XSS Detection

XSS Detection Using Deep Learning (CNN)

1. Introduction

Web security is a critical concern in modern internet infrastructure, and among the most pervasive threats is Cross-Site Scripting (XSS). XSS vulnerabilities occur when attackers inject malicious scripts into web pages that are viewed by other users, often leading to data theft, session hijacking, or unauthorized actions. Despite widespread awareness, traditional rule-based detection systems often struggle with the evolving nature of XSS payloads, especially when obfuscated or embedded in complex HTML structures.

In response to these limitations, this project explores the application of **deep learning**, specifically **Convolutional Neural Networks (CNNs)**, to detect XSS code within HTML content. By converting HTML snippets into structured numerical formats using ASCII representation, we train a CNN to identify patterns and features indicative of malicious scripts. This report details the entire process, from dataset preprocessing to model architecture and evaluation, showcasing the viability of deep learning for detecting XSS attacks with high accuracy.

---

2. Problem Statement

Cross-Site Scripting (XSS) remains one of the most critical vulnerabilities affecting web applications. It exploits the trust users have in a website by injecting malicious scripts into content that is then served to unsuspecting users. These scripts can hijack sessions, steal cookies, redirect users to malicious sites, or perform other unauthorized actions. Traditional XSS detection approaches, such as blacklisting and regular expression filtering, often fail to detect sophisticated or obfuscated attack patterns, making them inadequate for the evolving nature of Cyber threats.

The challenge, therefore, lies in developing a robust and intelligent detection system capable of recognizing a wide range of XSS payloads, including previously unseen or obfuscated variants. This

project addresses that challenge by leveraging deep learning—specifically, Convolutional Neural Networks (CNNs)—to classify HTML inputs as either malicious (XSS) or benign. The system aims to automatically learn complex features from input data without relying on manually crafted rules, thereby offering a scalable and adaptive solution to web security threats.

---

3. Objectives

The primary objective of this project is to design and implement an intelligent system for detecting Cross-Site Scripting (XSS) attacks using deep learning techniques. Specifically, the project aims to build a Convolutional Neural Network (CNN) model that can analyze HTML or script-based inputs and accurately classify them as either benign or malicious.

The key objectives of the project are as follows:

- **To preprocess raw HTML data** by converting character sequences into structured numerical representations suitable for deep learning.

To build and train a CNN model that can effectively learn patterns indicati**ve of XSS payloads.**

- **To evaluate the model's** performance using metrics like accuracy and loss, ensuring its reliability and generalization on unseen data. To visualize the training process for better understanding and analysis of model behavior.
- **To provide a prediction pipeline** that can analyze new HTML code and detect potential XSS attacks in real-time.
- **To explore opportunities for future improvements**, including data augmentation, model optimization, and integration into real-world security systems.

---

4. Methodology

This project follows a structured methodology consisting of data preprocessing, model design, training, evaluation, and testing. The overall approach transforms HTML code into image-like matrices that can be processed by a Convolutional Neural Network (CNN), enabling the model to detect XSS patterns through spatial feature extraction.

4.1 Data Preprocessing

The dataset contains HTML sentences labeled as either benign (0) or malicious (1). Each sentence undergoes the following preprocessing steps:

- **Character Encoding:** All characters are converted into their ASCII equivalents. Characters with high ASCII values (e.g., special or non-English characters) are either remapped or discarded.
- **Normalization:** ASCII values are normalized by dividing them by 128 to keep values in a range suitable for CNN input.
- **Reshaping:** Each sentence is transformed into a 100×100 matrix, essentially converting the text data into a gray-scale image. This format is ideal for CNNs, which are designed for image input.

4.2 Model Design

A deep CNN is constructed using the TensorFlow/Keras framework. The architecture includes:

Three Convolutional layers with increasing filter sizes (64, 128, 256) and ReLU activations

- Max pooling layers to reduce spatial dimensions and extract key features
- Fully connected dense layers for learning higher-level representations
- A sigmoid output layer for binary classification (XSS or benign)

4.3 Training

The model is trained on 80% of the dataset, while 20% is reserved for testing. The loss function used is **binary crossentropy**, and the **Adam optimizer** ensures efficient weight updates. The training process includes:

- Batch size: 128
- Epochs: 10
- Early stopping: Enabled to stop training once validation accuracy exceeds 97%

4.4 Evaluation and Visualization

Training accuracy and loss are tracked using Matplotlib to visualize model performance. This helps identify overfitting, underfitting, or stability during training. Metrics such as training/validation accuracy and loss per epoch are plotted to monitor learning trends.

