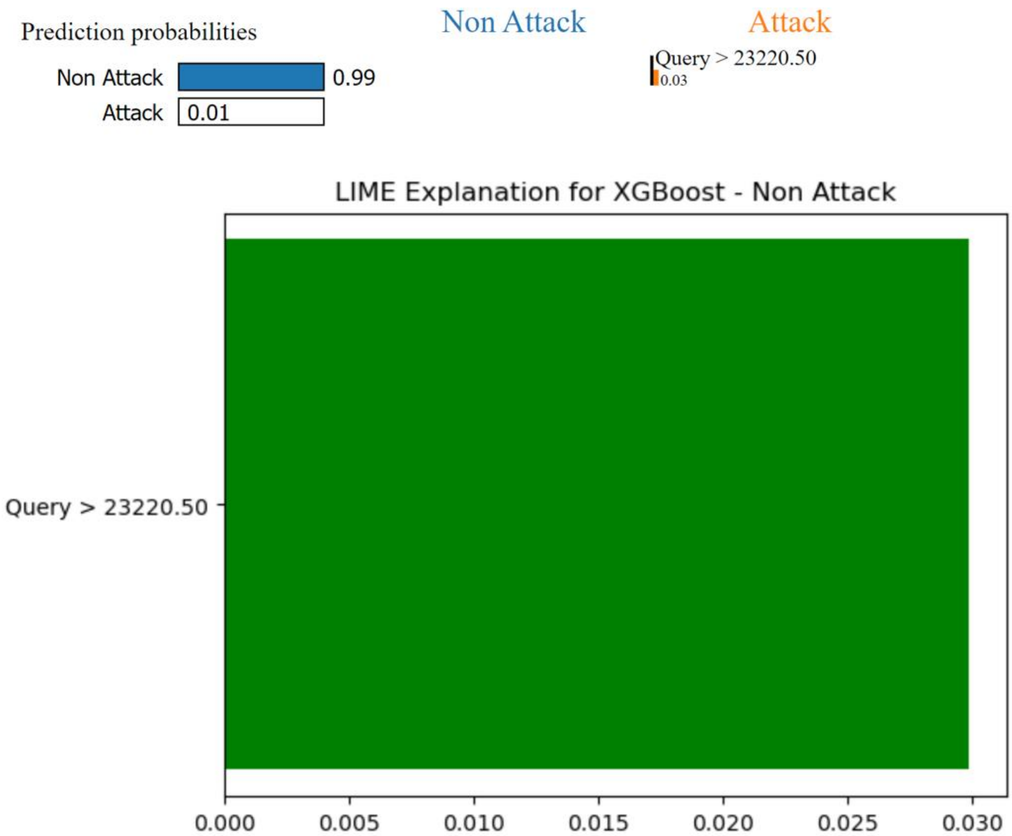


(**b**) Interpreting LIME values of random forest for normal sample.

Figure 8. Interpreting SHAP values in random forest model for normal and attack samples.

