SMARTX: A Scalable Machine Learning-Based Framework for Real-Time Detection of XSS Attacks in Web Applications

Somchart Fugkeaw, Thanapat Thaipakdee, Chawanakon Promsila, Sirapitch Boonyasampan, Ittiwat Nimitliupanit
Sirindhorn International Institute of Technology, Thammasat University, Thailand
somchart@siit.tu.ac.th , name22077@gmail.com, chawanakon.pro.work@gmail.com, sirapitchboonyasampan@gmail.com, ittiwatnimitfw@gmail.com

***Abstract*— Cross-Site Scripting (XSS) is a pervasive web application vulnerability that allows attackers to inject malicious scripts into web pages, compromising the security of other users. Designing a cloud-based XSS detection system that effectively handles large-scale web applications poses challenges in both accuracy and operational efficiency. Most existing solutions primarily focus on detection accuracy, often overlooking critical aspects such as visualization and attack detection efficiency for high volumes of transactions for multiple web applications. In this paper, we present a SMARTX model providing a solution for detecting XSS attacks using a machine learning-based approach, leveraging a Multi-Layer Perceptron (MLP) classifier. To enhance detection speed and scalability, we employ multiprocessing techniques for real-time threat analysis. Additionally, we introduce a graph-based model, implemented using the Neo4j database, to efficiently store and visualize detection results across multiple cloud-based web applications. This model establishes correlations between hosts, clients, and detection outcomes, enabling actionable insights for administrators. Our architecture integrates both client-side and server-side detection mechanisms, providing robust protection against various XSS injection vectors. Experimental results demonstrate that SMARTX model achieves high detection accuracy while optimizing processing times, making it highly suitable for real-world deployment in dynamic web environments.**

I. Introduction

Cross-Site Scripting (XSS) is one of the most common and severe web security vulnerabilities, frequently exploited by attackers to inject malicious scripts into web applications. These scripts are executed in the browsers of unsuspecting users, potentially leading to significant consequences such as data theft, session hijacking, redirection to malicious sites, and unauthorized actions performed on behalf of users. XSS attacks generally occur due to improper validation of user inputs in web applications. XSS attacks are generally classified into three categories: Stored XSS, where malicious scripts are permanently stored on the target server; Reflected XSS, where the attack is reflected off a web server and DOM-based XSS, where the client-side code is manipulated, exploiting vulnerabilities in the Document Object Model (DOM) without interaction with the server [1].

Traditional signature-based detection systems are often ineffective against sophisticated XSS attacks. Attackers can obfuscate their payloads, allowing malicious scripts to bypass static detection techniques. To address this challenge, machine learning models offer a more dynamic and adaptive solution. By learning patterns from a diverse set of XSS attack vectors, machine learning models [2, 3, 4, 5] can detect previously unseen or obfuscated attack attempts. In addition, Designing and developing a scalable cloud-based XSS detection system presents significant challenges in both accuracy and operational efficiency. Existing solutions tend to prioritize detection accuracy, but often overlook important aspects such as high-throughput performance and the ability to efficiently handle multiple web applications deployed in an outsourcing environment. These gaps hinder the deployment of such systems in real-world, high-traffic environments where operational management is as critical as detection precision.

This paper introduces an advanced XSS detection system that leverages a Multi-Layer Perceptron (MLP) classifier for real-time detection. The MLP model is trained on datasets of both benign and malicious inputs, allowing it to distinguish between safe and harmful traffic with high accuracy. To ensure scalability and real-time performance, the system also integrates multiprocessing allowing the system to handle large volumes of requests efficiently [4].

Upon detecting an XSS attack, the system records essential information such as the client IP address, the timestamp of the attack, and the malicious script in a Neo4j graph database. This database is designed to store and represent the relationships between various elements involved in the attack. Specifically, the system logs the attack using three primary nodes: (1) *Host* representing the server where the XSS attack was detected. This node helps in identifying the targeted web server or application, (2) Client referring to the entity or user initiating the request. Tracking the client allows deeper investigation into the source of the attack, potentially uncovering patterns in malicious behavior, (3) *Detection* holding detailed information about the XSS attempt, including the type of payload injected, the timestamp when the attack occurred, and whether or not the attack was successful [5].