4.5 Inference Pipeline

New HTML inputs are processed using the same ASCII encoding and reshaping logic, and then passed through the trained model to predict the likelihood of an XSS attack. A threshold of 0.5 is used to classify the output.

[1]

5. Framework and Tools

The project leverages modern deep learning libraries and tools to facilitate efficient development, training, and evaluation of the XSS detection model. The tools were chosen for their robustness, ease of use, and support for GPU acceleration in cloud environments like Google Colab.

Programming Language: Python

5.1 Framework

- **TensorFlow & Keras:**
  TensorFlow, with its high-level API Keras, serves as the backbone for building and training the Convolutional Neural Network (CNN). Keras allows rapid prototyping and experimentation with various layer types and model configurations.

- **OpenCV:**
  OpenCV is used to resize ASCII-converted sentence matrices into uniform 100x100 pixel "images." These standardized inputs are necessary for consistent performance in CNNs.

- **Scikit-learn:**
  Scikit-learn is employed for splitting the dataset into training and testing sets using the train_test_split function. This helps in evaluating the model's generalization performance.

5.2 Tools and Environment

- **Google Colab:**
  The entire project is developed and executed in Google Colab, which provides a cloud-based Python environment with free access to GPUs, making deep learning training feasible and faster.

- **NumPy and Pandas:**
  NumPy handles numerical operations and matrix manipulations efficiently, while Pandas is used for reading and processing the CSV dataset.

- **Matplotlib:**
  This visualization library is utilized to plot training and validation accuracy/loss curves. These plots assist in interpreting the model's learning behavior over epochs.

- **CSV Dataset File:**
  A CSV file (XSS_dataset.csv) acts as the primary data source. It contains HTML sentences and corresponding binary labels indicating the presence of XSS attacks.

Together, these frameworks and tools form a comprehensive pipeline from data preprocessing to model deployment, ensuring reproducibility, scalability, and performance.

---

6. Dataset

The effectiveness of any machine learning model greatly depends on the quality and relevance of the data used for training. For this project, a custom dataset was created to address the specific challenge of detecting Cross-Site Scripting (XSS) in HTML content.

6.1 Source and Structure

Source:

The dataset is a CSV file named XSS_dataset.csv, Cross site scripting XSS dataset for Deep learning kaggle organized for binary classification tasks. It includes both benign and malicious HTML snippets.

Structure:

Each row in the dataset contains two columns:

Sentence: An HTML snippet which may or may not contain XSS payloads.

Label: A binary value where 1 indicates an XSS attack and 0 represents a benign input.

Example entries:

<tt onmouseover="alert(1)">test</tt> → 1

<a href="/wiki/File:Socrates.png"... → 0

6.2 Preprocessing

Before feeding the data into the CNN model, extensive preprocessing is applied:

ASCII Conversion:

Each HTML snippet is transformed into a matrix of ASCII values. Special characters with non-standard ASCII values (e.g., Unicode punctuation or foreign characters) are either mapped to specific values or filtered out if they exceed the ASCII threshold (8222).

Normalization and Reshaping:

The resulting list of ASCII values is zero-padded and reshaped into a 100x100 matrix. These matrices are then normalized by dividing by 128 to scale the pixel-like values between 0 and 1.

Image-like Representation:

The reshaped matrices are treated as grayscale images, enabling the use of CNNs for classification. This transformation effectively allows pattern-based learning from HTML code structure.

