Table of contents

List of Figures

3. Chapter 3

Conception and Implementation

3.1 Introduction

In this chapter, we will discuss the design and implementation of our model for detecting SQL injections using deep learning. We will describe the general conception of our work and the materials used, including the dataset and the type of deep learning architecture employed. Furthermore, we will cover the preprocessing steps taken to ensure the accuracy and efficiency of our system. By detailing our approach to design and implementation, we aim to provide a comprehensive understanding of our methodology for detecting SQL injections through the use of deep learning.

3.2 General conception of the solution

The deep learning model that we have developed will be used for detecting SQL injection attacks in web applications. The model is based on the Bidirectional Encoder Representations from Transformers (BERT) architecture, which has been fine-tuned on a dataset of SQL injection attacks and normal SQL queries. The model is designed to function as a middleware layer (API or a WAF) between the web application and the database server as shown in Figure 3.1 Sql injection Detection Tool conception and architecture.

Our **Sqli Detection Model** analyzes incoming queries and detects any suspicious patterns that may indicate an SQL injection attack, it can be used on Web Application Firewalls (WAF) or as an API. Once the SQL injection is detected, the tool can either block the request or alert the system administrator, depending on the configuration.

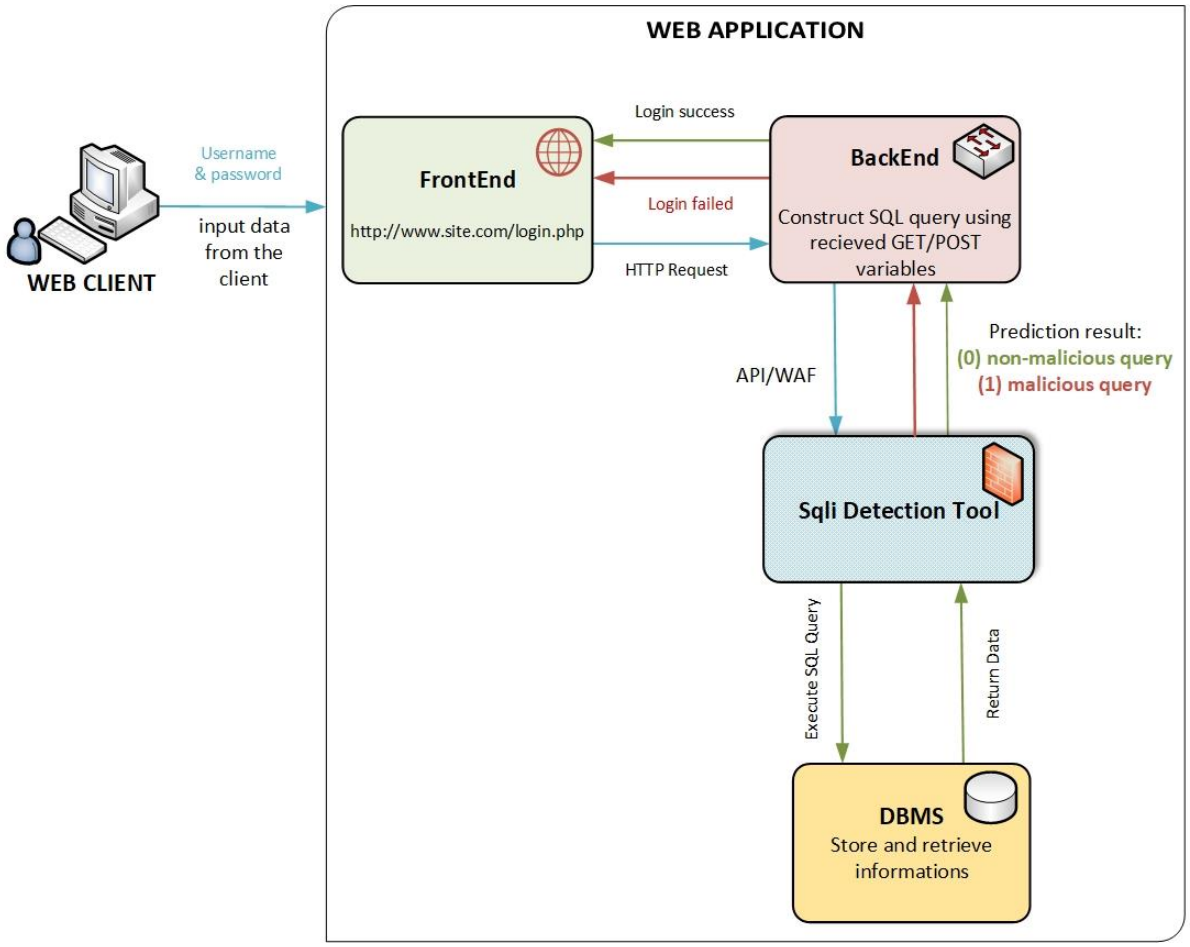

Figure 3.1 Sql injection Detection Tool conception and architecture.