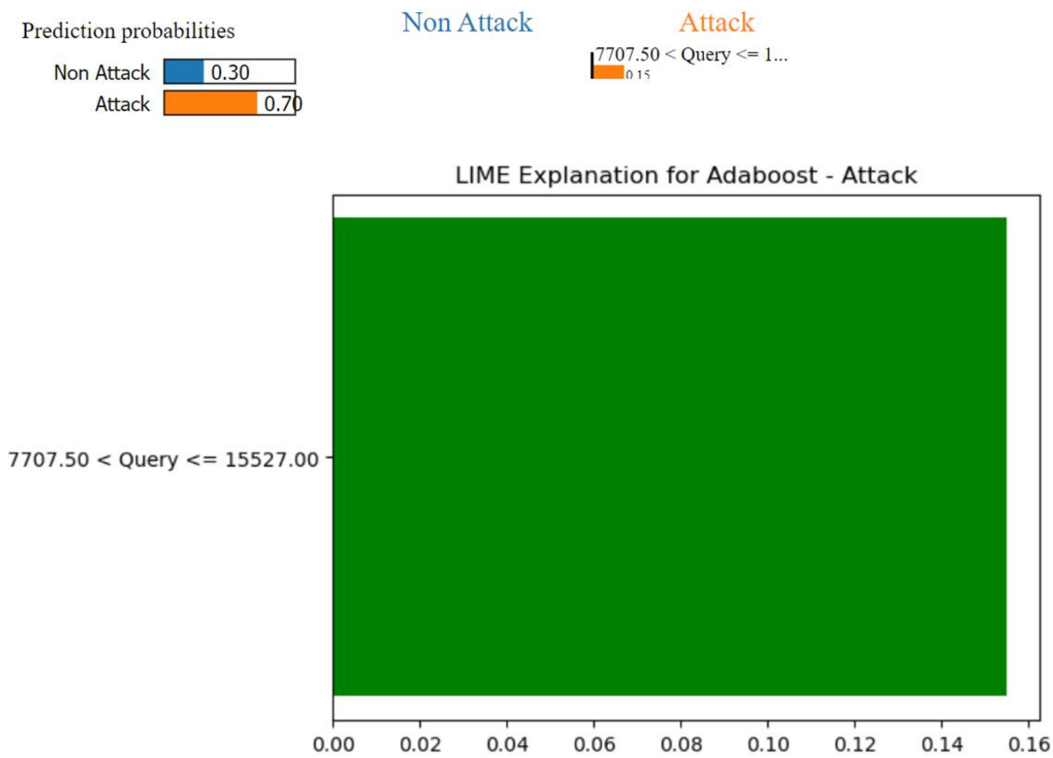


(**a**) Interpreting LIME values of XGBoost for attack sample.

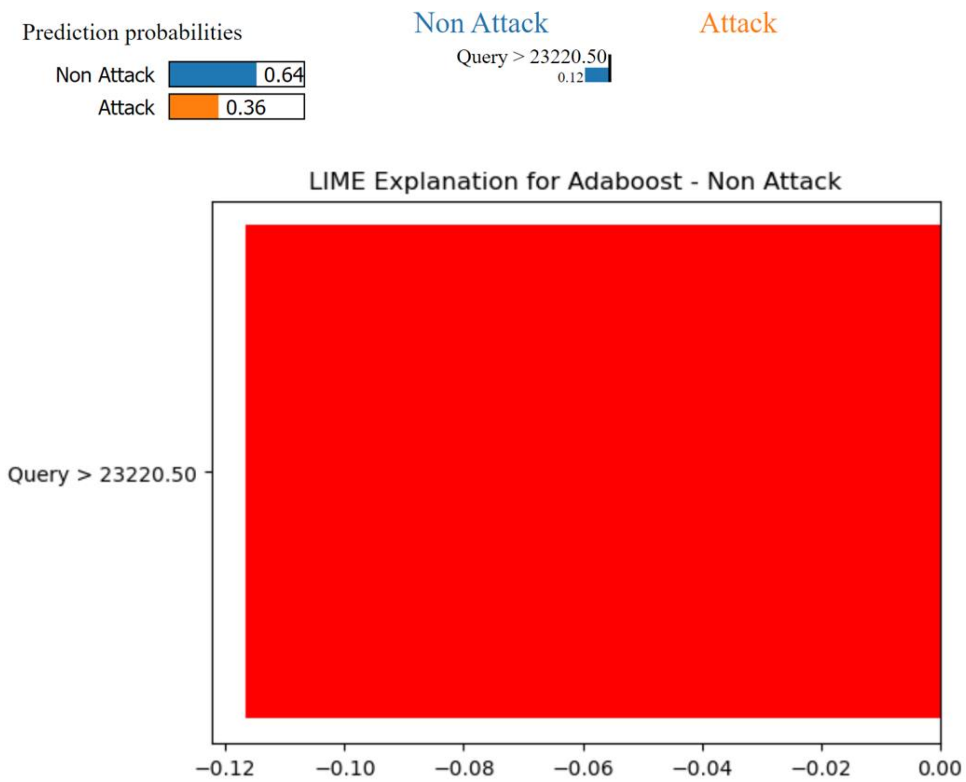


(**b**) Interpreting LIME values of XGBoost for normal sample.

Figure 9. Interpreting SHAP values in XGBoost model for normal and attack samples.