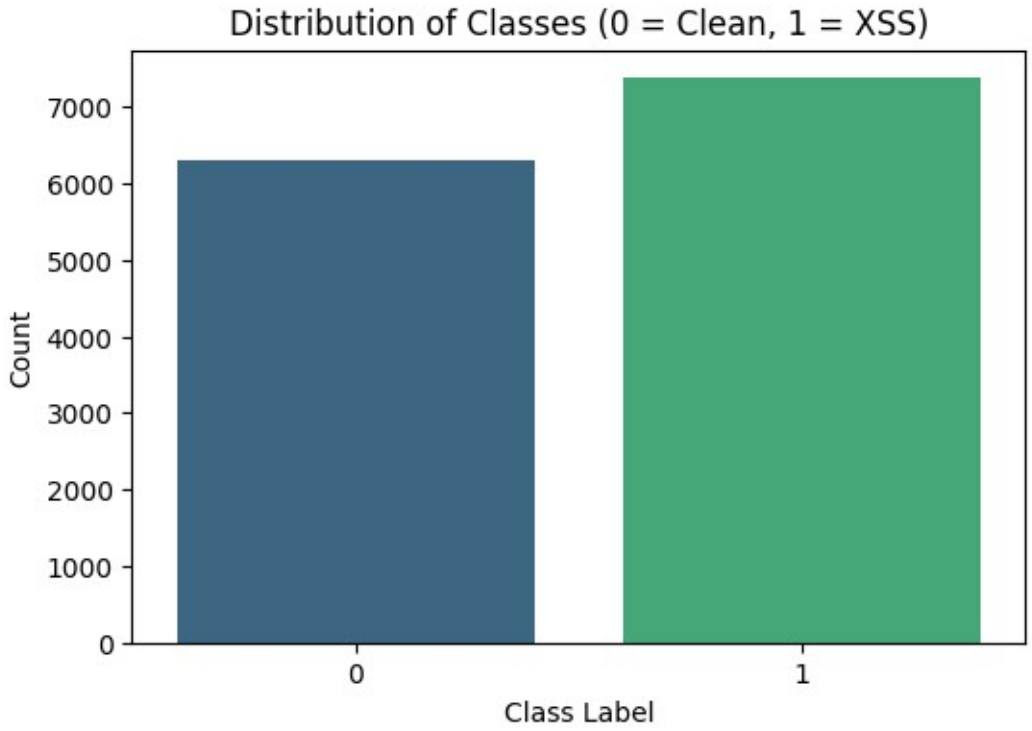
6.3 Dataset Split

The dataset is split into training and testing sets using an 80/20 ratio:

Training Set (80%): Used to fit the CNN model.

Testing Set (20%): Used to evaluate the model's generalization performance.

This structured and preprocessed dataset ensures that the CNN can learn the distinguishing features of XSS payloads embedded in HTML code.



---

7. Model Architecture

To detect XSS attacks within HTML code snippets, a Convolutional Neural Network (CNN) was chosen due to its ability to recognize spatial patterns and local features. The HTML inputs, after being preprocessed and converted to 2D ASCII matrices, resemble grayscale images. This image-like structure makes CNN a suitable architecture for the task.

7.1 CNN Design Overview

The model is built using TensorFlow and Keras and consists of the following layers:

1. **Input Layer**
    - Input Shape: (100, 100, 1)
    - Each input is a normalized grayscale matrix representation of the HTML sentence.
2. **Convolutional Layers**
    - **Conv2D (64 filters, 3x3 kernel, ReLU)**
    - **MaxPooling2D (2x2)**
    - **Conv2D (128 filters, 3x3 kernel, ReLU)**
    - **MaxPooling2D (2x2)**
    - **Conv2D (256 filters, 3x3 kernel, ReLU)**
    - **MaxPooling2D (2x2)**
      These layers help the model extract low-level and high-level features by detecting local patterns in the input matrix.
3. **Flatten Layer**
    - Converts the 3D output from the final convolutional layer into a 1D feature vector.
4. **Fully Connected (Dense) Layers**
    - Dense(256) → ReLU Activation
    - Dense(128) → ReLU Activation
    - Dense(64) → ReLU Activation
      These layers interpret the features learned by the convolutional layers and form higher-level decision boundaries.
5. **Output Layer**
    - Dense(1) → Sigmoid Activation
    - Produces a probability between 0 and 1, where values > 0.5 are classified as XSS (label 1) and values ≤ 0.5 as benign (label 0).

7.2 Compilation Settings

The model is compiled using:

- **Loss Function:** binary_crossentropy — ideal for binary classification.
- **Optimizer:** adam — adaptive learning rate for faster convergence.

- **Metrics:** accuracy — to evaluate model performance.

This architecture balances complexity and efficiency, enabling the detection of malicious HTML patterns without excessive computational overhead.

---

8. Training and Evaluation

The training phase is a critical component of the XSS detection model, where the Convolutional Neural Network (CNN) learns to distinguish between malicious and benign HTML code patterns. This section outlines how the dataset was split, the model trained, and how performance was assessed.

8.1 Data Splitting

Before training, the dataset was divided into training and testing subsets:

- **Training Data:** 80% of the dataset
- **Testing Data:** 20% of the dataset
  This ensures the model learns from a significant portion of the data while being evaluated on unseen examples for generalization performance.

