applied sciences

MDPI

Article

SQL Injection Detection Based on Lightweight Multi-Head Self-Attention

Rui-Teng Lo [1], Wen-Jyi Hwang [1,*] and Tsung-Ming Tai [2]

[1] Department of Computer Science and Information Engineering, National Taiwan Normal University, Taipei 116, Taiwan; 61147018s@ntnu.edu.tw
[2] NVIDIA AI Technology Center, NVIDIA Taiwan, Taipei 114, Taiwan; ntai@nvidia.com
* Correspondence: whwang@ntnu.edu.tw

Abstract: This paper presents a novel neural network model for the detection of Structured Query Language (SQL) injection attacks for web applications. The model features high detection accuracy, fast inference speed, and low weight size. The model is based on a novel Natural Language Processing (NLP) technique, where a tokenizer for converting SQL queries into tokens is adopted as a pre-processing stage for detection. Only SQL keywords and symbols are considered as tokens for removing noisy information from input queries. Moreover, semantic labels are assigned to tokens for highlighting malicious intentions. For the exploration of correlation among the tokens, a lightweight multi-head self-attention scheme with a position encoder is employed. Experimental results show that the proposed algorithm has high detection performance for SQL injection. In addition, compared to its lightweight NLP counterparts based on self-attention, the proposed algorithm has the lowest weight size and highest inference speed. It consumes only limited computation and storage overhead for web services. In addition, it can even be deployed in the edge devices with low computation capacity for online detection. The proposed algorithm therefore is an effective low-cost solution for SQL injection detection.

Keywords: cyber security; SQL injection detection; natural language processing; machine learning; deep learning

Academic Editors: Arcangelo Castiglione and Arkadiusz Biernacki

Received: 14 November 2024
Revised: 29 December 2024
Accepted: 6 January 2025
Published: 9 January 2025

Citation: Lo, R.-T.; Hwang, W.-J.; Tai, T.-M. SQL Injection Detection Based on Lightweight Multi-Head Self-Attention. *Appl. Sci.* **2025**, *15*, 571. https://doi.org/10.3390/app15020571

1. Introduction

Structured Query Language (SQL) is a programming language used for processing database. The SQL is a fundamental tool for web development for effective data management, authentication, and dynamic content generation. SQL injection is a type of attack that exploits a vulnerability in SQL statements that interacts with web databases. An attacker can inject malicious SQL commands into the user input, such as a URL parameter, and execute them on the database server. This could lead to unauthorized access to sensitive data and/or the corruption of database integrity. The Open WEB Application Security Project (OWASP) has listed SQL Injection as one of the top threats on web-based applications [1].

A number of studies have been undertaken to address the SQL injection threat. Traditional rule-based SQL injection detection methods [2–4], while useful for identifying known patterns of malicious activity, have limitations. These methods rely heavily on predefined rules, which can quickly become outdated as new techniques are being developed by attackers. This leads to a constant need for updates to ensure effectiveness. Moreover, the rule-based systems are often unable to detect sophisticated attacks that do not match the existing rule sets.

Alternatives to rule-based techniques are the methods of Machine Learning (ML) [5,6]. These techniques involve training models on datasets containing normal and malicious SQL queries to distinguish between legitimate traffic and potential threats. Algorithms such as decision trees [7,8], random forests [8,9], Support Vector Machine (SVM) [7,8,10,11], and artificial neural networks [8,12,13] are commonly employed due to their effectiveness in pattern recognition and anomaly detection. By continuously learning and adapting to new data, ML models can provide a robust defense against SQL injection.

Among ML techniques, the Bidirectional Encoder Representations from Transformers (BERT) [14] language model has been found to be superior for SQL injection [15,16]. BERT is a Natural Language Processing (NLP) model that stands out due to its unique approach to context. Unlike traditional models such as Recurrent Neural Networks (RNNs) [6] that process text in one direction, BERT analyzes words in relation to all other words in a sentence both before and after, providing a more comprehensive understanding. This scheme, termed self-attention [17,18], produces the bidirectional context crucial for understanding the intent behind SQL queries. This is beneficial for the detection of malicious attacks.

Although BERT is promising for SQL injection detection, there are still a number of limitations. Firstly, BERT's model size is substantial, making it challenging to deploy on web systems with limited memory and processing power. In addition, the computational requirements for running BERT inference tasks are intensive, which can lead to latency issues. These limitations necessitate the development of more efficient NLP models that maintain high accuracy while being suitable for SQL detection deployment.

The goal of this study is to present a novel lightweight NLP model for SQL injection detection. The model features high detection performance, low computation complexities, and low deployment costs. The model is based on a self-attention scheme for high accuracy injection detection. To facilitate the deployment of the proposed model and reduction of inference time, the weight size of the proposed model is significantly lower than that of the BERT model. This is accomplished by the adoption of a tokenizer specific to SQL languages, where only the keywords of the languages are regarded as the tokens. Furthermore, the tokens are associated with semantic labels so that malicious intentions from attackers could be highlighted. A lightweight multi-head attention scheme is then employed with a simple sinusoidal position encoder for the exploration of correlation among the tokens. In this way, self-attention operations are effectively carried out with low hardware costs. Therefore, high accuracy for SQL injection can be achieved with the proposed lightweight model.

The remaining parts of the paper are organized as follows. The related works to this study are included in Section 2. Section 3 presents the proposed NLP model for SQL injection detection in detail. The experimental results and evaluations of the proposed model are then revealed in Section 4. Finally, concluding remarks of this work are provided in Section 5.

2. Related Works

NLP offers significant advantages in detecting SQL injection attacks. NLP techniques can understand and interpret the intent behind text, which is crucial in distinguishing between benign SQL queries and malicious injections. In studies [11,12], SQL-specific tokenizers were used for parsing SQL queries. In addition, word-to-vector models were employed for subsequent feature extraction [11,13]. Term Frequency–Inverse Document Frequency (TF-IDF) may also be beneficial for feature extraction [19,20]. The SVM [11,19], Convolutional Neural Network (CNN) [12,13] and bidirectional Long Short-Term Memory (bi-LSTM) [20] have then been adopted as classifiers for malicious injections.

With the advent of NLP models based on self-attention [17,18], a number of studies have been made to adopt the models for SQL injection detection [15,16,21,22]. The self-

attention allows models to capture the context and semantics of SQL queries more effectively than traditional methods. This leads to improved accuracy in distinguishing between benign and malicious queries, reducing both false positives and false negatives. To perform the self attention operations, tokens are first derived from an SQL query. Examples of tokens could be words, subwords, or characters. Each token in the SQL query is then transformed into a query, key, and value vectors through learned linear transformations. Attention scores are subsequently computed from the query, key, and value vectors to capture the relationship among tokens from the SQL query.

For the studies in [21,22], multi-head self-attention layers were employed so that multiple attention mechanisms could be operated in parallel with different sets of learned parameters for exploring different aspects of correlations among tokens. In this way, more diverse representations from malicious injection attacks could be learned. Furthermore, in the BERT model, multi-head self-attention layers could be cascaded with feedforward neural networks for building Transformer encoder layers for further enhancing the detection performance. Promising results have been observed from BERT-based SQL injection detection techniques [15,16,23]. However, high detection accuracy was achieved at the expense of considerable computation and memory loads.