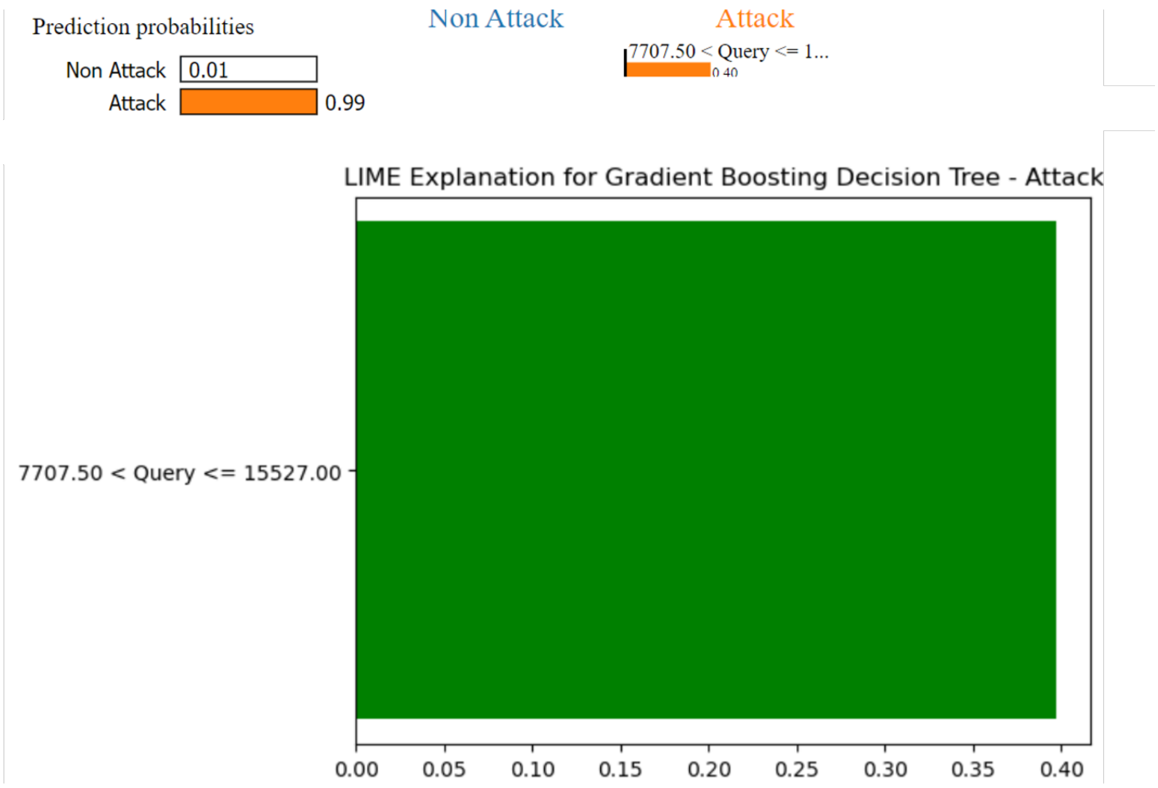


(**a**) Interpreting LIME values of AdaBoost for attack sample.

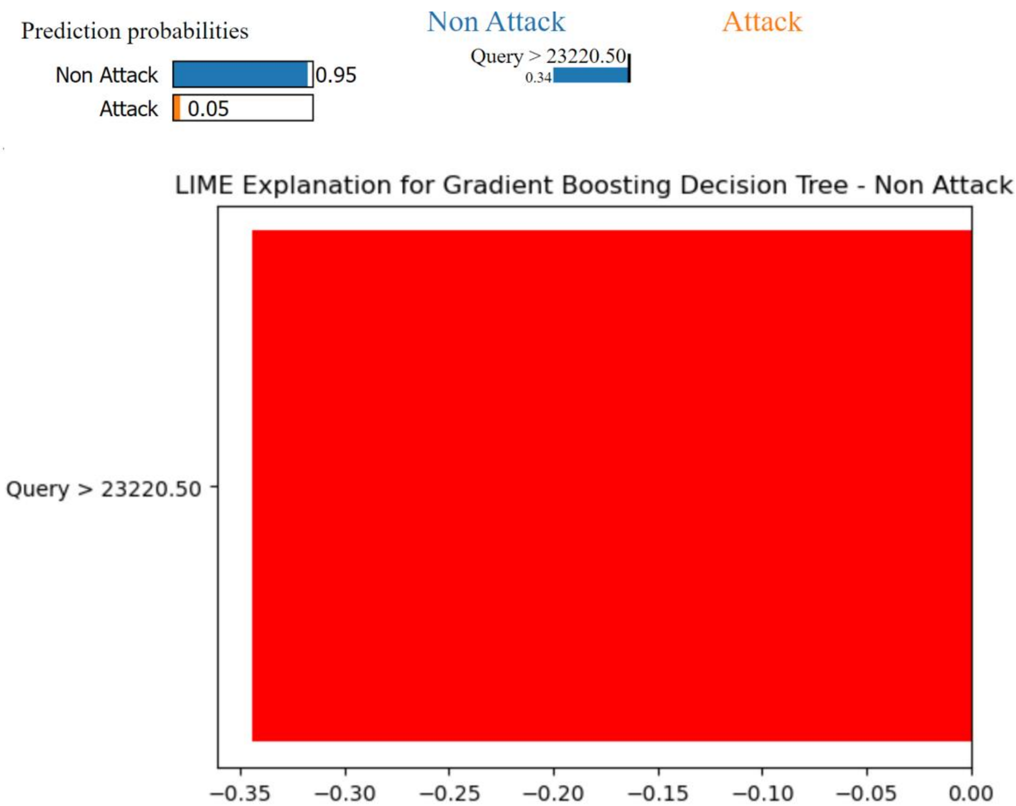


(**b**) Interpreting LIME values of AdaBoost for normal sample.