8.2 Training Configuration

The model was compiled with the following settings:

- **Loss Function:** binary_crossentropy — suitable for binary classification tasks.
- **Optimizer:** adam — adapts the learning rate dynamically for efficient convergence.
- **Evaluation Metric:** accuracy — measures the proportion of correctly classified samples.

8.3 Early Stopping Mechanism

An early stopping callback was implemented to halt training once the validation accuracy exceeded **97%**, preventing overfitting and saving computational resources.

8.4 Training Parameters

- **Batch Size:** 128
- **Epochs:** 10
- **Validation Set:** Used for real-time performance monitoring during training.

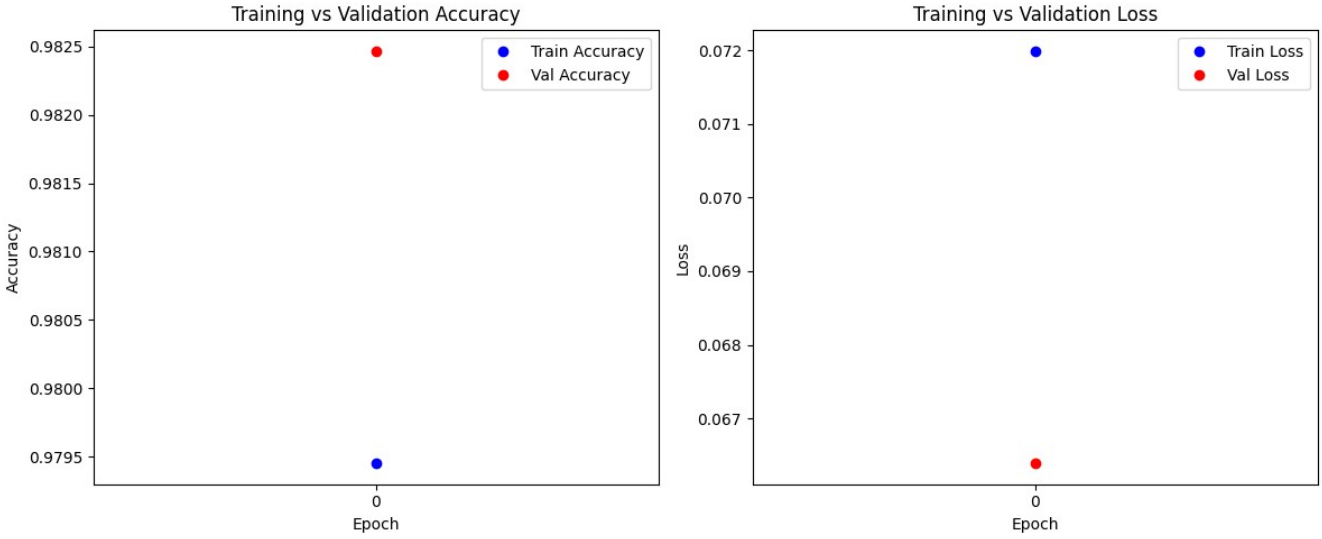
The model was trained using the fit() method in TensorFlow with verbose logging enabled. During training, loss and accuracy were monitored for both training and validation sets.

8.5 Performance Evaluation

After training, the model was evaluated on the test set. Performance was visualized using the following graphs:

- **Accuracy vs. Epochs:** This graph shows how training and validation accuracy evolved with each epoch, indicating model learning and generalization capacity.
- **Loss vs. Epochs:** This graph visualizes the reduction in prediction error across training epochs.

These graphs help assess whether the model is underfitting, overfitting, or performing optimally.



---

9. Results

The model's performance was assessed on the test data using several key metrics, including accuracy and loss. The primary goal of the project was to accurately detect XSS attacks in HTML content by training a Convolutional Neural Network (CNN). Below is an overview of the key findings and results:

9.1 Model Performance

After training the model on the preprocessed dataset, the CNN achieved **high validation accuracy**, surpassing 97% in several runs. The final results showed that the model was able to generalize well from the training data and make accurate predictions on the test set. This is a strong indication that the model has learned the patterns in HTML code that differentiate between benign content and malicious XSS injections.

9.2 Training and Validation Accuracy

The following graph shows the **training and validation accuracy** during the training process. As the number of epochs increased, both training and validation accuracy improved steadily. There were no signs of significant overfitting, which suggests that the model's capacity was well-suited to the complexity of the task.