To address this issue, a number of lightweight transformer models have been proposed. An example is the A Little BERT (ALBERT) [24] model, where the weight size was lowered by factorizing the embedding matrix into two smaller matrices. Furthermore, weights across different layers were shared. Another approach, termed distilled BERT (distilBERT) [25,26], reduces its weight size using knowledge distillation. In distilBERT, a teacher–student training process is employed, where a BERT model acts as the larger, pre-trained teacher model. The teacher model transfers knowledge to the distilBERT, which contains smaller number of weights. Efficiently Learning an Encoder that Classifies Token Replacements Accurately (ELECTRA) [27] has also been found to be effective for the reduction of weight size. It is accomplished by designing a more efficient training objective, termed Replaced Token Detection (RTD) objective, which enables the better utilization of a smaller model for maintaining competitive performance.

To further accelerate the detection speed, cascaded approaches combining lightweight transformers with classical machine learning models have been proposed [28]. In the approaches, classical machine learning models such as SVM are adopted as the detector at the first stage for the fast detection of normal queries. The lightweight model at the second stage then re-examine the queries found to be malicious at the first stage. Although the cascaded approach can further reduce the computation costs [28], the employment of transformer models is still necessary. The deployment difficulty may still be high for edge devices with limited hardware resources.

Compared to the existing lightweight NLP models stated above, the proposed model has the following advantages. First of all, its weight size is significantly less than that of the existing models. This would facilitate the deployment of the model in the edge devices with limited storage capacity. Furthermore, because only a smaller number of weights are needed for inference operations, the proposed algorithm has faster speed for SQL injection detection. Finally, similar to the existing techniques, the proposed model is based on multi-head self-attention operations for exploring the correlation among different tokens in the input SQL queries for the anomaly detection. The proposed model attained detection accuracy comparable to the existing techniques.

3. The Proposed Approach

In this section, the proposed approach for the detection of SQL injection is presented in detail. We first describe the web system with the proposed neural network model for

SQL injection detection. The overview of the proposed model is then provided. The model can be separated into two parts: input embedding and detection. There are separate subsections for the descriptions of these two parts. Finally, the training of the proposed model is considered. A list of frequently used symbols is summarized in Appendix A to facilitate the discussions of the proposed model.

3.1. Web System with SQL Injection Detection

For web applications, the SQL is a powerful language for managing and manipulating database. SQL codes are commonly used for remotely add, delete, modify, and query data in a database in web servers. Nevertheless, web applications are also vulnerable to SQL injection attacks, which occur when an attacker sends malicious SQL codes to the databases of the applications through web forms or URL parameters. Some impacts of injection attacks include authentication bypass, compromised data integrity, compromised availability of data, and information disclosure. Figure 1 shows a simple example of SQL injection attacks, where attackers use tautology SQL injection to bypass the authentication. In this example, the administration account is hacked, and the entire database can be retrieved without a password. As shown in Figure 1, the administrator account can be accessed without the correct password (e.g., **root2024**). This is accomplished by injecting a malicious statement (e.g., **OR '1' = '1'- - '**) after the statement specifying the administration account name (e.g., **username = 'administor'**) in the SQL query from the hacker. The command for password check (e.g., **AND password= 'dummy123';**) would then be ignored for authentication. Consequently, even though the password in the hacker's SQL is not correct, the database can still be accessed.

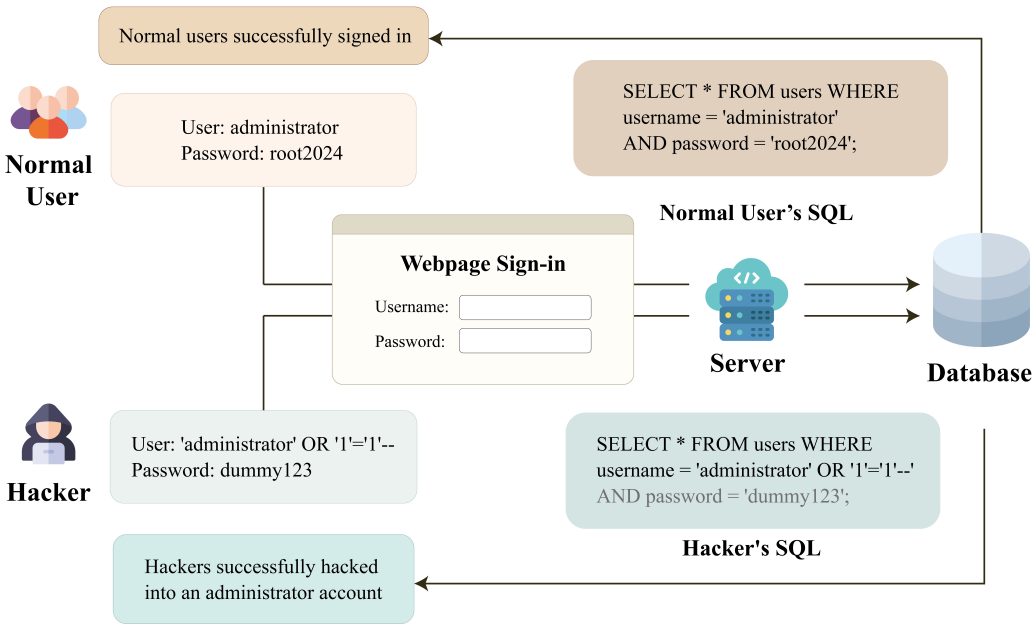


Figure 1. A simple example of SQL injection. The authentication for database access is bypassed by tautology SQL injection in this example. The hacker is therefore able to retrieve the entire database without the correct password to the administration account.

One way to solve this vulnerability issue is by the employment of a detection system. The system is able to detect and drop malicious SQL statements before they reach the database. A common approach for implementing a detection system is by the accommodation of a large neural network model in a cloud. However, when the system is deployed in a remote cloud, delivery of the SQL codes to the cloud for detection may incur long latency. Moreover, even for the local deployment of the detection system, a large neural

network model may impose high computation and storage burdens for the local servers. It is therefore desirable to employ a lightweight neural network model with high accuracy for SQL injection detection.

Figure 2 shows an example for the deployment of the proposed model in the local server. In the example, the proposed model and its corresponding web server and database are accommodated locally in an edge device, which is a low-cost Personal Computer (PC) with only limited computation and/or storage resources. Because the network size of the proposed model is small, computation overhead for the inference operations would be low even for the low-cost PC. Furthermore, no delivery latency is incurred because the model and its protection targets are in the same edge device. Fast and accurate detection can then be achieved without the utilization of cloud services.

