**4.1 Introduction**

In this chapter, we present our detection model using the BERT language model to prevent from the SQL injection attacks. We begin by laying down BERT's architecture and the reasons behind choosing it for this task. We then move on to explaining the preprocessing that must be done in order to make our dataset compatible with BERT's input requirements, and then move on to explaining the fine-tuning process employed in order to train the model for binary classification.

**4.2 General Conception**

The deep learning-based model introduced in this work will detect SQL injection attacks. The architecture of the model is built upon BERT (Bidirectional Encoder Representations from Transformers), and has been fine-tuned for binary classification using labeled SQL injection and benign SQL queries.

**4.3 Why BERT for SQL Injection Detection**

We selected BERT as our SQL injection detection model due to its high capacity to comprehend both meaning and context of textual information, particularly in structured input such as SQL queries. What makes BERT distinctively effective is the fact that it processes the whole input bidirectionally which helps it pick up on subtle patterns that might indicate an attack. Unlike older models that generally rely on basic keyword matching or predefined rules, BERT learns from the actual composition and intent of the query. After fine-tuning on a database consisting of normal and malicious SQL queries, the model was able to accurately identify suspicious inputs that could potentially represent injection attempts. Thus, BERT proves to be a reliable and effective tool for web application security improvement.

**4.4 BERT Architecture Overview**

The BERT model architecture is a multilayer bidirectional Transformer encoder, based on the encoder component of the original Transformer model. While the original Transformer use an encoder-decoder structure where the encoder processes input sequences using self-attention mechanisms and the decoder combines self-attention with encoder-decoder attention, BERT focuses exclusively on the encoder to learn deep contextual representations.

The BERT architecture is available in two main configurations : BERTBASE and BERTLARGE. BERTBASE consists of 12 encoder layers, 768 hidden units per layer, and 12 attentions heads, resulting in approximately 110 million parameters. In contrast, BERTLARGE extends the architecture to 24 encoder layers, 1024 hidden units, and 16 attentions heads, totaling around 340 million parameters.

As a result, BERT models are capable of learning more complex contextual representations from input sequences.

(hedi tbenli ma ndirouhech ) {The input to the model begins with a classification token, [CLS], followed by a sequence of words. The [CLS] token is exclusively utilized for classification. The input is then fed into the sequence of encoder layers, wherein each is executing self-attention mechanisms followed by feedforward neural networks, and then to the subsequent encoder layer.

The output of the model is a 768-dimensional hidden vector for BERTBASE. In classification, the [CLS] token's output is typically used as the representation of the input sequence and passed through a classifier layer. helps generate more detailed language representations which specifically benefit tasks dealing with syntax-sensitive input such as SQL queries.}

We decided to use the BERTBASE model for our implementation mainly because our dataset contains 30,614 labeled queries. Since BERTBASE is less complex than BERTLARGE, it trains faster and uses fewer resources, while still performing well for classification tasks like detecting SQL injection attacks.

{ To adapt BERT for the task of detecting SQL injection attacks, we performed fine-tuning using a dataset labeled with both benign and malicious SQL queries. In this stage, the model was trained to classify each input as either safe or potentially harmful. Rather than retraining the entire network from scratch, we kept the core BERT layers intact and focused on updating the final classification layer. This approach allowed us to take advantage of BERT’s deep language understanding while tailoring the model specifically to the patterns found in SQL injection attempts.}

**4.4 Development Environment Overview**

**4.4.1 Programming language**

**Python**

Python is widely used in machine learning and deep learning, and for good reason. Its clean and simple syntax makes it easy to learn and work with, which is especially helpful when building and testing complex models.  A key strong point of Python is its huge selection of libraries, which include TensorFlow, PyTorch, Scikit-learn, and Keras. These libraries offer pre-written code and functionsthat save time and make the development process easier. Python also has a huge community of developers and researchers, which means there’s plenty of documentation, tutorials, and support available online. Python offers simplicity, flexibility, and strong tools, all of which render it an optimal language for machine learning and deep learning projects.

**4.4.2 Libraries**

**Numpy :** The fundamental library for numerical computation in Python, NumPy, handles arrays, mathematical operations, and prediction outputs such as probability arrays and class labels.  
  
**Pandas :** The pandas library is efficient for data manipulation and data analysis. we used it in this study to import the CSV dataset, remove duplicate rows, handle missing data, and cast data into appropriate formats for training models  
Chardet: Chardet library is a character encoding detection library, and we have used it in order to detect an appropriate encoding for our CSV file which could be read by pandas successfully.  
  
**Ktrain :** A high-level TensorFlow/Keras library that facilitates easy training, testing, and interpretation of deep learning models. Consistent with this, we used it to transform data into BERT-compatible format, train a text classification model, and evaluate the performance of the model.

**Matplotlib :** matplotlib is a graph and chart creation library. We leveraged it to plot class distributions, training loss and validation loss, training accuracy and validation accuracy, and the confusion matrix.  
**4.4.3 Development setup**

**4.4.3.1 Visual Studio Code**

Visual Studio Code (VS Code) is free, open-source software meant for coding, developed by Microsoft. Its support many programming languages such as Python, C++, JavaScript, etc. Equipped with syntax highlighting, code completion (IntelliSense), a debugging tools, and a customizable user interface, and an integrated terminal.