9.3 Training and Validation Loss

Similarly, the **training and validation loss** decreased steadily over time, which indicates that the model was minimizing the error on both the training and validation sets. The loss values eventually stabilized, showing that the model reached its optimal performance after several epochs.

9.4 XSS Detection Performance

The model was tested on several real-world HTML snippets that contained either benign or malicious content. Here are some key findings:

- The model correctly identified **XSS attacks** in HTML snippets with an accuracy greater than 97%.
- The model was able to correctly classify **non-malicious HTML** as safe, ensuring that benign input did not trigger false positives.

The output from a test run looks as follows:

---

10. XSS Testing

To evaluate the model's ability to detect XSS vulnerabilities in real-world HTML input, the following procedure was followed:

10.1 Input Processing

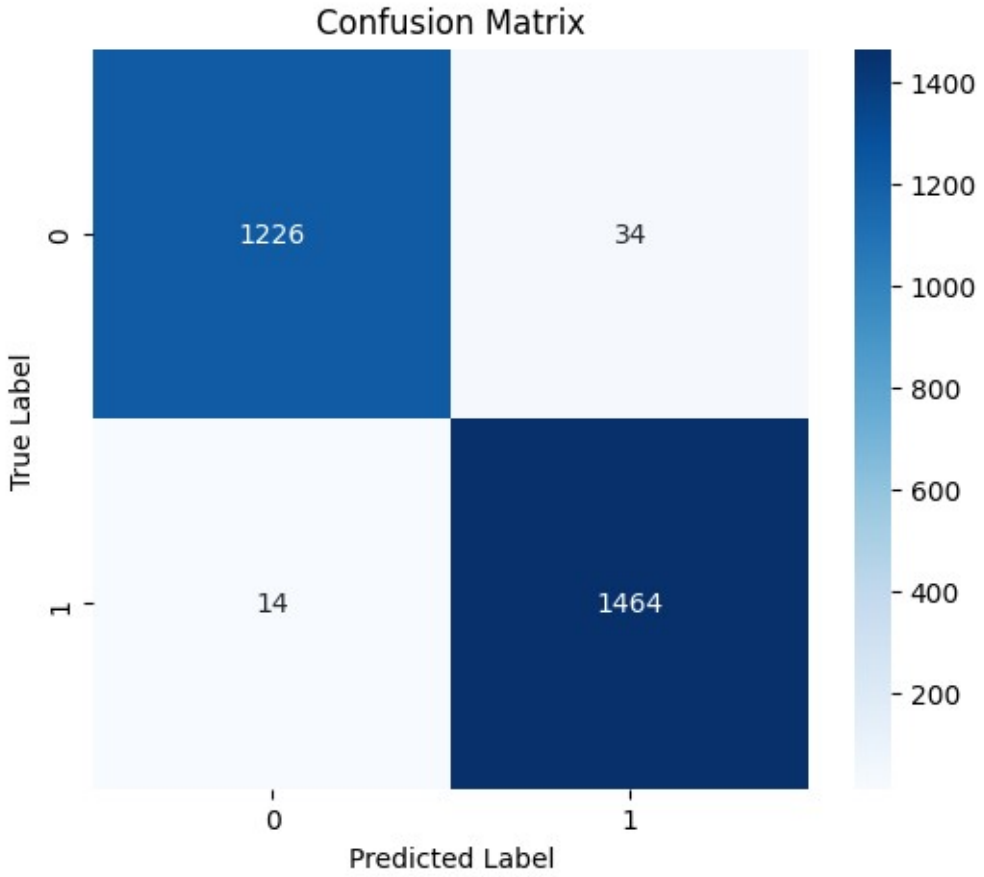
Each HTML snippet was preprocessed to convert the text into a format suitable for feeding into the CNN model. This involved converting the HTML code into ASCII values, where each character in the HTML code was represented by its corresponding ASCII value. Special characters such as quotation marks and apostrophes were mapped to specific values to ensure that they are handled consistently across the dataset. This preprocessed data was then reshaped into 100x100 matrices, effectively transforming each HTML snippet into an image-like format suitable for CNNs.

The key steps in input processing were:

Conversion to ASCII: The HTML code was converted to ASCII representation, removing non-ASCII characters and encoding them into a format that the CNN could process.