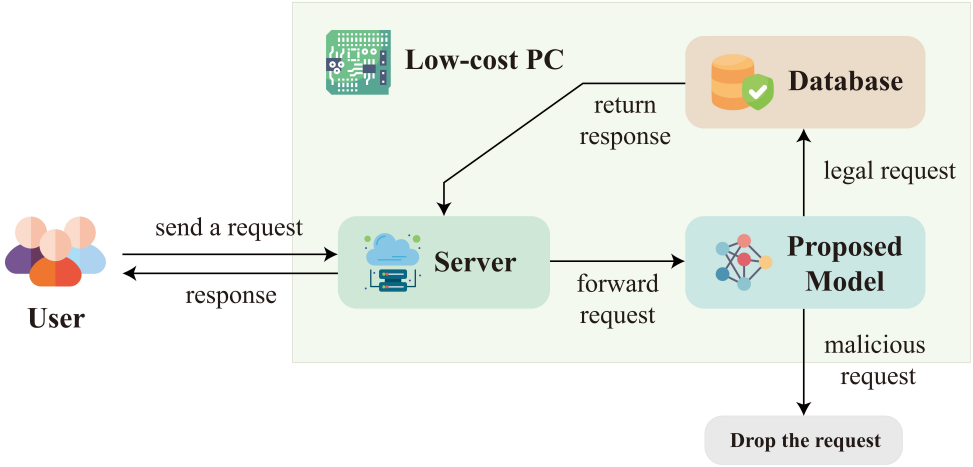


Figure 2. The deployment of the proposed lightweight model in a low cost computer containing the web server for SQL injection detection. The proposed model is able to perform accurate detection with low latency even when the computer is limited in computation and storage capacity.

3.2. Overview of the Proposed Model

The proposed lightweight neural network model is able to accurately identify malicious SQL codes without imposing heavy computation and storage overhead. As shown in Figure 3, the proposed model can be separated into two major portions: input embedding and detection. The embedding operations can be viewed as the pre-processing step of the model. Its goal is to map the SQL statements into numerical matrices. These matrices capture semantic meaning and context, allowing the proposed model to process the SQL statements.

Based on the matrices delivered by input embedding operations, the detection model then produces the final results. The goal of detection model is to extract both local and global features from the input numerical matrices so that detection operations can be accurately carried out. Both CNN and multi-head attention operations are employed in the model. Detailed descriptions of the neural networks for input embedding and detection operations are revealed in the following two subsections.

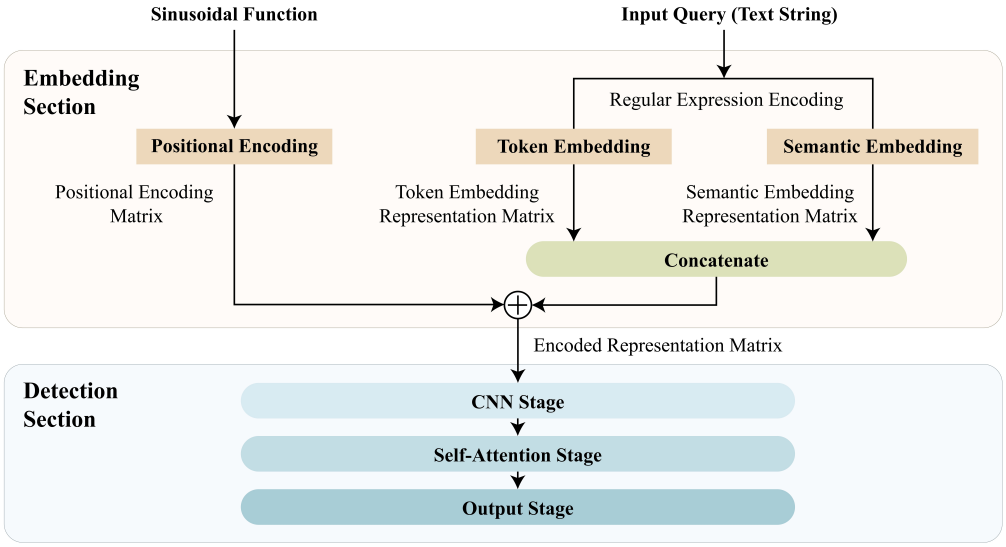


Figure 3. The overview of the proposed model. It contains two sections: input embedded section and detection section. For an input SQL query, the input embedding section provides a feature matrix, termed encoded representation matrix. An estimated probability for malicious attack is then produced by the detection section. A malicious query is detected when the estimated probability is above a pre-specified threshold.

3.3. Input Embedding

To perform the input embedding operations, a SQL-specific tokenizer is adopted. The tokenizer converts words and symbols in the SQL codes into numeric tokens. The user-defined identifiers in the codes are not considered for the conversion. It has been found [12,29] that the removal of user-defined identifiers or attributes is beneficial for the avoidance of noisy information in the SQL queries. The removal operations can be easily implemented by first defining a set of vocabularies $\mathcal{V}$, which consists of words and symbols from SQL languages. In the tokenizer, words or symbols from the input queries not belonging to $\mathcal{V}$ are removed. Otherwise, they are converted into numerical tokens.

To facilitate the training operations for neural network models, a semantic label could also be assigned to each numerical token [12] in the queries. In this study, there are three different classes of semantic labels: command, symbol, and expression. An example of the results produced by the SQL-specific tokenizer is shown in Figure 4, where the SQL query after removing user-defined identifiers or attributes is termed the skeleton query. Each element in the skeleton query is a token. As shown in the figure, each token is associated with a token index and a semantic label. Therefore, a token list and a semantic list can be formed after the skeleton query is acquired from the input query.

Let N be the number of tokens accepted by the proposed algorithm from an input SQL query. It is fixed so that zero padding is necessary when the actual number of tokens is less than N. Based on the example of the lists produced by the SQL-specific tokenizer in Figure 4a, the results of the proposed embedding scheme are revealed in Figure 4b. In the example, it can be observed from Figure 4b that each token and its associated semantic label from the lists are mapped to two numerical vectors by functions f_1 and f_2, respectively. Both f_1 and f_2 are fully connected neural networks with weights determined by training. Let M_1 and M_2 be the dimensions of vectors produced by f_1 and f_2, respectively. The numerical vectors for all the tokens and semantic labels are then concatenated to form the embedding matrix for tokens and semantic labels. The resulting matrix, denoted by $\mathbf{F}$, is termed combined representation matrix in this study. As shown in Figure 4, it has dimension $N \times M$, where $M = M_1 + M_2$.