Figure 10. Interpreting SHAP values in AdaBoost model for normal and attack samples.

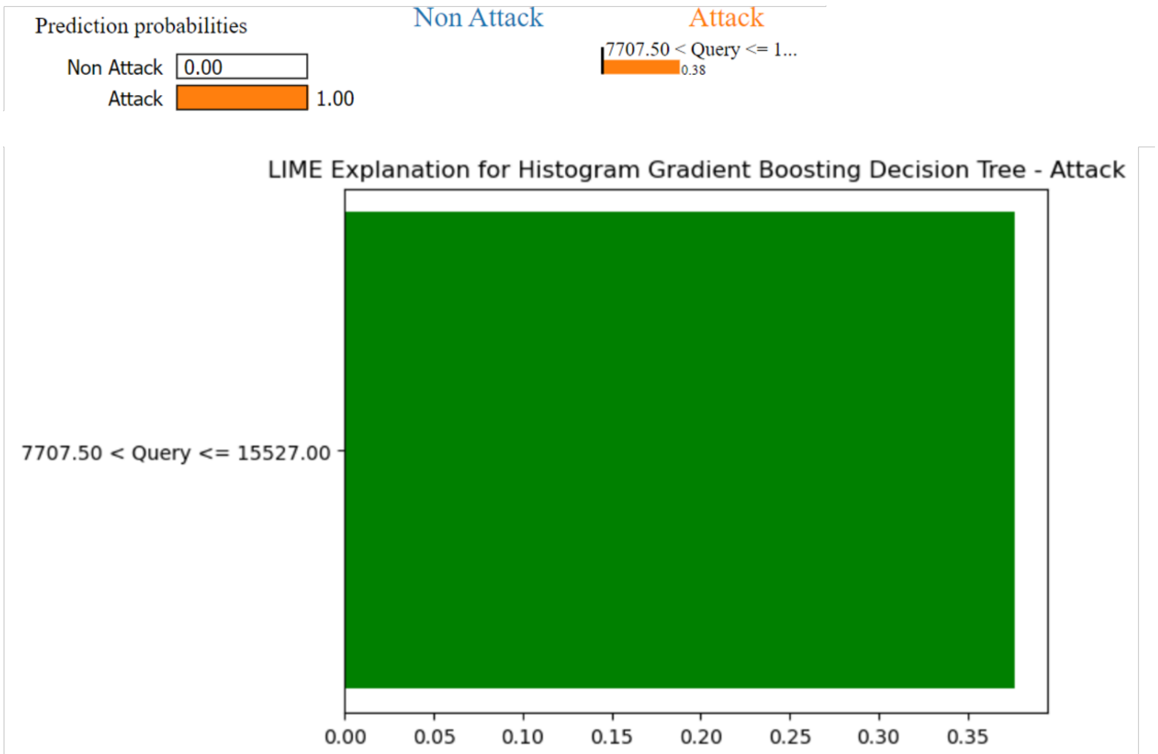


(**a**) Interpreting LIME values of GBDT for attack sample.

