**Chapter 3**

**Conception and Implementation**

**3.1Introduction**

The chapter describes the models' design and implementation for SQL injection detection. Different architectures evaluated in our work include the traditional machine learning, deep learning, and transformer-based approaches. The rationale behind this approach is to assess and compare the models' performances on the same dataset in order to find the one most suitable for accurately detecting SQL injection attempts. The next sections describe the dataset employed, as well as the preprocessing done on it, followed by the design for each of the models and the evaluation strategy considered for training and testing.

**3.2. Dataset**

To train a effective deep learning model successfuly, we need to make sure the dataset is properly chosen. For our project, we needed a dataset that includes samples classified into two categories: queries containing SQL injection and those that do not.Using this dataset, the trained model will be able to detect whether a query is a SQL injection or a normal query.

 We found a collection of "SQL Injection Datasets" on Kaggle. Among these datasets, we discovered a collection of datasets that put togetherby a person named " Syed Saqlain Hussain Shah" and it contain three (3) datasets.

**SQLi.csv** contains 3951 samples with 78% classified as normal queries and 28% as malicious queries.

**SQLiV2.csv** contains 33726 samples with 66% classified as normal queries and 34% as malicious queries.

**SQLiV3.csv** contains 30873 samples with 62% classified as normal queries and 37% as malicious queries and 1% as other.

After training and testing multiple machine learning and deep learning models on the three datasets, we found SQLIV3.CSV to be the most suitable dataset for our project Models trained on this dataset were better able to detect SQL injection attacks and benign queries.Compared to other datasets, SQLIV3.csv consisyently led to a higher accuracy and better generalization across different models, making it the optimal choice for our final implementation.

Before training the models, the SQLIV3.csv dataset needs to be preprocessed as follows :

-We found two trailing commas (,,) at the end of all the lines in the CSV file and removed these unnecessary characters using a Python script.

- Filters the rows to keep only those that contain exactly 2 columns and removes empty rows.

By applying the above preprocessing steps, we ultimately created a partitioned dataset containing:

* **30,614 SQL queries**.
* Each entry consists of a **Sentence** (the SQL query) and a **Label** (0 for benign, 1 for SQL Injection).

**Label distribution**:

* **Normal (Label = 0)**: 19,268 queries
* **Malicious (Label = 1)**: 11,346 queries



**3.3 Models Implemented**

This section on models presents the various models that have been developed for detecting SQL injections. A number of machine learning models and deep learning models were trained and evaluated with varying architecture and hyperparameters. The aim is to see how traditional techniques and modern, state-of-the-art techniques differ in input to achieve results on the same dataset.

For each model, we describe the structure, the main hyperparameters used during training, and the results obtained. These models are as follows:

Support Vector Machine (SVM)

Logistic Regression (LR)

Multilayer Perceptron (MLP)

Simple Neural Network (SN)

Recurrent Neural Network (RNN)

Long Short-Term Memory (LSTM)

BERT Transformer Model (dedicated in Chapter 4)

**3.3.1 Support Vector Machine (SVM)**

We used a Support Vector Machine with a linear kernel (`kernel='linear'`) and regularization parameter C = 1. The input queries were transformed into vectors by applying TF-IDF with 3000 features as the maximum. We dropped duplicates and split the dataset into 80% training and 20% testing sets prior to model training.

**Training Performance:**

* **Accuracy:** 99.44%
* **Precision:** 99.73%

Evaluation on Test Set: :

- Accuracy: 98.27%

- Precision: 100%

- Recall: 99.13%

- F1-score: 99.13%

Additional Evaluation Metrics:

 **Detected SQL injection queries:** 11,232 out of 11,424

 **Detected benign queries:** 22,263 out of 22,301

**3.3.2 Logistic Regression (LR)**

We trained a Logistic Regression classifier to detect SQL injection payloads using TF-IDF features (maximum of 3000 features). The classifier was initialized with solver='liblinear' and penalty='l1'. Duplicate queries were removed, and the dataset was split into 80% training and 20% testing.

|  |
| --- |
| # Train a logistic regression model with L1 regularization  lrc = LogisticRegression(solver='liblinear', penalty='l1')  lrc.fit(X\_train, y\_train)  # Predict labels on the test set  y\_pred = lrc.predict(X\_test) |

**Training Performance:**

* Accuracy: 98.12%
* Precision: 99.72%

**Test Performance:**

* Accuracy: 95.36%
* Precision: 100%
* Recall: 95.36%
* F1-score: 97.63%

**Additional Evaluation Metrics:**

* **Detected SQL injection queries:** 10,902 out of 11,424
* **Detected benign queries:** 22,196 out of 22,303

**3.3.3 Multilayer Perceptron (MLP)**

Using ReLU activation functions, we trained a Multi-Layer Perceptron (MLP) neural network with three hidden layers comprising 512, 256, and 128 units respectively. For binary classification, the output layer employed a sigmoid activation function. Using TF-IDF with a maximum of 3000 characteristics, input queries were vectorized.  
  