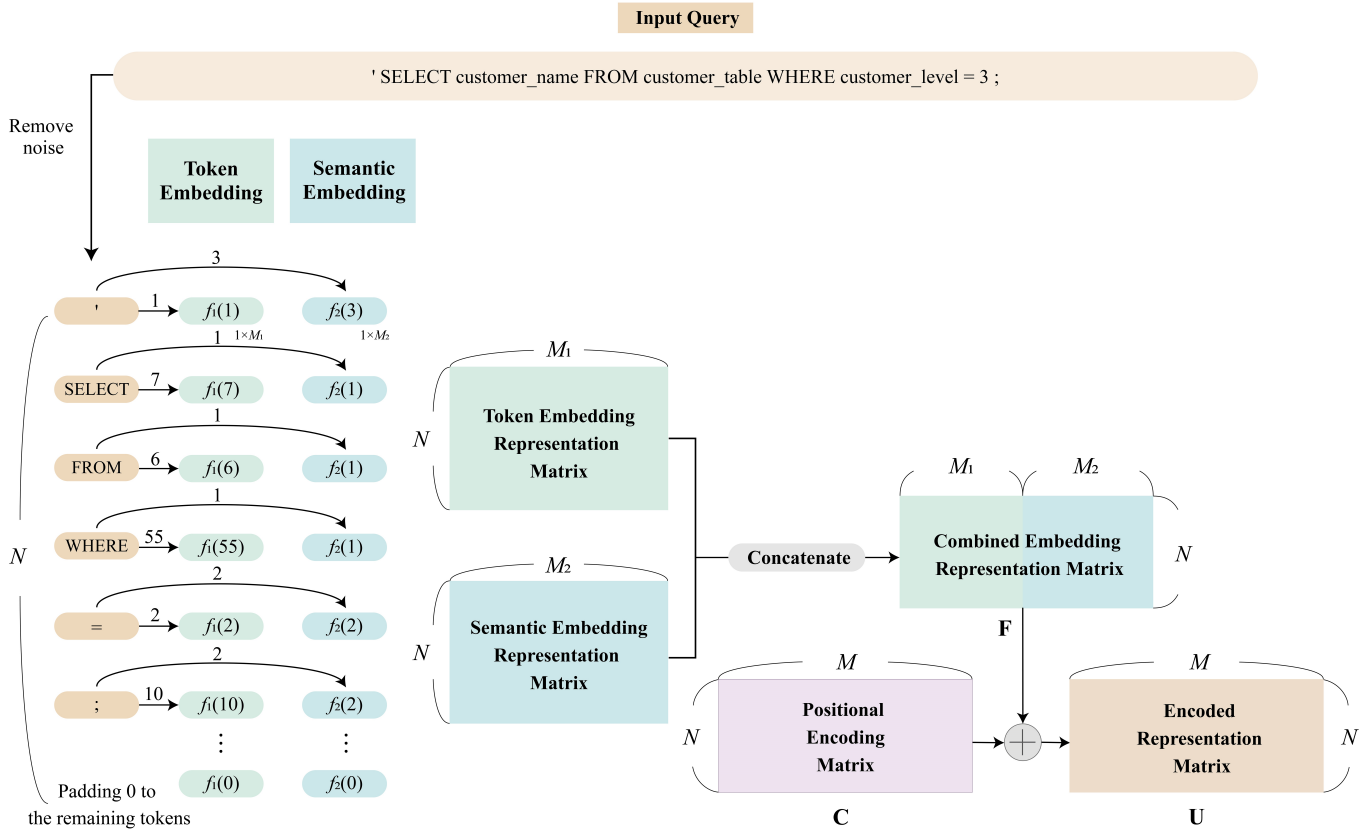
Input Query: ' SELECT customer_name FROM customer_table WHERE customer_level = 3 ;

Skeleton Query: ' SELECT FROM WHERE = ;

Token List: [1, 7, 6, 55, 2, 10]

Semantic List: [3, 1, 1, 1, 2, 2]

(**a**)



(**b**)

Figure 4. An example of the results produced by the proposed embedding scheme. (**a**) Based on an input query, a skeleton query is first formed by removing user-defined identifiers and attributes. Lists of N tokens and their corresponding semantic labels are then created from the skeleton query. (**b**) From the lists of tokens and labels, the combined embedding representation matrix **F** is then produced, which contains numerical vectors representing the tokens and the semantic labels. The position information of tokens, denoted by **C**, is then added to **F**. The results are termed the encoded representation matrix, denoted by **U**.

To facilitate the subsequent self-attention operations, the information about position of the tokens in the input SQL query is also provided. In this study, the position information is included in a position encoding matrix **C** with dimension $N \times M$. The (i, j)-th element, $i = 1, ..., N, j = 1, ..., M$, of the matrix **C**, denoted by $C(i, j)$, contains the position information for the (i, j)-th element of the matrix **F**. A sinusoidal encoding approach [17] is adopted for computing $C(i, j)$, as shown below.

$$C(i,j) = \begin{cases} \sin(i/10000^{j/N}) & \text{when } j \text{ is even,} \\ \cos(i/10000^{(j-1)/N}) & \text{otherwise.} \end{cases} \quad (1)$$

Let **U** be the matrix produced by the input embedding section. The matrix, termed encoded representation matrix, is given by

$$\mathbf{U} = \mathbf{F} + \mathbf{C}. \tag{2}$$

The dimension of the encoded representation matrix **U** is therefore also $N \times M$. That is, in the matrix **U**, token content and sematic labels are combined with the position encoding information. The semantic meaning and context for SQL queries can then be effectively utilized for the subsequent detection operations.

3.4. Detection

The detection operations can be separated into three stages: the CNN, self-attention, and output stages. As shown in Figure 5, the goal of the CNN stage is to extract local features from the input matrix **U**. There are two CNN layers with maximum pooling operations in the stage. Let **X** be the output of the CNN stage. Based on **X**, the global features are highlighted in the self-attention stage. In this study, the scaled dot-product attention algorithm [17] has been adopted. Its simplest form can be characterized by three weight matrices $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ for the computation of query **Q**, key **K**, and value **V**, which are given by

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V. \tag{3}$$

The corresponding self-attention operations, denoted by S, for a set of fixed-weight matrices $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ are then given by

$$S(\mathbf{X}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}})\mathbf{V}, \tag{4}$$

where d is a scaling factor. This attention scheme is termed single-head attention.

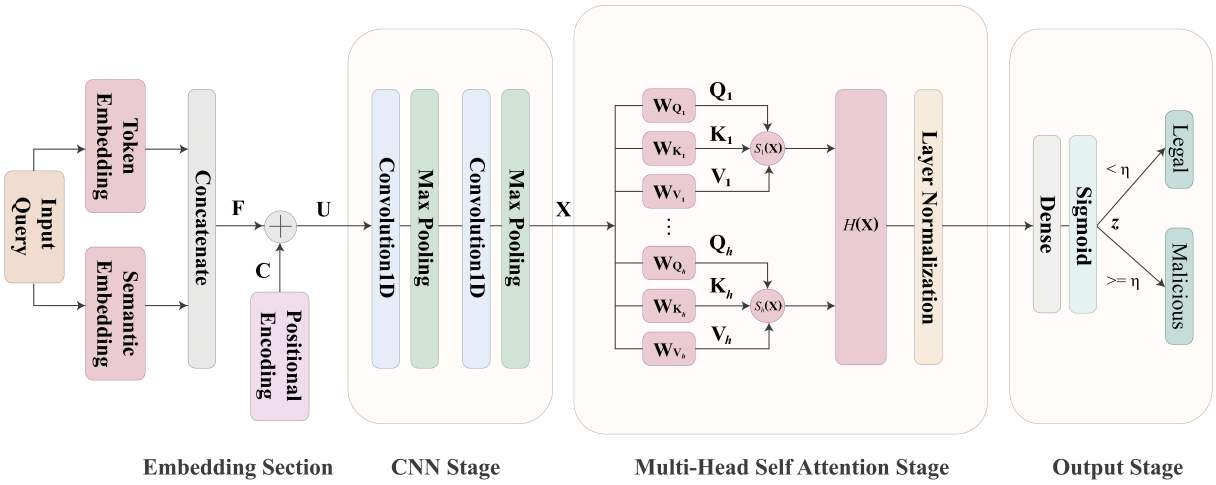


Figure 5. Proposed model for detection operations, which consists of three stages: CNN, self-attention, and output stages. The detection section takes the encoded representation matrix **U** as the input. It then produces z, the estimated probability of malicious attack by the input query. When $z \geq \eta$, the query is malicious.