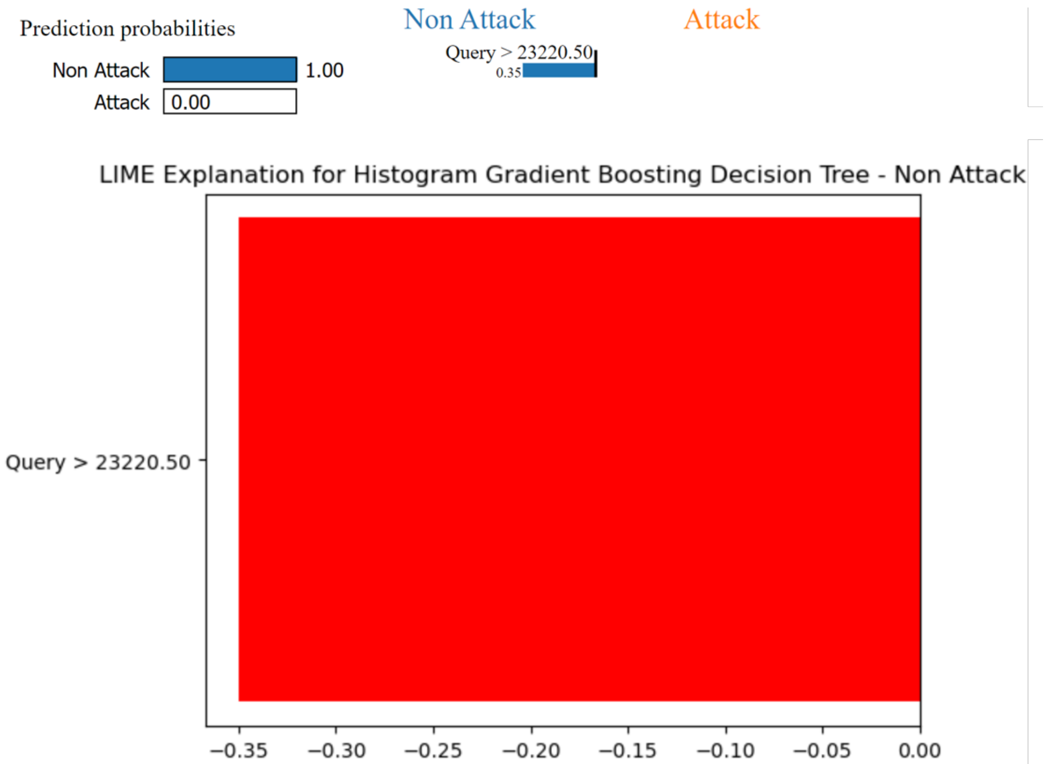


(**b**) Interpreting LIME values of GBDT for normal sample.

Figure 11. Interpreting SHAP values in GBDT model for normal and attack samples.



(**a**) Interpreting LIME values of HGBDT for attack sample.



(**b**) Interpreting LIME values of HGBDT for normal sample.

Figure 12. Interpreting SHAP values in HGBDT model for normal and attack samples.

Table 5. LIME values for classification models on sample data.

Model	Sample Data Label	LIME Range Value	Query Range Value
Decision Tree	Attack	[0–0.37]	[7707.50–15,527.00]
	Normal	[−0.30–0]	>23,220.50
Random Forest	Attack	[0–0.36]	[7707.50–15,527.00]
	Normal	[−0.35–0]	>23,220.50
XGBoost	Attack	[0–0.30]	[7707.50–15,527.00]
	Normal	[0–0.03]	>23,220.50
AdaBoost	Attack	[0–0.15]	[7707.50–15,527.00]
	Normal	[−0.12–0]	>23,220.50
GBDT	Attack	[0–0.40]	[7707.50–15,527.00]
	Normal	[−0.34–0]	>23,220.50
HGBDT	Attack	[0–0.38]	[7707.50–15,527.00]
	Normal	[−0.35–0]	>23,220.50

5. Discussion

This study's experimental findings reveal that decision tree, random forest, and AdaBoost models excel in detecting SQL injection attacks, with metrics such as 99.50% accuracy, a 99.33% F1 score, and a 99% AUC-ROC. These results emphasize the stability and robustness of these ensemble methods in SQL injection detection. On the other hand, while XGBoost is often effective in various domains, it demonstrated comparatively weaker performance in this application. This may be due to the nature of the SQL injection dataset, where simpler ensemble models like random forest and AdaBoost can better handle feature interactions without requiring extensive parameter tuning. XGBoost's complexity and sensitivity to hyperparameters, combined with the relatively straightforward structure of SQL injection data, may have contributed to its limitations in this field.

The success of tree-based and ensemble methods in this context can be attributed to several factors. Firstly, SQL injection attacks often involve specific patterns or structures in the input data. Tree-based models excel at capturing these hierarchical relationships, allowing them to distinguish between benign and malicious SQL queries effectively. Secondly, ensemble methods (random forest and AdaBoost) leverage the power of multiple models, which is particularly beneficial in cybersecurity applications where attack vectors can be diverse and evolving.

When we compare our results with the existing literature, our top-performing models achieve slightly higher accuracy than some previous studies. For instance, the study [36] reported a 97.71% F1 score using Naive Bayes, while our best models exceed a 99% F1 score. This improvement, although marginal, could translate to significant real-world benefits in reducing false positives and negatives in SQL injection detection systems.

The high performance of simpler models like decision trees is particularly noteworthy. Although complex deep learning approaches have gained popularity in various domains, our results suggest that simpler, interpretable models can equally effectively detect SQL injection. This finding has important implications for practical implementations, as these models are computationally less intensive and more accessible to deploy and maintain in real-time detection systems.

5.1. Insights from Local Explanation Results

The application of SHAP and LIME for local interpretability provides additional valuable information about the decision-making process of our models. The SHAP force plots and LIME explanations clearly identify the features that contribute the most to each model's classification of attack or normal samples.

From the SHAP visualizations, it is evident that certain features have a strong positive impact on predicting SQL injection attacks, while others consistently support normal behavior classifications. These findings highlight that specific input patterns, such as certain SQL query components, are consistently recognized as attack indicators by models like decision tree and random forest. This consistency across various samples demonstrates the reliability of our models' pattern-recognition capabilities.