Storing this information in a graph database provides a powerful way to visualize and analyze the relationships between attack entities. The graph structure allows security teams to see how an attack propagates through the system and helps identify connections between multiple attacks. This approach enables deeper insights into recurring attack patterns, as well as relationships between different clients and hosts [6].

II. Related Work

This section discusses XSS detection and prevention approaches ranging from traditional rule-based systems to advanced machine learning models that adapt to new and obfuscated attack patterns.

In [7], Murugan et al. introduced a rule-based approach using predefined patterns to detect XSS attacks. While effective for known attack vectors, these methods have shown limitations in adapting to novel or obfuscated attacks. Attackers can easily bypass these systems by altering their payloads to avoid detection.

In [5], Iyer et al. proposed a machine learning-based system using Support Vector Machines (SVMs) and Random Forests to detect XSS attacks. These models were trained on datasets of both benign and malicious inputs, allowing them to recognize attack patterns that traditional methods often miss. However, scalability and the challenge of reducing false positives remain critical hurdles for these systems.

In [6], Ali et al. presented an architecture leveraging multiprocessing to improve the speed and efficiency of XSS detection in high-traffic environments. By processing multiple requests in parallel, the system ensures timely detection without sacrificing accuracy. This approach is particularly useful in large-scale web applications where the volume of traffic is high, and real-time detection is crucial.

In [9], Zheng et al. highlighted the importance of using graph-based databases for analyzing relationships between different components involved in XSS attacks. By storing attack data in a Neo4j graph database, relationships between attack vectors, affected hosts, and clients could be mapped out, providing deeper insights into the nature of XSS attacks. This method not only supports threat analysis but also helps in detecting recurring patterns of attacks, which are often missed by traditional databases. However, the authors did not tackle the detection performance when the system deals with a high volume of transactions for multiple web-applications.

Recently, several approaches [10, 11, 12, 13, 14, 15, 16, 17] applied machine learning techniques to adaptively detect the XSS attacks. For example, Fadli et al. [10] presented a hybrid XSS detection framework integrating both machine learning models and signature-based methods. While machine learning models detected novel attacks, the signature-based component ensured quick identification of known attack vectors.

In [12], Suzuki et al. proposed the unsupervised learning approach using clustering techniques to separate benign from malicious scripts, the model reduced false positives while maintaining a high detection rate. This approach, while promising, lacked the ability to adapt to rapidly changing attack vectors, highlighting the need for continuous learning systems in security applications.

In [13], Gil et al. leveraged machine learning (ML) and deep learning (DL) techniques for improving XSS detection accuracy. Algorithms like Support Vector Machines (SVM) and Convolutional Neural Networks (CNN) effectively detect XSS attacks when combined with robust preprocessing methods. However, challenges such as scalability and handling novel attacks remain.

In [15], Patterson et al. introduced a model using the paths-attention method to detect reflective XSS vulnerabilities. It improves detection by converting code into abstract syntax trees and applying attention mechanisms to focus on key paths.

In [16], the authors present a hybrid machine learning approach for detecting XSS attacks. Among the models tested, the hybrid ensemble of decision trees achieved the best performance with a detection accuracy of 99.8% and a very short detection time of 103.1 microseconds. The approach was validated using the XSS Attacks-2019 dataset, showing that hybrid learning is highly effective for XSS detection.

In [17], Pillai et al. propose a Praise-Worthy Authentication technique to detect phishing websites using hyperlink properties, and a Maximal-Munch Algorithm-based ANN to prevent XSS attacks by analyzing URL parameters and text patterns. A Torrent Deep network with a weight-bolster algorithm is also used to detect SQL injection attempts, preventing data leaks and website paralysis.