3.3 Chosen model: BERT

BERT (Bidirectional Encoder Representations from Transformers) is a state-of-the-art language model that has revolutionized natural language processing tasks. It is based on transformer architecture, which enables it to effectively capture contextual information from input text. Unlike traditional models that process text sequentially, BERT takes into account the entire context surrounding each word by using a bidirectional approach. By pre-training on large amounts of unlabeled text data, BERT learns to generate high-quality word representations that encode rich semantic and syntactic information. These pre-trained representations can then be fine-tuned on specific tasks, such as detecting SQL injections in our case. With its ability to understand the nuanced context of language, BERT has demonstrated

exceptional performance on various natural language processing tasks, making it a suitable choice for enhancing the detection and prevention of SQL injection attacks in this study.

BERT has two variants: BERT-base and BERT-large, which differ in the number of layers and parameters. BERT-base has 12 transformer layers and 110 million parameters, while BERT-large has 24 transformer layers and 340 million parameters [19].

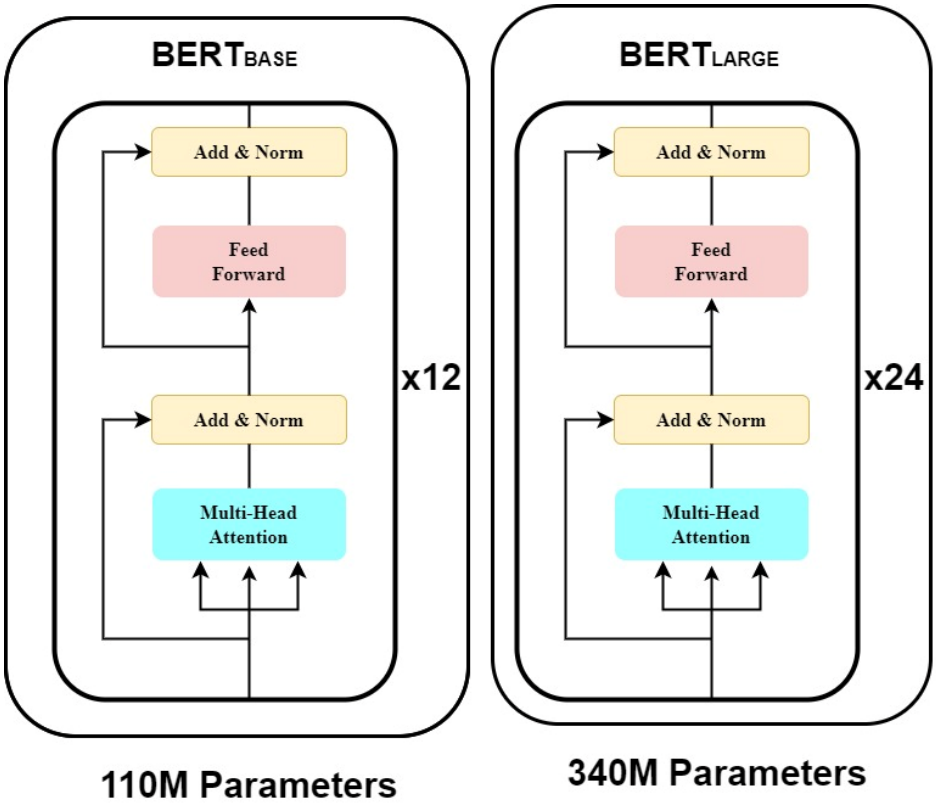

Figure 3.2 BERT model size.

In this study, the choice of the BERT-Base model was driven by the size of our dataset, which consisted of approximately 22,000 samples. Given the limited size of the dataset, using BERT-Large would not have been the most suitable option.