LIME further reinforces these observations by showing the prediction probabilities and explaining feature contributions in a clear and interpretable manner. The alignment between SHAP and LIME results strengthens confidence in the robustness of the models and their transparency, making them more trustworthy for practical cybersecurity applications.

Both SHAP and LIME provided consistent indications of the influence of characteristics, enhancing the transparency of the model without contradictory conclusions. This complementary analysis highlights the strengths of ensemble methods in managing feature complexity and accurately detecting SQL injections.

Despite their strengths, both SHAP and LIME have limitations. SHAP can be computationally intensive, particularly with large datasets or complex models, as it requires evaluating all possible combinations of feature contributions. This can lead to longer computation times. Furthermore, while SHAP provides accurate values of the importance of features, the visual complexity of SHAP plots can sometimes overwhelm users, particularly those unfamiliar with the method. LIME has its own set of limitations, mainly related to its reliance on local approximations. Since LIME focuses on interpreting individual predictions, it may not always capture the global behavior of the model, leading to potentially misleading interpretations if the model is highly nonlinear. Furthermore, the choice of the perturbation method and the local model can significantly influence the explanations produced by LIME, which may introduce variability in the results.

5.2. Limitations and Future Work

Our experiments demonstrate strong performance in detecting SQL injection attacks; however, they were conducted in a controlled environment, primarily focused on known attack patterns. Recognizing the limitations of this approach, future work should evaluate the robustness of these models in real-world, adversarial settings where SQL injection tactics continue to evolve. To better address the sophistication of modern attacks, we plan to introduce adversarial training techniques, fine-tuning with varied attack scenarios, and model evaluation under conditions designed to simulate complex evasion attempts. Testing in such environments will clarify the models' resilience and effectiveness in dynamic, real-world applications.

- Deployment Strategies: We recommend deploying models within a Web Application Firewall (WAF) framework or as part of an intrusion detection system (IDS) that monitors SQL query traffic in real-time. This would involve integrating our models into the backend of web applications or database servers to analyze incoming queries dynamically.
- Real-Time Performance Considerations: When deploying these models, real-time performance is a critical factor. Models should be optimized for low latency, ensuring quick response times to maintain user experience. Techniques such as model pruning, quantization, or using lighter architectures may be employed to reduce inference time without significantly sacrificing accuracy. Moreover, integrating blockchain technology can enhance the security and integrity of database transactions [37]. By utilizing a blockchain framework, we can establish a robust system that not only supports the real-time detection of SQL injection attacks, but also ensures the immutability of transaction records. This dual approach allows for immediate response capabilities while maintaining a secure environment for data handling, ultimately improving the resilience of SQL injection detection systems in dynamic web applications.
- Handling Continuous Data Streams: The models need to be adaptable to accommodate continuous data streams or dynamic SQL content. This can be achieved through an

online learning approach, where the models are periodically re-trained on new data to adapt to evolving attack patterns and query structures. Additionally, implementing a feedback loop that captures false positives and negatives can enhance model accuracy by allowing for iterative improvements based on real-world usage.

- Monitoring and Maintenance: Finally, continuous monitoring of model performance in the production environment is necessary. Precision, recall, and overall accuracy should be tracked to detect any degradation over time. Regular updates and retraining based on the latest data trends will help ensure that the models remain effective against emerging SQL injection techniques.

6. Conclusions

This paper presents a comparative analysis of various decision models for detecting SQL injection attacks, focusing on decision tree, random forest, XGBoost, AdaBoost, GBDT, and HGBDT. Among these, the decision tree, random forest, and AdaBoost models exhibit superior performance, achieving high accuracy in identifying SQL injection attacks. Their robust performance, with accuracy metrics exceeding 99%, underscores their reliability and suitability for real-world deployment in enhancing cybersecurity defenses.

While gradient-boosting methods like GBDT and HGBDT demonstrate competitive results, they do not outperform simpler ensemble methods like random forest and AdaBoost. In particular, XGBoost, despite its popularity, shows limitations in this specific application, suggesting that additional refinements or alternative techniques may be necessary to optimize its performance for SQL injection detection.

Incorporating local explanation methods such as SHAP and LIME has enriched our understanding of model behavior, providing transparency to the decision-making processes of our models. These explanations reveal how certain features consistently contribute to attack and normal classifications, thereby increasing trust in the models' predictions. This interpretability is crucial in cybersecurity contexts, where understanding the reasoning behind attack detection is as important as the accuracy of the detection itself.