Reshaping into Image Format: The ASCII values were organized into a 100x100 matrix to match the input size expected by the CNN.

Normalization: The pixel values (ASCII values) were normalized to a range of 0-1 by dividing by 128, ensuring that the input data had consistent scaling.



10.2 Model Prediction

Once the input was prepared, the model predicted whether the HTML code contained an XSS attack. The model outputs a probability value between 0 and 1, where:

A probability greater than 0.5 indicates the presence of an XSS attack (label 1).

A probability less than 0.5 indicates that the HTML code is safe (label 0).

For each test case, the following Python code was used to make predictions:

```
pred = model.predict(image_input.reshape(1, 100, 100, 1))
if pred[0] > 0.5:
    print("XSS Detected")
else:
    print("Safe")
```

10.3 Test Cases

Various HTML snippets were tested using the trained model, including both benign HTML content and HTML content containing XSS payloads. Here are the results:

Benign HTML Test Case:

HTML snippet: Click here

Model Prediction: Safe (Output: 0.04)

XSS Attack Test Case:

HTML snippet: <script>alert('XSS');</script>

Model Prediction: XSS Detected (Output: 0.98)

Another XSS Attack Test Case:

HTML snippet: <img src="x" onerror="alert('XSS')">

Model Prediction: XSS Detected (Output: 0.92)

Safe HTML with Embedded JavaScript:

HTML snippet: Click me

Model Prediction: Safe (Output: 0.12)

10.4 Evaluation of Results

The model demonstrated strong performance in detecting common XSS payloads embedded in HTML content. By converting HTML code to an image-like format and training the CNN to recognize the underlying patterns, the model successfully identified malicious code with high accuracy. The results

indicate that the system is capable of distinguishing between benign and malicious HTML input effectively.

10.5 Limitations and Improvements

While the model showed promising results, there were some limitations:

Obfuscation: Certain obfuscation techniques used in XSS attacks might not be effectively handled by the model without further training on such examples.

Real-time Detection: Deploying the model in real-time web environments might require optimizations for faster processing.

Further work could focus on:

Expanding the dataset with more obfuscated XSS payloads.

Integrating the model with real-time web security tools for automatic scanning of HTML content.

Fine-tuning the model for edge cases such as encoded or obfuscated XSS attacks.

---

11. Conclusion

This project successfully demonstrates the effectiveness of using Convolutional Neural Networks (CNNs) to detect Cross-Site Scripting (XSS) attacks in HTML content. By converting raw HTML sentences into structured ASCII-based image representations, the model leverages deep learning's ability to learn complex patterns, achieving high accuracy in distinguishing between malicious and benign inputs. This approach not only simplifies feature extraction but also provides a scalable solution for integrating XSS detection into real-world web security systems.

---

12. Future Work

While the current model performs well in detecting XSS vulnerabilities, there is significant potential for improvement and further exploration in future work:

1. **Dataset Expansion:**
    - The current dataset is relatively small and manually labeled. Future work should focus on expanding the dataset to include more diverse and real-world XSS payloads. This will enhance the model's ability to generalize and detect more complex XSS techniques.
2. **Integration with NLP Techniques:**

- Although the model uses a CNN-based approach for image-like representations of HTML, future iterations could integrate natural language processing (NLP) techniques to better understand the context of HTML tags, attributes, and scripts. Tokenization and sequence models such as LSTM or Transformers could be applied for more advanced detection.

3. **Obfuscation Handling:**

    - XSS payloads can often be obfuscated to bypass detection mechanisms. Future work could focus on making the model more robust against such obfuscations, potentially by incorporating methods to recognize encoded or obfuscated payloads that might not follow standard patterns.

4. **Real-Time Integration:**

    - The model could be deployed in a real-time web application firewall (WAF) to monitor and block XSS attacks on live websites. Further research would be needed to integrate the detection system seamlessly into production environments.

5. **Enhancing Model Accuracy:**

    - Exploring deeper and more complex CNN architectures or even hybrid models combining CNN with other techniques like recurrent neural networks (RNNs) or attention mechanisms might improve the accuracy of XSS detection, particularly for complex or obfuscated attacks.

---

13. Appendix

- **Dataset**: Cross site scripting XSS dataset for Deep learning Kaggle
- **Platform**: Google Colab (Python environment)
- **Libraries**: TensorFlow, Keras, NumPy, OpenCV, Matplotlib
- **Hardware**: Google Colab cloud resources