To further learn more diversity for detection, we applied a multi-head self-attention in the proposed model. Let h be the number of heads in the model. Furthermore, let $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$ be the query, key, and value associated with the i-th head, $i = 1, ..., h$. They are

associated with weight matrices $\mathbf{W}_{Q_i}$, $\mathbf{W}_{K_i}$, and $\mathbf{W}_{V_i}$. Analogous to the computation of $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ by (3), $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$ are acquired by

$$\mathbf{Q}_i = \mathbf{X}\mathbf{W}_{Q_i}, \ \ \mathbf{K}_i = \mathbf{X}\mathbf{W}_{K_i}, \ \ \mathbf{V}_i = \mathbf{X}\mathbf{W}_{V_i}. \tag{5}$$

Let $S_i(\mathbf{X})$ be the self-attention operation $S(\mathbf{X})$ in (3) for $\mathbf{Q}_i$, $\mathbf{K}_i$ and $\mathbf{V}_i$. The multi-head self-attention, denoted by $H(\mathbf{X})$, is computed by

$$H(\mathbf{X}) = [S_1(\mathbf{X}) \otimes \cdots \otimes S_h(\mathbf{X})]\mathbf{W}_M, \tag{6}$$

where $\otimes$ is a concatenation operation, and $\mathbf{W}_M$ is the weight matrix for combining the results of different single-head self-attentions. Note that each head is a unique linear transformation of $\mathbf{X}$ as a query, a key, and a value. Therefore, the concatenation of individual heads allows for a combination of different linear transformation results. This could provide a diverse representation of global information for $\mathbf{X}$.

The third stage at the detection section is the output stage. It produces the final detection result z from the self-attention outcome, where $0 \leq z \leq 1$. The output stage is a fully connected layer with sigmoid activation. The detection result z indicates the possibility that the input query is malicious. The result z is then compared with a pre-specified threshold value η. A malicious query is detected when $z > \eta$.

3.5. Training

The training of the proposed model involves both the training of the input embedding and detection parts of the model. Let $s_j, j = 1, .., L$, be the j-th SQL query in the training set $\mathcal{T}$, where L is the number of queries.

Each query is associated with a label indicating whether the query is legal or malicious. The training process can then be regarded as the one for a two-class classification problem. The Binary Cross-Entropy (BCE) function was adopted as the loss function for training. Let B_j be the BCE associated with s_j, which is given by

$$B_j = -y_j \log p_j - (1 - y_j) \log(1 - p_j), \tag{7}$$

where y_j indicates whether the estimated class matches the ground truth. We set $y_j = 1$ when the match occurred. Otherwise, we set $y_j = 0$. The p_j is the estimated probability of the ground truth class. It can be derived from the detection result z of the proposed model. That is,

$$p_j = \begin{cases} z & \text{when ground truth is malicious,} \\ 1 - z & \text{otherwise.} \end{cases} \tag{8}$$

The loss function for the training, denoted by B, is given by

$$B = \frac{1}{L} \sum_{j=1}^{L} B_j. \tag{9}$$

The end-to-end training operations are then carried out. That is, the weights in fully connected neural networks f_1 and f_2 of the input embedding section and the weights in the neural networks at the CNN stage, self-attention stage, and output stage in detection section are updated iteratively for the minimization of the loss function in (9) over training set $\mathcal{T}$. We stop the iterations after the convergence of loss function B is observed.

4. Experimental Results

The evaluations of the proposed model for SQL injection detection are provided in this section. The setup of the experiments is presented first. We then define the performance metrics for the model. Based on the metrices, the numerical evaluations and comparisons among the proposed model and other existing techniques are included.

4.1. Experimental Setup

The databases considered in the experiments are the MySQL, PostgreSQL, Oracle, and Microsoft SQL servers. The training set $\mathcal{T}$ and two test sets $\mathcal{S}$ and $\mathcal{U}$ of the experiments can be found in the Kaggle platform [30]. There are 98,275 SQL queries (i.e., $L = 98,275$) in the training set $\mathcal{T}$. Among the queries for training, there are 55,915 malicious queries and 42,360 legal queries. The test set $\mathcal{S}$ contains 24,707 queries. The number of malicious and legal queries in the test set are 11,573 and 13,134, respectively. The test set $\mathcal{U}$ consists of queries with length exceeding 1000 characters. There are 500 queries in the test set $\mathcal{U}$, where 100 queries are legal, and the others are malicious. The queries in the test set $\mathcal{S}$ and $\mathcal{U}$ are different from the ones in the training set $\mathcal{T}$.

Note that the length of queries in sets $\mathcal{T}$ and $\mathcal{S}$ are below 1000 characters. However, for the NLP techniques based on self-attentions, it may be necessary to find relationship among tokens far apart in an input query. Therefore, the inclusion of the test set $\mathcal{U}$ containing lengthy queries is beneficial for the observation of the effectiveness of self attention operations. In this way, the robustness of neural network models to the attacks with lengthy queries can be evaluated. A summary of the datasets considered in the experiments can be found in Table 1.

Table 1. Number of queries in training and test sets for the experiments.

Dataset	Malicious Queries	Legal Queries	Total
Training Set $\mathcal{T}$	55,915	42,360	98,275
Test Set $\mathcal{S}$	11,573	13,134	24,707
Test Set $\mathcal{U}$ [1]	400	100	500

[1] Length of each query in the dataset exceeds 1000 characters.

For the proposed model, the vocabulary size (i.e., the size of the set $\mathcal{V}$ of vocabularies) for the SQL-specific tokenizer is 158. There are three semantic labels for the vocabularies: command, expression, and symbol. The number of vocabularies having semantic labels as command, expression, and symbol are 139, 14, and 5, respectively. The maximum number of tokens from an input query was set to be $N = 512$. The dimensions of numerical vectors produced by mappings f_1 and f_2 are $M_1 = 10$ and $M_2 = 1$, respectively. Therefore, $M = M_1 + M_2 = 11$. The dimension of the encoded representation matrix $\mathbf{U}$ in (2) is then $N \times M = 512 \times 11$. The number of heads for the self attention operations is $h = 4$. The threshold value η was set to be $\eta = 0.5$ for determining whether the query is malicious. All the training operations were carried out by an NVIDIA GPU Geforce RTX 3080 Ti with a memory size of 64 GB. Furthermore, the software platform used for the training operations is TensorFlow 2.15.

4.2. Performance Metrics

The network size, computation complexity of the proposed model, and the detection accuracy were considered as the performance metrics in this study. The network size is defined as the number of weights in the network model. It reveals the memory size required for the deployment of the network. We define the computation complexity of a

network model as the inference time of the model, which indicates the promptness of the model for the detection of malicious attacks.