3.3.1 BERT architecture

BERT uses the Transformer architecture, which is an attention mechanism designed to learn contextual relationships between words or sub-words in a text. In its original form, the Transformer consists of two mechanisms: an encoder that processes the text input and a decoder

that generates predictions for the task. However, since BERT's objective is to generate a language model, only the encoder mechanism is necessary.

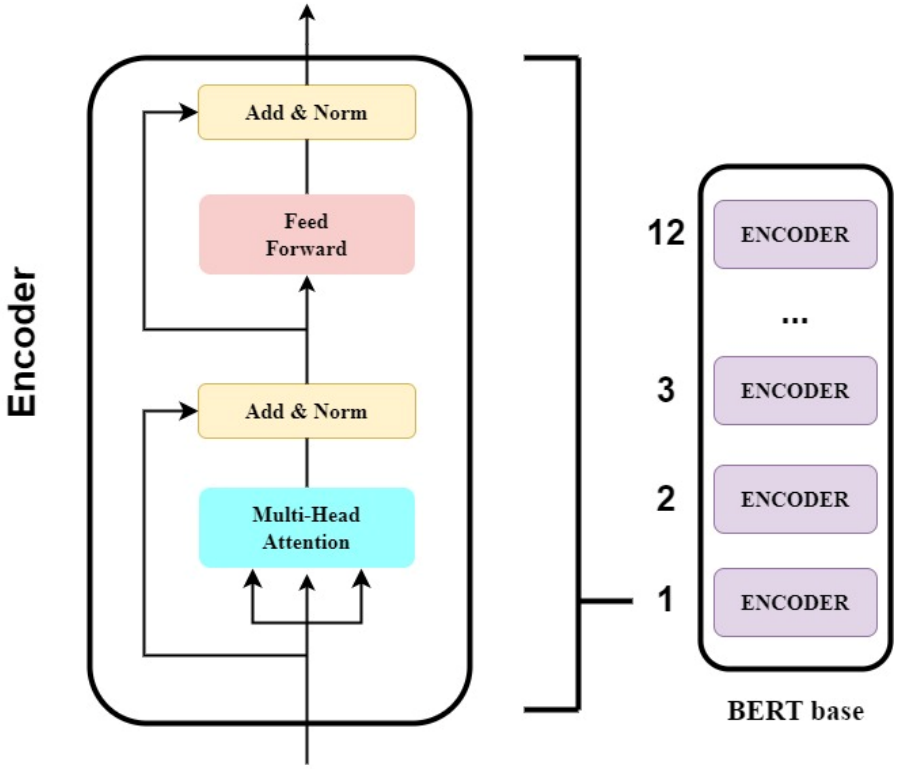

Figure 3.3 BERT model architecture.

The encoder in BERT comprises a stack of N = 6 identical layers, with each layer consisting of two sub-layers. The first sub-layer is a multi-head self-attention mechanism, which allows the model to attend to different positions within the input sequence simultaneously. This mechanism employs multiple attention heads to capture dependencies between words or sub-words in a text. The second sub-layer is a simple feed-forward network, which processes the outputs from the previous self-attention layer. This network consists of fully connected layers and applies a linear transformation to each position independently. Both sub-layers in each encoder layer are followed by a normalization layer. This normalization helps in stabilizing the learning process by normalizing the outputs of each sub-layer. Additionally, to ensure smooth connectivity between layers in the model, all sub-layers produce outputs of dimension dmodel = 512, ensuring consistent input and output dimensions throughout the architecture.

3.3.2 BERT for text classification

When processing input, BERT expects a sequence of tokens with a maximum length of 512 tokens. The sequence can be divided into one or two segments. The first token of the sequence is a special token that holds a special classification embedding. For text classification tasks, BERT considers the final hidden state h of the first token as the representation of the entire sequence. This hidden state captures the information from the entire input sequence and serves as a condensed representation. To make predictions, a simple softmax classifier is added on top of BERT. The classifier employs a task-specific parameter matrix W to compute the probability of each label c given the representation h.

3.3.3 Why BERT was chosen

In the context of SQL injection detection, BERT was chosen for its ability to learn rich representations of text data, including queries, comments, and other textual elements that are typically associated with SQL injection attacks. BERT has been shown to outperform other state-of-the-art models on various NLP tasks [20], making it a promising candidate for SQL injection detection. Its transformer-based architecture allows it to process input sequences in a bidirectional manner and generate contextualized word representations. Fine-tuning BERT on a labeled dataset of SQL queries allowed us to develop a model that can detect SQL injection attacks with high accuracy.