Using the binary cross-entropy loss function and the SGD optimizer with a learning rate of 0.01, the model was compiled. Using 20% of the training set for validation, it was trained for 27 epochs on 80% of the data with early stopping activated (patience = 3). To guarantee clean input data, duplicates were deleted from the test set prior to evaluation.

|  |
| --- |
| model = Sequential([  Dense(512, input\_dim=3000, activation='relu'),  Dense(256, activation='relu'),  Dense(128, activation='relu'),  Dense(1, activation='sigmoid')  ])  model.compile(loss='binary\_crossentropy', optimizer=SGD, metrics=['accuracy'])  early\_stop = EarlyStopping(patience=3, restore\_best\_weights=True)  history = model.fit(train\_x, train\_y, epochs=27 , batch\_size=32, verbose=2, validation\_split=0.2,callbacks=[early\_stop]) |

|  |
| --- |
| test\_x = vectorizer.transform(test.dropna(subset=['Sentence']).drop\_duplicates(subset='Sentence')['Sentence']).toarray()  predicted\_classes = (model.predict(test\_x) >= 0.5).astype(int) |

**Training Performance:**

* Accuracy: 99.39%
* Final loss: 0.034

**Evaluation on Test Set:**

* Accuracy: 98.68%
* Precision: 100%
* Recall: 98.68%
* F1-score: 99.33%

**Additional Evaluation Metrics:**

* Correctly detected SQL injection payloads: 11,272 out of 11,424
* Correctly detected benign payloads: 22,146 out of 22,303

**3.3.5 Recurrent Neural Network (RNN)**

To discover sequential patterns in SQL injection attempts, we built a Recurrent Neural Network (RNN) upon tokenized and padded input sentences, retaining SQL syntax and casing by turning off character filters and preserving case sensitivity.

The architecture consists of an embedding layer (128 dimensions), followed by a single SimpleRNN layer with 128 units. A dropout layer with a rate of 0.5 is applied to reduce overfitting. This is followed by a Dense layer with 64 units using ReLU activation and L2 regularization (λ = 0.01), and a final sigmoid output layer for binary classification. The model was trained for 20 epochs using the Adam optimizer (learning rate = 0.001) and binary cross-entropy as the loss function. Early stopping was applied with a patience of 3 epochs, and the best model weights were restored based on validation performance.

|  |
| --- |
| model = Sequential([  Embedding(input\_dim=15000, output\_dim=128, input\_length=max\_len),  SimpleRNN(128, return\_sequences=False),  Dropout(0.5),  Dense(64, activation='relu', kernel\_regularizer=regularizers.l2(0.01)),  Dense(1, activation='sigmoid')  ]) early\_stop = EarlyStopping(patience=3, restore\_best\_weights=True)  optimizer = optimizers.Adam(learning\_rate=0.001)  model.compile(loss='binary\_crossentropy', optimizer=optimizer, metrics=['accuracy'])  history = model.fit(  X\_train, y\_train,  epochs=20,  batch\_size=64,  validation\_data=(X\_val, y\_val),  callbacks=[early\_stop]  ) |

**Training Performance:**

* Final loss: 0.0499
* Accuracy: 99.22%

.

**Test Performance:**

Accuracy: **98.34%**  
Precision: **100.00%**  
 Recall: **98.34%**  
 F1 Score: **99.16%**

**Additional Evaluation Metrics:**

* **Detected SQL injection queries:** 11,237 out of 11,424
* **Detected benign queries:** 22,158 out of 22,303

**3.3.6 Long Short-Term Memory (LSTM)**

To further enhance the detection of sequential patterns in SQL injection scenarios, we implemented a Long Short-Term Memory (LSTM) network on tokenized and padded input sentences, allowing SQL syntax to remain intact while preserving case sensitivity by having character filters turned off and case sensitivity on.

The architecture is built up of an embedding layer (256 dimensions), two stacked LSTM layers (the first has 256 units, the second has 128 units), Dense layer (64 units ReLU activation), and a final sigmoid output layer for binary classification. The model was trained for 30 epochs using the Adam optimizer (learning rate = 0.0001) and used binary cross-entropy as our loss function.

|  |
| --- |
| model = Sequential([  Embedding(input\_dim=15000, output\_dim=256, input\_length=max\_len),  LSTM(256, return\_sequences=True),  Dropout(0.3),  LSTM(128),  Dense(64, activation='relu'),  Dense(1, activation='sigmoid')  ]) |

**Training Performance:**

* Final loss: 0.0417
* Accuracy: 99.42%

.

**Test Performance:**

Accuracy: **99.35%**  
Precision: **100.00%**  
 Recall: **99.35%**  
 F1 Score: **99.68%**

**Additional Evaluation Metrics:**

* **Detected SQL injection queries:** 11,350 out of 11,424
* **Detected benign queries:** 21,715 out of 22,303

This shows that the model is effective in classifying normal queries correctly but with high precision and recall. In a real-world situation, it is important for the system to be able to distinguish between legitimate queries and malicious queries.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithme | Class | Accuracy | Precision | Recall | F1-Score |
| SVM | Malicious  Benign  Average | 98.32%  99.83%  99.075% | 100%  100% | 98.32%  99.83%  99.075% | 99.15%  99.91%  99.53% |
| LR | Malicious  Benign  Average | 95.36%  99.52%  97.44% | 100%  100% | 95.36%  99.52%  97.44% | 97.63%  99.76%  98.70% |
| MLP | Malicious  Benign  Average | 98.67%  99.30%  98.99% | 100%  100% | 98.67%  99.30%  98.99% | 99.33%  99.65%  99.49% |
| RNN | Malicious  Benign  Average | 98.34%  99.35%  98.85% | 100%  100% | 98.34%  99.35%  98.85% | 99.16%  99.68%  99.42% |
| LSTM | Malicious  Benign  Average | 99.35%  97.35%  98.35% | 100%  100% | 99.35%  97.35%  98.35% | 99.68%  98.66 %  99.17% |