For the detection accuracy, four measurements were considered: the precision rate, recall rate, accuracy rate, and F1 score. Let TP (True Positive) and FN (False Negative) be the number of malicious queries in the test set that are detected and missed, respectively. Let FP (False Positive) be the number of legal queries that are falsely detected as malicious queries. Conversely, the TN (True Negative) is the number of legal queries that have successfully passed the detection without triggering alarms. Based on the TP, FN, FP, and TN, the precision rate, recall rate, accuracy rate, and F1 score are then defined as

$$\text{Precision Rate} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall Rate} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{10}$$

$$\text{Accuracy Rate} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}, \tag{11}$$

and

$$\text{F1 Score} = \frac{2 \times \text{Precision Rate} \times \text{Recall Rate}}{\text{Precision Rate} + \text{Recall Rate}}, \tag{12}$$

respectively.

4.3. Numerical Evaluations

Table 2 shows the weight size of the proposed model. We broke down the total weight size of the model into the ones for embedding section and detection section, respectively. Furthermore, the weight size for detection section was divided into the ones for the CNN, self-attention, and output stages, respectively. It can be observed from Table 2 that the weight size of the proposed model came out to only 69,269. Therefore, the proposed model can be easily deployed in the devices with only limited storage size. We can also see from Table 2 that there are only 1404 weights in the embedding section in the proposed model. This is because the proposed tokenizer is SQL-specific. That is, only pre-defined keywords and symbols in SQL codes are considered as vocabularies. In fact, there are only 158 vocabularies in the tokenizer. Simple networks may then suffice for the embedding operations.

Table 2. The weight sizes of the proposed model with different tokenizers.

Model	Embedding Section	Detection Section			Total
		CNN	Self Attention	Output	
Proposed Model (SQL-Specific Tokenizer+ Proposed Detection Model)	1404	7680	51,992	8193	69,269
SentencePiece Tokenizer [31]+ Proposed Detection Model	600,000	8544	132,800	8193	749,537

Although generic tokenizers can be adopted in the proposed model, the resulting network size may be significantly increased. To show this fact, we included the weight size of the proposed model combined with generic SentencePiece tokenizer in Table 2. To facilitate the deployment of SentencePiece, the tokenizer used by ALBERT was adopted, which contains 30,000 vocabularies. The weight size of the tokenizer is 60,000. To accommodate the SentencePiece tokenizer [31], the weight size of the CNN and self attention stages were also enlarged. The total weight size came out to 749,537. The proposed model with SQL-specific tokenizer requires only 9.24% of the weight size of the proposed model with SentencePiece tokenizer (i.e., 69,269 vs. 749,537).

In addition to weight sizes, the precision rate, recall rate, F1 score, and accuracy of the proposed model were also considered, as shown in Table 3. To achieve meaningful comparisons, all the models in Table 3 were trained by the same training set $\mathcal{T}$ and were evaluated by the same test set $\mathcal{S}$. With a significantly lower weight size, we can see from Table 3 that the proposed model with SQL-specific tokenizer yielded accuracy, precision rate, recall rate, and F1 score values comparable to that of the proposed model with the SentencePiece tokenizer. These results justify the employment of the SQL-specific tokenizer for the malicious detection.

Table 3. Performance of the proposed model with different tokenizers for the test set $\mathcal{S}$. There are 24,707 queries in the test set. The length of each query is below 1000 characters.

Model	Accuracy	Precision	Recall	F1 Score	Weight Size
Proposed Model (SQL-specific Tokenizer+ Proposed Detection Model)	98.98%	98.01%	99.84%	98.92%	69,269
SentencePiece Tokenizer [31]+ Proposed Detection Model	99.15%	98.36%	99.86%	99.10%	749,537

To further demonstrate the effectiveness of the proposed model, we compared the precision rate, recall rate, accuracy, F1 score, and weight size of the proposed model with existing studies, as shown in Table 4. In the experiment, the proposed model and the model in [22] were trained by the same training set $\mathcal{T}$. The models ALBERT [24], DistilBERT [25], and ELECTRA [27] in Table 4 are the lightweight BERT models. These models have been pre-trained and can be directly used for SQL injection detection. However, in this study, they were subsequently fine-tuned by the training set $\mathcal{T}$ to further optimize their detection performance.

Table 4. Performance of various models for SQL injection detection for the test set $\mathcal{S}$. There are 24,707 queries in the test set. The length of each query is below 1000 characters.

Model	Accuracy	Precision	Recall	F1 Score	Weight Size
Proposed Model	98.98%	98.01%	99.84%	98.92%	69,269
Multi-Model [22]	97.88%	97.17%	98.35%	97.76%	77,008
ALBERT [24]	99.16%	98.31%	99.92%	99.11%	11,685,122
DistilBERT [25]	99.60%	99.25%	99.90%	99.57%	66,955,010
ELECTRA [27]	99.33%	98.72%	99.88%	99.30%	13,549,314

The evaluations of all the models in Table 4 are based on the same test set $\mathcal{S}$ with 24,707 queries. It can be observed from Table 4 that the proposed model has lowest weight size compared to the existing models. In particular, its weight size is only 0.1% of that of the DistilBERT [25] (i.e., 69,269 versus 66,966,010). In addition, it has comparable precision rate, recall rate, F1 score and accuracy values to those of ALBERT [24], DistilBERT [25], and ELECTRA [27]. The proposed model also has superior precision rate, recall rate, and accuracy values over those of Multi-Model [22]. We can conclude from the results that the proposed model is able to provide accurate SQL injection detection even with a small set of weights.

A special type of attacks in the test set $\mathcal{U}$ was also considered in this study, where the length of each query in $\mathcal{U}$ exceeded 1000 characters. Note that the training set $\mathcal{T}$ consists of only the SQL queries with short lengths. Therefore, the evaluations of the models on the test set $\mathcal{U}$ reveal the robustness of the models on the individual attacks with long

lengths. Table 5 reveals the corresponding results. We can see from the table that the proposed model and models in [24,25,27] were able to maintain high accuracy, precision rate, recall rate and F1 score values for the test set $\mathcal{U}$. Compared to the models in [24,25,27], the proposed model is still advantageous because it requires a significantly lower memory size for storing weights. Although the proposed model and the model in [22] have similar weight sizes, the proposed model has higher accuracy, precision rate, recall rate and F1 score values. The proposed model has the superior performance because its self-attention operations are based on the skeleton queries after removing noisy information. That is, the proposed model is able to abstract only the useful information for subsequent detection. It could then be a cost-effective solution for the SQL injection detection applications.

Table 5. Performance of various models for SQL injection detection for the test set $\mathcal{U}$. There are 500 queries in the test set. The length of each query is above 1000 characters.

	Proposed Model	**Multi-Model [22]**	**ALBERT [24]**	**DistilBERT [25]**	**ELECTRA [27]**
Accuracy	96.60%	82.40%	96.40%	96.60%	97.40%
Precision	99.23%	98.75%	99.74%	96.59%	98.50%
Recall	96.50%	79.00%	95.75%	99.25%	98.25%
F1 Score	97.85%	87.78%	97.70%	97.90%	98.37%

To further elaborate the effectiveness of the proposed model, Table 6 shows the average inference time of different models on various edge devices over the test set $\mathcal{S}$. In the experiments, the average inference time was measured as the average CPU time required to produce detection result per SQL query in the test set $\mathcal{S}$. The table also includes the standard deviation for each model in each edge device. There were two edge devices considered in the experiments: Up Board and PC. The Up Board uses an Intel Atom X5 Z8350 CPU and 4 GB of main memory. In the PC, the CPU is an Intel Core I9 9900K, and size of its main memory is 64 GB. The PC is also equipped with an NVIDIA GPU RX 3080 Ti.