3.3.4 Fine-tuning BERT for SQL injection detection:

To use BERT for SQL injection detection, we fine-tuned the pre-trained BERT model on a labeled dataset of normal SQL queries and SQL injections. During fine-tuning, the model was trained to predict whether a given query is a SQL injection or not. The fine-tuning process involved adjusting the weights of the classification layer while keeping the weights of the pre-trained BERT layers fixed.

3.4 Presentation of development tools

3.4.1 Programming language

3.4.1.1 Python 

Python is a popular programming language in the field of machine learning and artificial intelligence due to its simple syntax, extensive libraries, and ease of use. Python provides a variety of libraries for machine learning such as TensorFlow, PyTorch, and Keras, making it a

go-to language for many data scientists and machine learning practitioners. Its libraries provide an extensive range of functionalities, from data preprocessing to complex neural network architectures [21].

Furthermore, Python's community is continuously contributing to its open-source libraries, ensuring a broad range of features and capabilities. Python is also known for its versatility as it can be used not only for machine learning but also for web development, scientific computing, and data analysis. However, it is important to note that while Python is a popular choice, it is not the only programming language used in machine learning. Other languages, such as R and Java, are also used for machine learning tasks [22].

3.4.2 Development environment

3.4.2.1 **Google Colab Pro**

Since training deep learning models often requires high-performance hardware or can be time-consuming, we selected Google Colab Pro as the platform for our project. Google Colab Pro is a cloud-based service that provides access to a powerful GPU and RAM, which allows us to train our model in a reasonable amount of time.

Specifically, Google Colab Pro provides access to a Tesla K80 GPU, which has 12 GB of GDDR5 VRAM and 4992 CUDA cores. This GPU is suitable for training deep learning models with moderate to high computational requirements.

In addition, **Google Colab Pro** also provides access to TPUs, which are specifically designed for accelerating deep learning computations. TPUs are available for users on a case-by-case basis and require a separate application process.

The TPU offered by Google Colab Pro is the TPU v3-8, which has 8 TPU cores and 64 GB of High Bandwidth Memory (HBM). This TPU is designed for high-throughput deep learning workloads and can accelerate training times by orders of magnitude compared to a traditional GPU.

Overall, the availability of both GPU and TPU on Google Colab Pro made it a suitable platform for our deep learning project, enabling us to train and test our model efficiently and effectively."

Additionally, Google Colab Pro provides a user-friendly interface that allows us to write and execute our code using Jupyter notebooks. This platform also offers other useful features such as version control, collaboration tools, and cloud storage for our data and code.

3.4.2.2 **Jupyter notebook**

Jupyter notebook is an open-source web application that allows users to create and share documents that contain live code, equations, visualizations, and narrative text. It supports over 40 programming languages, including Python, R, and Julia, making it a versatile tool for data analysis and machine learning [23].

Jupyter notebooks are interactive and allow users to execute code in a step-by-step manner, making it easy to debug and analyze results. They also support the use of Markdown, a markup language for text formatting, making it easy to create readable and well-structured documents. Jupyter notebooks are widely used in the data science community due to their flexibility, interactivity, and ease of use [24].

3.5 Dataset

In order to train a successful and effective deep learning model, the dataset must be carefully processed, to achieve this, we needed to find a Dataset consisting of samples divided into two classes: "malicious queries" and "non-malicious queries". By using this Dataset, the model can learn to distinguish between the two classes and accurately identify SQL injection attacks.

During our search for a suitable Datasets, we came across the "SQL Injection Dataset" list on the Kaggle platform [25]. We found a list of datasets that were created by “Syed Hussain” that contain Three (03) versions:

- ✓ **SQLi.csv** (723.15 kB) contains **3951** samples with **78%** classified as normal queries and **28%** as malicious queries.
- ✓ **SQLiV2.csv** (3.61 MB) contains **33726** samples with **66%** classified as normal queries and **34%** as malicious queries.
- ✓ **SQLiV3.csv** (2.32 MB) contains **30873** samples with **62%** classified as normal queries and **37%** as malicious queries and **1%** as other.