In future work, we propose a framework for integrating our top-performing models with real-time monitoring systems deployed in either cloud or on-premises environments to process incoming SQL queries. This integration will facilitate continuous learning through feedback loops that update models based on new attack patterns, ensuring adaptability to evolving threats. We will also establish performance monitoring metrics focusing on precision and recall to address imbalance issues. In addition, future research will explore hybrid approaches, including blockchain technology, that combine the strengths of various classifiers to improve detection rates and resilience against sophisticated techniques. As SQL injection attacks continue to evolve, adaptive and interpretable models, such as those presented in this study, will be essential to maintain robust security in database-driven applications.

Author Contributions: Conceptualization, T.-T.-H.L. and Y.H.; methodology, T.-T.-H.L.; software, T.-T.-H.L. and Y.H.; validation, T.-T.-H.L. and H.K.; formal analysis, Y.H., C.C.; investigation, Y.H. and C.C.; resources, C.C., R.W.W. and D.S.C.P.; data curation, Y.H. and C.C.; writing—original draft preparation, T.-T.-H.L., R.W.W. and D.S.C.P.; writing—review and editing, T.-T.-H.L. and H.K.; visualization, T.-T.-H.L., Y.H., C.C., R.W.W. and D.S.C.P.; supervision, T.-T.-H.L. and H.K.; project administration, H.K.; funding acquisition, H.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the Convergence security core talent training business (Pusan National University) support program (RS-2022-II221201) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation) and was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (RS-2020-II201797) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: Author Changwoo Choi was employed by the company SmartM2M. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. OWASP Top Ten. The Ten Most Critical Web Application Security Risks. Available online: https://owasp.org/www-project-top-ten/ (accessed on 2 May 2024).
2. Halfond, W.G.; Viegas, J.; Orso, A. A classification of SQL-injection attacks and countermeasures. In Proceedings of the IEEE International Symposium on Secure Software Engineering, Washington, DC, USA, 13–15 March 2006; IEEE: Piscataway, NJ, USA, 2006; Volume 1, pp. 13–15.
3. Sajjadi, S.M.S.; Pour, B.T. Study of SQL Injection attacks and countermeasures. *Int. J. Comput. Commun. Eng.* **2013**, *2*, 539. [CrossRef]
4. Su, Z.; Wassermann, G. The essence of command injection attacks in web applications. *Acm Sigplan Not.* **2006**, *41*, 372–382. [CrossRef]
5. Shar, L.K.; Tan, H.B.K. Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Inf. Softw. Technol.* **2013**, *55*, 1767–1780. [CrossRef]
6. Nair, S.S. Securing Against Advanced Cyber Threats: A Comprehensive Guide to Phishing, XSS, and SQL Injection Defense. *J. Comput. Sci. Technol. Stud.* **2024**, *6*, 76–93. [CrossRef]
7. Anley, C. *Advanced SQL Injection in SQL Server Applications*; NGS Software Insight Security Research (NISR): 2002. Available online: http://www.ngssoftware.com (accessed on 4 November 2024).
8. Chandola, V.; Banerjee, A.; Kumar, V. Anomaly detection: A survey. *Acm Comput. Surv.* **2009**, *41*, 1–58. [CrossRef]
9. Dietterich, T.G. Ensemble methods in machine learning. In *International Workshop on Multiple Classifier Systems*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 1–15.
10. Le, T.T.H.; Kim, H.; Kang, H.; Kim, H. Classification and explanation for intrusion detection system based on ensemble trees and SHAP method. *Sensors* **2022**, *22*, 1154. [CrossRef] [PubMed]
11. Breiman, L. Random forests. *Mach. Learn.* **2001**, *45*, 5–32. [CrossRef]
12. Le, T.T.H.; Oktian, Y.E.; Kim, H. XGBoost for imbalanced multiclass classification-based industrial internet of things intrusion detection systems. *Sustainability* **2022**, *14*, 8707. [CrossRef]
13. Le, T.T.H.; Wardhani, R.W.; Putranto, D.S.C.; Jo, U.; Kim, H. Toward Enhanced Attack Detection and Explanation in Intrusion Detection System-Based IoT Environment Data. *IEEE Access* **2023**, *11*, 131661–131676. [CrossRef]
14. Friedman, J.H. Greedy function approximation: A gradient boosting machine. *Ann. Stat.* **2001**, *29*, 1189–1232. [CrossRef]
15. Recio-García, J.A.; Orozco-del-Castillo, M.G.; Soladrero, J.A. Case-Based Explanation of Classification Models for the Detection of SQL Injection Attacks. In Proceedings of the ICCBR XCBR'23: Workshop on Case-based Reasoning for the Explanation of Intelligent Systems at ICCBR2023, Aberdeen, UK, 17–20 July 2023; pp. 200–215.
16. Cumi-Guzman, B.A.; Espinosa-Chim, A.D.; Orozco-del-Castillo, M.G.; Recio-García, J.A. Counterfactual Explanation of a Classification Model for Detecting SQL Injection Attacks. In Proceedings of the ICCBR XCBR'24: Workshop on Case-Based Reasoning for the Explanation of Intelligent Systems at ICCBR2024, Mérida, Mexico, 1 July 2024.
17. Le, T.T.H.; Prihatno, A.T.; Oktian, Y.E.; Kang, H.; Kim, H. Exploring local explanation of practical industrial AI applications: A systematic literature review. *Appl. Sci.* **2023**, *13*, 5809. [CrossRef]
18. Lundberg, S. A unified approach to interpreting model predictions. *arXiv* **2017**, arXiv:1705.07874.
19. Ribeiro, M.T.; Singh, S.; Guestrin, C. Why should i trust you? Explaining the predictions of any classifier. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 1135–1144.
20. Demilie, W.B.; Deriba, F.G. Detection and prevention of SQLI attacks and developing compressive framework using machine learning and hybrid techniques. *J. Big Data* **2022**, *9*, 124. [CrossRef]
21. Valeur, F.; Mutz, D.; Vigna, G. A Learning-Based Approach to the Detection of SQL Attacks. In Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment, Vienna, Austria, 7–8 July 2005.
22. Gao, Q.; Li, H.; Huang, T. Detecting SQL Injection Attack Using an Artificial Neural Network. In Proceedings of the International Conference on Computational Intelligence and Security, Suzhou, China, 13–17 December 2008.
23. Xu, D.; Huang, H.; Yin, H. An Improved SQL Injection Detection Model Based on Machine Learning. *J. Softw. Eng. Appl.* **2010**, *3*, 1131–1135.
24. Pan, X.; Liu, L.; Yan, H. SQL Injection Detection Based on AdaBoost Algorithm. In Proceedings of the IEEE International Conference on Computer and Information Technology, Nadi, Fiji, 8–10 December 2016.
25. Le, T.Q.; Tran, D.H.; Nguyen, H.T. SQL Injection Detection Using Gradient Boosting Decision Trees. In Proceedings of the International Conference on Information and Communication Technology, Bandung, Indonesia, 3–4 May 2018.

26. Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016.
27. Ke, G.; Meng, Q.; Finley, T.; Wang, T.; Chen, W.; Ma, W.; Liu, T.Y. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017.
28. Nguyen, D.M.; Tran, M.T.; Pham, B.T. A Comparative Study of Machine Learning Algorithms for SQL Injection Detection. *J. Inf. Secur. Appl.* **2019**, *44*, 144–154.
29. Hwang, Y.; Le, T.-T.-H.; Wardhani, R.W.; Puranto, D.S.C.; Kim, H. Enhancing SQL Injection Detection Using Ensemble Learning and Boosting Models. In Proceedings of the 2024 International Conference on Platform Technology and Service (PlatCon-24), Jeju, Republic of Korea, 26–28 August 2024.
30. Chawla, N.V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Intell. Res.* **2002**, *16*, 321–357. [CrossRef]
31. Deriba, F.G.; Salau, A.O.; Mohammed, S.H.; Kassa, T.M.; Demilie, W.B. Development of a compressive framework using machine learning approaches for SQL injection attacks. *Prz. Elektrotechniczny* **2022**, *98*, 181–187. [CrossRef]
32. SQL Injection Dataset. Available online: https://www.kaggle.com/datasets/sajid576/sql-injection-dataset (accessed on 25 May 2024).
33. Hosam, E.; Hosny, H.; Ashraf, W.; Kaseb, A.S. Sql injection detection using machine learning techniques. In Proceedings of the 2021 8th International Conference on Soft Computing & Machine Intelligence (ISCMI), Cario, Egypt, 26–27 November 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 15–20.
34. Gowtham, M.; Pramod, H.B. Semantic query-featured ensemble learning model for SQL-injection attack detection in IoT-ecosystems. *IEEE Trans. Reliab.* **2021**, *71*, 1057–1074.
35. Abdulhamza, F.R.; Al-Janabi, R.J.S. SQL injection detection using 2D-convolutional neural networks (2D-CNN). In Proceedings of the 2022 International Conference on Data Science and Intelligent Computing (ICDSIC), Karbala, Iraq, 1–2 November 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 212–217.
36. Roy, P.; Kumar, R.; Rani, P. SQL injection attack detection by machine learning classifier. In Proceedings of the 2022 International Conference on Applied Artificial Intelligence and Computing (ICAAIC), Salem, India, 9–11 May 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 394–400.
37. Bhuva, D.; Kumar, S. Securing Space Cognitive Communication with Blockchain. In Proceedings of the 2023 IEEE Cognitive Communications for Aerospace Applications Workshop (CCAAW), Cleveland, OH, USA, 20–22 June 2023; pp. 1–6. [CrossRef]