Table 6. Average inference time of various models over the test set $\mathcal{S}$ on different edge devices for SQL injection detection.

		Proposed Model	**Multi-Model [22]**	**ALBERT [24]**	**DistilBERT [25]**	**ELECTRA [27]**
UP Board	Average Time	175.92 ms	187.47 ms	NA	NA	NA
	Standard Deviation	0.006	0.008	NA	NA	NA
PC	Average Time	51.55 ms	76.82 ms	148.23 ms	123.51 ms	121.26 ms
	Standard Deviation	0.010	0.010	0.097	0.121	0.062

We can see from Table 6 that the average inference time of the proposed model is lower than existing models for each edge device. The proposed model has lower computation complexities for inference because of its small weight size. As a result, it can be accommodated in the Up Board for realtime inference operations. By contrast, although ALBERT [24], DistilBERT [25], and ELECTRA [27] are lightweight BERT models, their weight sizes still exceed the memory capacity of Up Board. Therefore, they can only be deployed in the PC with sufficient memory size for online inference. This limits the flexibility of the existing lightweight BERT models for SQL injection detection applications. All these facts demonstrate the effectiveness of the proposed model.

5. Conclusions

The proposed neural network model based on multi-head self-attention has been found to be effective for SQL injection detection. The model has the advantages of low cost for model deployment, fast inference computation, and high SQL injection detection performance. While having comparable detection accuracy, precision, and recall rate values to the existing lightweight BERT models, the proposed model requires significantly lower weight size. In fact, the weight size of the proposed model is only 69,269, which is 0.1% of that of the DistilBERT model. Therefore, the proposed model can be deployed in edge devices with limited memory capacity such as UP Board. The average inference time of the proposed model for the SQL injection detection for each query is only 51.55 ms on a PC. Furthermore, the average inference time for the Up Board is 175.92 ms. For the test set where the length of each query is less than 1000 characters, the accuracy, precision rate, recall rate, and F1 score of the proposed model are 98.98%, 98.01%, 99.84%, and 98.92%, respectively. The detection performance can be effectively maintained for the test set with queries above 1000 characters in length. It can be concluded from the experimental results that proposed model is an effective alternative for applications requiring fast SQL injection detection with high accuracy and low deployment overhead.

Author Contributions: Conceptualization, R.-T.L. and W.-J.H.; methodology, R.-T.L. and T.-M.T.; software, R.-T.L.; validation, R.-T.L., W.-J.H., and T.-M.T.; investigation, R.-T.L.; resources, W.-J.H. and T.-M.T.; writing—original draft preparation, W.-J.H.; writing—review and editing, T.-M.T.; supervision, W.-J.H.; project administration, W.-J.H. and T.-M.T.; funding acquisition, W.-J.H. All authors have read and agreed to the published version of the manuscript.

Funding: The original research work presented in this paper was made possible, in part, by the National Science and Technology Council, Taiwan, under grants MOST 111-2221-E-003-009-MY2 and NSTC 113-2221-E-003-027-MY2.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The training and test sets supporting the reported results can be found in the link https://www.kaggle.com/datasets/rayten/sql-injection-dataset (accessed on 8 November 2024).

Acknowledgments: The authors would like to thank Zi-Yao Lin for the collections of test set for the evaluation of neural network models.

Conflicts of Interest: The author Tsung-Ming Tai is employed by the company "NVIDIA". 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.

Abbreviations

The following abbreviations are used in this manuscript:

ALBERT	A Little Bidirectional Encoder Representations from Transformers
BCP	Binary Cross-Entropy
BERT	Bidirectional Encoder Representations from Transformers
bi-LSTM	bidirectional Long Short-Term Memory
CNN	Convolutional Neural Network
ELECTRA	Efficiently Learning an Encoder that Classifies Token Replacements Accurately
FN	False Negative
FP	False Positive

GRU	Gated Recurrent Unit
ML	Machine Learning
NLP	Natural Language Processing
OWASP	Open WEB Application Security Project
PC	Personal Computer
RTD	Replaced Token Detection
SQL	Structured Query Language
SVM	Support Vector Machine
TF-IDF	Term Frequency–Inverse Document Frequency
TN	True Negative
TP	True Positive

Appendix A. Frequently Used Symbols

Table A1. A list of symbols used in this study.

B	The loss function for training.
$\mathbf{C}$	The positional encoding matrix.
d	The scaling factor for self-attention operation.
$\mathbf{F}$	The combined embedding representation matrix for tokens and semantic labels.
f_1	The mapping from a token to a numerical vector.
f_2	The mapping from a semantic label to a numerical vector.
h	Number of heads.
$H(\mathbf{X})$	Multi-head self-attention operation for matrix $\mathbf{X}$.
$\mathbf{K}$	The key matrix for single-head self-attention operations.
$\mathbf{K}_i$	The i-th key matrix for multi-head self-attention operations.
L	Number of queries in the training dataset.
M_1	The dimension of numerical vectors produced by f_1.
M_2	The dimension of numerical vectors produced by f_2.
M	$M = M_1 + M_2$. The dimensions of $\mathbf{C}$, $\mathbf{F}$ and $\mathbf{U}$ are $N \times M$.
N	Number of tokens from an input SQL query.
$\mathbf{Q}$	The query matrix for single-head self-attention operations.
$\mathbf{Q}_i$	The i-th query matrix for multi-head self-attention operations.
$S(\mathbf{X})$	Self-attention operation for matrix $\mathbf{X}$ based on $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$.
$S_i(\mathbf{X})$	Self-attention operation for matrix $\mathbf{X}$ based on $\mathbf{Q}_i$, $\mathbf{K}_i$, $\mathbf{V}_i$.
s_j	The j-th query in the training set.
$\mathcal{S}$	The test set for the experiments, where the length of each query is below 1000 characters.
$\mathcal{T}$	The training set for the experiments, where the length of each query is below 1000 characters.
$\mathcal{U}$	The test set for the experiments, where the length of each query is below 1000 characters.
$\mathbf{U}$	The encoded representation matrix produced by the input embedding section.
$\mathcal{V}$	The set of vocabularies for the SQL-specific tokenizer.
$\mathbf{V}$	The value matrix for single-head self-attention operations.
$\mathbf{V}_i$	The i-th value matrix for multi-head self-attention operations.
$\mathbf{W}_K$	The weight matrix for the computation of $\mathbf{K}$.
$\mathbf{W}_{K_i}$	The weight matrix for the computation of $\mathbf{K}_i$.
$\mathbf{W}_M$	The weight matrix for combining the results of different single-head self-attentions.
$\mathbf{W}_Q$	The weight matrix for the computation of $\mathbf{Q}$.
$\mathbf{W}_{Q_i}$	The weight matrix for the computation of $\mathbf{Q}_i$.
$\mathbf{W}_V$	The weight matrix for the computation of $\mathbf{V}$.
$\mathbf{W}_{V_i}$	The weight matrix for the computation of $\mathbf{V}_i$.
$\mathbf{X}$	Input matrix for self attention operations.
z	The estimated probability that the input query is malicious.
η	The threshold value for determining whether the input query is malicious.