At first vision we choose the **SQLiV2** Dataset because of the size of samples in it in comparison to others, we trained our model with it in the first time but unfortunately, the results

were not satisfied while predicting normal queries. After analyzing the situation, we found that there were no normal queries in the dataset, only free text in place flagged as normal queries.

Because of the big issue in **SQLiV2.csv**, we tried the **SQLiV3.csv** Dataset**.** After reviewing it, we identified certain deficiencies that need to be cleared using some preprocessing steps like:

- ✓ Remove any unnecessary Strings or characters in the SQL statements. We found two commas (,,) at the end of all the queries, we removed these unnecessary commas using a Register-Expression technique on a python script.
- ✓ Remove not valid empty columns. We need only two valid columns, the **Statement** and the **Label**, in that Dataset we found two empty and not valid columns that were removed using the Office Excel Software.
- ✓ Remove empty and free text Rows. We found many empty and free text rows that were removed using the Office Excel Software.
- ✓ Remove wrong SQL queries. We found not valid SQL statements that were identified and cleared manually.

By applying the above-preprocessed steps, we finally came up with a partitioned version that has **11308 (≈ 50%)** "non-malicious queries" against **11291 (≈ 50%)** "malicious queries".

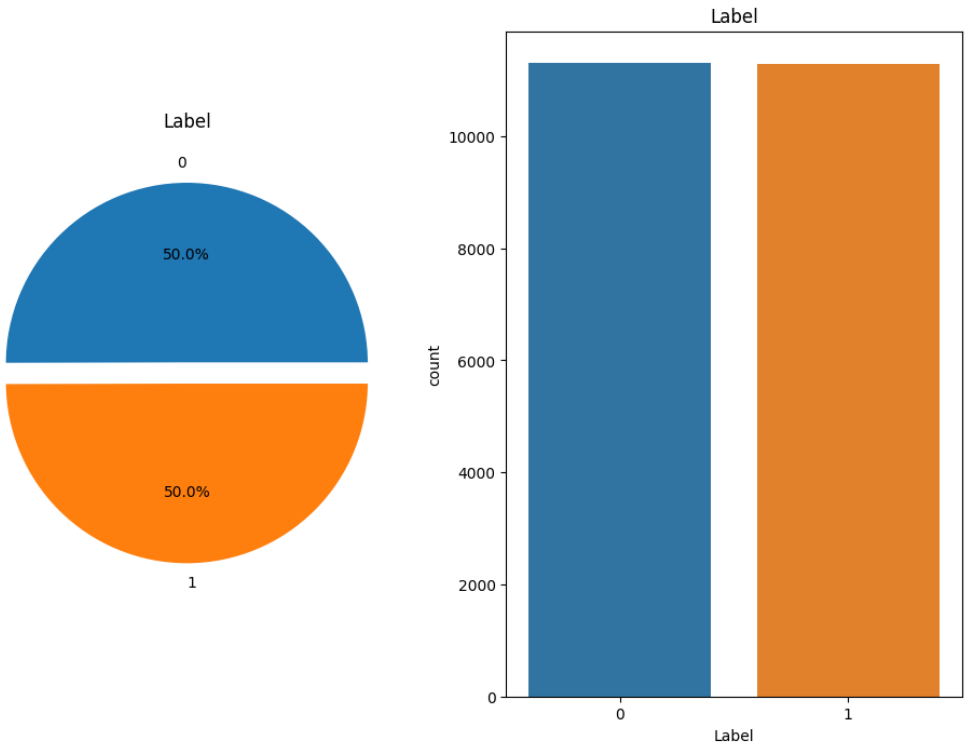

Figure 3.4 Dataset query classes distribution.

After clearing the different deficiencies, we came up with a new valid preprocessed Dataset that has sufficient diversity in both categories and is ready to be used in the next step. We ensured that our deep learning model would only train on preprocessed samples that belong to one of the two classes and could, therefore, learn to differentiate between malicious and non-malicious queries effectively.

3.6 Code and implementation

The implementation of a deep learning model can be a challenging task, especially when it involves complex architectures and large datasets. In this section, we will present the code and implementation details of our SQL injection detection model based on the BERT architecture.