VS Code is very extensible and its capability can, therefore, be enhanced by any user with thousands of extensions that support frameworks, debugger.

It has support for Markdown to style text, allowing users to document alongside their code. With its lightweight might combined with powerful development features, Visual Studio Code is among the most popular editors for web developers, data scientists, DevOps engineers, and others.

**4.4.3.2 Jupyter Notebook**

The Jupyter Notebook is an open-source, browser-based interactive environment that offers users the ability to create and share documents that contain live code, equations, visualizations, and narrative text. Data scientists, machine learning specialists and scientific researchers are massive user bases of Jupyter Notebooks. They are used for easy experimentation, reproducibility, and collaborative purposes. A notebook is divided into cells, cells can be individually run. Having such a structure makes the Jupyter Notebook very handy for stepwise analysis, prototyping, and reporting. The format enhances understanding by combining code with explanations and visual output.

**4.4.3.3 Google colab**

Google Colaboratory (or Colab) is a free cloud service that allows users to write and execute Python in Jupyter Notebook environment without the necessity of local setup. Supporting machine learning, deep learning, data analysis, and research workflows. Colab offers access to computing resources , i.e., CPUs, GPUs, and TPUs, right within the browser. It is majorly used for prototyping, or collaborative development, built-in support for Google Drive and support for major Python libraries such as TensorFlow, PyTorch, and Pandas.

For our work, we used The NVIDIA Tesla T4. It is a powerful energy-efficient GPU designed for a wide range of AI workloads. It was built over the Turing architecture and includes Tensor Cores to accelerate deep learning compute and RT Cores for ray tracing. The T4 GPU comes with 16 GB of GDDR6 memory and supports mixed precision computing. It also delivers up to 260 TOPS (Tera Operations per Second) for INT8 inference. It is mostly used in cloud environments (like Google Colab) for efficiently accelerated machine learning workflows.

**4.5 BERT Code and Implimentation**

In this section, we present the code and detailed implementation of our SQL injection detection model, which is based on the BERT architecture.

**1 Reading and displaying the dataset**

****

This code detects the character encoding of the CSV file SQLiV3\_cleaned2.csv using the chardet library and then reads the file into a pandas DataFrame(df) with the correct encoding. This ensures the file is read without encoding errors, especially if it's not in UTF-8 format.

**2 Creating the training and test sets**

****

In this code, the text data (Sentence column) and the corresponding label data (Label column) are extracted from the DataFrame and converted Python lists. Afterward, the texts\_from\_array function of ktrain is used to do the preprocessing of these texts so that they can be used for training the model. With preprocess\_mode='bert', the word text is passed through a BERT tokenizer that lowercases, tokenizes into word pieces, pads, and then sequences exceeding 220 tokens are automatically shortened. It is intentionally split into training and validation (with 20% of samples used for validation) sets (val\_pct=0.2), the class\_names parameter specifies the two label classes allowing the function to handle the binary classification task of SQL injection detection Then, the function returns the preprocessed train and test sets (x\_train, y\_train, x\_test, y\_test), while the preproc object contains the tokenizer and configurations used during the preprocessing stage.

**2 Building the model**

****

This code initializes a text classification model using the BERT architecture with the preprocessed training data (x\_train, y\_train) and the associated preprocessing configuration preproc. It sets up the model ready for fine-tuning on our dataset.

**3 Fine-tuning the BERT model**

****

In this code, the ktrain.get\_learner function is used to create a learner object that encapsulates the model and the training process. The model argument receives the prebuilt or preloaded machine learning model that you want to train or fine-tune. The training data (train\_data) is passed as a tuple containing the input features (x\_train) and their corresponding labels (y\_train). Similarly, the validation data (val\_data) is provided as a tuple with the validation inputs (x\_test) and their labels (y\_test), which will be used to monitor the model's performance during training.

The batch\_size=32 parameter specifies that the training data should be divided into mini-batches of 32 samples each, allowing for more efficient gradient updates and memory management during training. The returned learner object now holds the model and data in a form that can be easily trained, validated, and later used for prediction or evaluation.

{need to be huamnized}

The model and the training procedure are combined into a learner object in this code using the ktrain.get\_learner function. The model argument receives the pre-trained BERTmodel that we want fine-tune The input features (x\_train) and their matching labels (y\_train) make up the training data (train\_data), which is supplied as a tuple. To track the model's performance during training, the validation data (val\_data) is also supplied as a tuple including the validation inputs (x\_test) and their labels (y\_test).

In order to facilitate more effective gradient updates and memory management during training, the batch\_size=32 parameter indicates that the training data should be split up into mini-batches of 32 samples each.{this too}



This method of fit\_onecycle is used to fine-tune the pretrained model. The learning rate (lr=2e-5) control the update of the model weights and has been made very low for stable fine-tuning. The number of epochs is 4, meaning that the training dataset is passed through the model four times.   
 The One Cycle learning rate policy applied by this method slowly increases the learning rate from a very low value to max value and then brings it down to its lowest rate, helping to improve training speed and overall model performance.