References

1. OWASP Top Ten. Available online: https://owasp.org/www-project-top-ten (accessed on 22 April 2024).
2. Alnabulsi, H.; Islam, M.R.; Mamun, Q. Detecting SQL Injection Attacks Using SNORT IDS. In Proceedings of the Asia-Pacific World Congress on Computer Science and Engineering, Nadi, Fiji, 4–5 November 2014.
3. Maheshwarkar, B.; Maheshwarkar, N. SIUQAPTT: SQL Injection Union Query Attacks Prevention Using Tokenization Technique. In Proceedings of the International Conference on Information and Communication Technology for Competitive Strategies, Udaipur, India, 4–5 March 2016.
4. Katole, R.; Sherekar, S.; Thakare, V.M. Detection of SQL Injection Attacks by Removing the Parameter Values of SQL Query. In Proceedings of the International Conference on Inventive Systems and Control, Coimbatore, India, 19–20 January 2018; pp. 736–741.
5. Russell, S.J.; Norvig, P. *Artificial Intelligence: A Modern Approach*, 4th ed.; Pearson: Hoboken, NJ, USA, 2023.
6. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016.
7. Kamtuo, K.; Soomlek, C. Machine Learning for SQL Injection Prevention on Server-Side Scripting. In Proceedings of the IEEE International Computer Science and Engineering Conference, Chiang Mai, Thailand, 14–17 December 2016.
8. Hosam, E.; Hosny, H.; Ashraf, W.; Kaseb, A. SQL Injection Detection Using Machine Learning Techniques. In Proceedings of the International Conference on Soft Computing and Machine Intelligence, Cairo, Egypt, 26–27 November 2021; pp. 15–20.
9. Aggarwal, P.; Kumar, A.; Michael, K.; Nemade, J.; Sharma, S.; Kumar, P. Random Decision Forest Approach for Mitigating SQL Injection Attacks. In Proceedings of the IEEE International Conference on Electronics, Computing and Communication Technologies, Bangalore, India, 9–11 July 2021.
10. Uwagbole, S.O.; Buchanan, W.J.; Fan, L. Applied Machine Learning Predictive Analytics to SQL Injection Attack Detection and Prevention. In Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management, Lisbon, Portugal, 8–12 May 2017.
11. Gogoi, B.; Ahmed, T.; Dutta, A. Defending Against SQL Injection Attacks in Web Applications Using Machine Learning and Natural Language Processing. In Proceedings of the IEEE India Council International Conference, Guwahati, India, 19–21 December 2021.
12. Abaimov, S.; Bianchi, G. CODDLE: Code-Injection Detection with Deep Learning. *IEEE Access* **2019**, *7*, 128617–128627. [CrossRef]
13. Chen, D.; Yan, Q.; Wu, C.; Zhao, J. SQL Injection Attack Detection and Prevention Techniques Using Deep Learning. *J. Phys. Conf. Ser.* **2021**, *1757*, 012055. [CrossRef]
14. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv* **2019**, arXiv:1810.04805.
15. Seyyar, Y.E.; Yavuz, A.G.; Unver, H.M. An Attack Detection Framework Based on BERT and Deep Learning. *IEEE Access* **2022**, *10*, 68633–68644. [CrossRef]
16. Lodha, S.; Gundawar, A. SQL Injection and Its Detection Using Machine Learning Algorithms and BERT. In *Cognitive Computing and Cyber Physical Systems*; Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering; Gupta, N., Pareek, P., Reis, M., Eds.; Springer: Berlin, Germany, 2023; Volume 472.
17. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomes, A.N.; Kaiser, L.; Polosukhin, I. Attention Is All you Need. In Proceedings of the Conference on Neural Information Processing System, Long Beach, CA, USA, 4–9 December 2017.
18. Galassi, A.; Lippi, M.; Torroni, P. Attention in Natural Language Processing. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *32*, 4291–4308. [CrossRef] [PubMed]
19. Li, Y.; Zhang, B. Detection of SQL Injection Attacks Based on Improved TFIDF Algorithm. *J. Phys. Conf. Ser.* **2019**, *1395*, 012013. [CrossRef]
20. Ghozali, I.; Asyari, M.F.; Triarjo, S.; Ramadhani, H.M.; Studiawan, H.; Shiddiqi, A.M. A Novel SQL Injection Detection Using Bi-LSTM and TF-IDF. In Proceedings of the Conference on Information and Network Technologies, Okinawa, Japan, 21–23 May 2022.
21. Liu, M.; Li, K.; Chen, T. DeepSQLi: Deep Semantic Learning for Testing SQL Injection. *arXiv* **2020**, arXiv:2005.11728.
22. Hsiao, W.C.; Wang, C.H. Detection of SQL Injection and Cross-site Scripting Based on Multi-model CNN Combined with Bidirectional GRU and Multi-head Self-attention. In Proceedings of the International Conference on Computer Communication and the Internet, Fujisawa, Japan, 23–25 June 2023; pp. 142–150.
23. Liu, Y.; Dai, Y. Deep Learning in Cybersecurity: A Hybrid BERT-LSTM Network for SQL Injection Attack Detection. *IET Inf. Secur.* **2024**, *1*, 5565950. [CrossRef]
24. Lan, Z.; Chen, M.; Goodman, S.; Gimpel, K.; Sharma, P.; Soricut, R. ALBERT: A Lite BERT for Self-Supervised Learning of Language Representations. In Proceedings of the International Conference on Learning Representations, Addis Ababa, Ethiopia, 27–30 April 2020.
25. Sanh, V.; Debut, L.; Chaumond, J.; Wolf, T. DistilBERT, a distilled version of BERT: Smaller, Faster, Cheaper and Lighter. *arXiv* **2020**, arXiv:1910.01108.

26. Bokolo, B.G.; Chen, L.; Liu, Q. Detection of Web-Attack Using DistilBERT, RNN, and LSTM. In Proceedings of the International Symposium on Digital Forensics and Security, Chattanooga, TN, USA, 11–12 May 2023.
27. Clark, K.; Luong, M.-T.; Le, Q.V.; Manning, C.D. ELECTRA: Pre-Training Text Encoders as Discriminators Rather Than Generator. In Proceedings of the International Conference on Learning Representations, Addis Ababa, Ethiopia, 27–30 April 2020.
28. Tasdemir, K.; Khan, R.; Siddiqui, F.; Sezer, S.; Kurugollu, F.; Yengec-Tasdemir, S.B.; Bolat, A. Advancing SQL Injection Detection for High-Speed Data Centers: A Novel Approach Using Cascaded NLP. *arXiv* **2023**, arXiv:2312.13041.
29. Lee, I.; Jeong, S.; Yeo, S.; Moon, J. A Novel Method for SQL Injection Attack Detection Based on Removing SQL Query Attribute Values. *Math. Comput. Model.* **2012**, *55*, 58–68. [CrossRef]
30. SQL Injection Data Set. Available online: https://www.kaggle.com/datasets/rayten/sql-injection-dataset (accessed on 8 November 2024).
31. Kudo, T.; Richardson, J. SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing. In Proceedings of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Brussels, Belgium, 31 October–4 November 2018.