3.6.1 Import necessary libraries and tools

Importing necessary libraries and tools is an essential step when working on any data science project. These libraries provide functionality for common data manipulation, exploration, and deep learning tasks. In this project, we used a number of libraries to preprocess and classify text data as the following:

numpy: NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. We used this library to work with numerical data in our project.

ktrain: ktrain is a lightweight wrapper for the Keras deep learning library to help simplify the training of neural networks. We used this library to build and train our text classification model.

pandas: Pandas is a library used for data manipulation and analysis. It provides data structures for efficiently storing and manipulating large datasets. We used this library to read in and preprocess our text data.

chardet: Chardet is a Python library used for character encoding detection. We used this library to ensure that our text data is properly encoded before processing.

matplotlib: Matplotlib is a data visualization library used for creating static, animated, and interactive visualizations in Python. We used this library to create visualizations of our data and model performance.

seaborn: Seaborn is a data visualization library based on matplotlib. It provides a high-level interface for creating informative and attractive statistical graphics. We used this library to visualize the distribution of our data.

pickle: Pickle is a Python module used for serializing and de-serializing Python objects. We used this library to save our trained model for future use.

3.6.2 Split and preprocess data for the BERT model

```
sentences = df['Sentence'].tolist()

labels = df['Label'].tolist()

(x_train, y_train), (x_test, y_test), preproc =
text.texts_from_array( sentences,labels,
preprocess_mode='bert', maxlen=500,
val_pct=0.2,class_names=list(set(labels))
)
```

Figure 3.5 Split data into training and testing sets and preprocess data for BERT model

In Figure 3.5 Split data into training and testing sets and preprocess data for BERT model, the code is splitting the data into training and testing sets and prepares it to be used for training the model. It starts by extracting the 'Sentence' and 'Label' columns from the pandas dataframe and creating two lists out of them. Next, the 'texts_from_array' method from the ktrain library is used to convert the data into arrays that can be used for training a BERT model. The 'preprocess_mode' parameter is set to 'bert', which means that the data will be preprocessed according to the requirements of the BERT model. val_pct=0.2 indicates that 20% of the data will be used for validation during training. The 'maxlen' parameter is set to 500, which means that the maximum length of a sentence is set to 500 tokens. The 'class_names' parameter is set to the unique labels in the training set, which will be used to create a mapping between the label values and their corresponding names. The method returns four variables: x_train and x_test

are the preprocessed arrays of sentences, y_train and y_test are the label arrays, and preproc is a preprocessor object that was used to preprocess the data.

3.6.3 Build the BERT model

```
  model = text.text_classifier('bert', (x_train, y_train),
preproc=preproc)
```

Figure 3.6 Build BERT model Python code.

In this code line, a BERT (Bidirectional Encoder Representations from Transformers) model is being built using the text_classifier function from the ktrain library. This function takes the name of the model as the first argument (in this case, 'bert'), the training data (x_train, y_train), and the preprocessing object preproc as input.

3.6.4 Train the BERT model

```
  learner = ktrain.get_learner(model=model,
                               train_data=(x_train, y_train),
                               val_data=(x_test, y_test),
                               batch_size=6)
  learner.fit onecycle(lr=2e-5, epochs=4)
```

Figure 3.7 Train BERT model Python code.

In the "Train BERT model" section, the model is trained using ktrain's get_learner method and the one-cycle policy, which involves training the model with a learning rate that linearly increases for the first half of the epochs and then linearly decreases for the second half of the epochs.

First, get_learner is called, which creates a Learner object for the specified model and training data (x_train, y_train). It also includes validation data (x_test, y_test) to monitor the model's performance during training, and the batch_size is set to 6.

Then, the fit_onecycle method is called on the learner object. This method trains the model using the one-cycle policy for a specified lr (learning rate) and number of epochs. In this case, the learning rate is set to 2e-5 and the number of epochs is 4.

During training, the model's loss and accuracy are displayed for each epoch. The goal is to minimize the loss and maximize the accuracy on the validation set to create a well-performing model.

3.6.5 Make predictions with the BERT model

```
predictor = ktrain.get_predictor(learner.model, preproc)

# make predictions

samples = [

"1'; DROP TABLE users;--;",
"INSERT INTO users (username, password) VALUES ('testuser',
'testpassword'); DROP TABLE users;",
"SELECT COUNT(*) FROM users WHERE username = 'admin' OR 1 = 1",
"UPDATE users SET password = 'newpassword' WHERE username =
'admin';" ,
"select * from generate_series ( 5980,5980,case
when ( 5980 = 5063 ) then 1 else 0 end ) limit 1--",
"SELECT TOP 3 * FROM growth SELECT * FROM catch 3SELECT * FROM
mainly",
"SELECT * FROM users WHERE username = '' OR 1:1",
"INSERT INTO column ( white, does, certain, curious, first, our
) VALUES ( 'rose', 'anyone'. close', 'remove', 'force', 'feet',
'fell' )",
"SELECT * FROM users WHERE username = '' OR 1=2 --' AND password =
'input_password'"]

prediction = predictor.predict(samples)
```

Figure 3.8 Make predictions with the trained model.

This code block shows how to use the BERT model trained to detect SQL injection attacks to make predictions on new data. The get_predictor() function loads the trained BERT model and pre-processing pipeline, which are necessary to make predictions on new data. The predict() function takes a single input and returns the predicted label.

3.7 Choice of hyperparameters

The choice of hyperparameters plays a critical role in the design and performance of our system for detecting SQL injections using deep learning. In this section, we discuss the key hyperparameters we selected and the reasoning behind these choices.

3.7.1 Preprocessing hyperparameters

Max Length: We set the maximum length of input sequences to 500. This value was determined based on the analysis of the dataset, ensuring that most SQL injection statements can be adequately captured within this limit.

Preprocess Mode: We utilized the "bert" preprocess mode, which applies BERT-specific tokenization and formatting to the input text data. This mode is specifically designed for BERT models and helps optimize the preprocessing step, enhancing the model's ability to understand the context and semantics of the text.

3.7.2 Model training hyperparameters

Batch Size: We chose a batch size of 6, which determines the number of training samples processed in each iteration. This value strikes a balance between training speed and memory consumption, considering the available computational resources and dataset size.

Learning Rate: We set the learning rate to 2e-5, a value recommended by Google for fine-tuning BERT models. This learning rate choice enables effective convergence during training while minimizing the risk of overshooting the optimal solution.

Number of Epochs: The model was trained for 4 epochs, meaning the entire training dataset was processed four times. This number of epochs allows the model to learn patterns and generalize well to the dataset without over-fitting.

3.7.3 Data split hyperparameters

Test Size: We partitioned the dataset into training and testing sets using a test size of 0.2 (20%). This split allocates 80% of the data for training and 20% for testing, ensuring a sufficient amount of data for evaluation while preserving a sizable training set.

By carefully selecting these Hyperparameters, including the test size, max length, preprocess mode, batch size, learning rate, and epochs, we aimed to optimize the performance of our

system for detecting SQL injections. These choices were based on prior knowledge, best practices and recommendations. The hyperparameters collectively contribute to the effectiveness and accuracy of our model in identifying SQL injection attacks.

3.8 Conclusion

In this chapter, we have described the design and implementation of a system for detecting SQL injections using deep learning. We utilized a dataset of SQL injection attacks and normal queries to train a BERT model for classification. The preprocessing steps involved converting the text data into a format suitable for BERT model input and splitting the data into training and testing sets. Our results showed that the trained model was effective in detecting SQL injection attacks with a high degree of accuracy. By sharing the details of our approach, we have provided a comprehensive understanding of how deep learning can be utilized for SQL injection detection.

4. Bibliographie

[19] M.-W. C. K. L. a. K. T. Jacob Devlin, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, Volume 1 (Long and Short Papers), 2019.

[20] H. Z. X. H. a. J. Z. Yan Zhang, Benchmarking Neural Network Models for SQL Injection Detection, 2020.

[21] A. Géron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow Concepts, Tools, and Techniques to Build Intelligent Systems, O'Reilly Media, 2019.

[22] S. &. M. V. Raschka, Python Machine Learning: Machine Learning and Deep Learning with Python, Packt Publishing, 2019.

[23] T. R.-K. B. P. F. G. B. E. B. M. F. J. Kluyver, Jupyter Notebooks—a publishing format for reproducible computational workflows, In ELPUB, 2016.

[24] A. T. A. &. H. J. D. Rule, Rapid prototyping interactive data visualizations with Jupyter notebooks, IEEE transactions on visualization and computer graphics, 2018.

[25] "Kaggle," 2023. [Online]. Available: https://www.kaggle.com/datasets/syedsaqlainhussain/sql-injection-dataset.