Nevertheless, real-time scalability and the efficiency of incident monitoring for supporting multiple web applications are not addressed by these schemes.

III. Our proposed System

A. System Overview

Our system is designed to protect against Cross-Site Scripting attacks using a deep learning model hosted in the cloud for scalability and real-time performance. Figure 1 presents our proposed XSS detection system model.

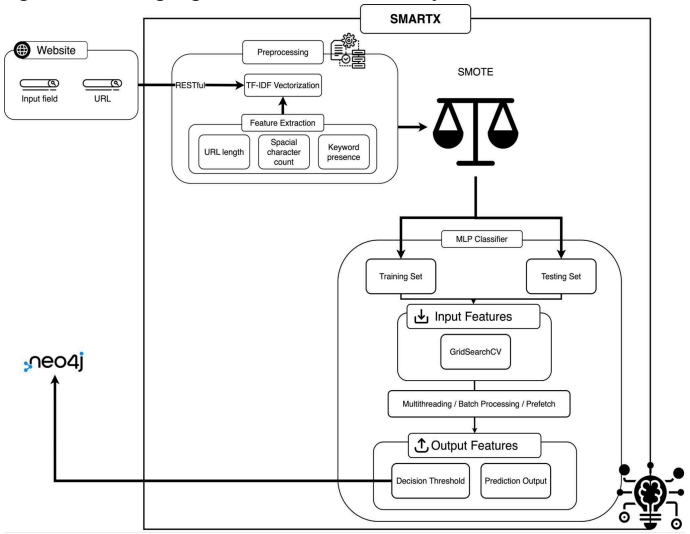

Fig. 1. Our proposed SMARTX System Model.

The system model consists of three major parts as follows.

1. Websites or web applications where the raw input data (URLs) is submitted by users. The input URL data flows into the system via a RESTful API, initiating the detection process. This component is crucial as it acts as the gateway for receiving real-time input from users. The input fields on the website accept user-generated data, which may include potential XSS attacks. These inputs are sent to the detection model for further analysis.

2. SMARTX Detection Model is responsible for analyzing the URL inputs and determining if they contain potential XSS threats. In SMARTX, we applied SMOTE (Synthetic Minority Oversampling Technique) to handle the balancing of the dataset to improve the accuracy of XSS detection. We also devise a detection algorithm based on MLP classifier with the utilization of multithread processing.

3. Neo4j Graph Database stores and visualizes the results of the detection process are logged. Once the detection model makes a prediction, key details such as Client ID, Host IP, timestamp, and whether the URL is an XSS attack or benign are recorded in the Neo4j database. This allows for easy querying, analysis, and visualization of attack patterns over time. The graph structure of Neo4j enables the system to establish relationships between different entities, such as clients and hosts, making it an effective tool for tracking recurring vulnerabilities.

B. Deep Learning Model

Our SMARTX Detection Model combines multiple processes to enhance both accuracy and efficiency. The model consists of several stages, each of which contributes to the overall performance. Below is an explanation of how each component in the flow contributes to these improvements:

1.Preprocessing
In this stage, the raw input data (URLs) is prepared for analysis. The TF-IDF Vectorization technique transforms the URLs into numerical vectors that represent the relevance of each character and word in the input. This helps the model better understand the structure and content of URLs, allowing it to differentiate between normal inputs and potential XSS attacks. The addition of custom features such as URL length, special character count (like <, >, &, etc.), and keyword presence (like script, alert, img, etc.) further enhances the model's ability to detect XSS attacks. These features allow the model to capture patterns and characteristics commonly associated with XSS attacks, improving its detection accuracy.

2. SMOTE is applied to oversample the minority class (XSS attack data). This ensures that the model is not biased toward the majority (benign URLs) and improves its ability to detect XSS attacks in real-time. SMOTE plays a critical role in improving the accuracy of the model, particularly when dealing with imbalanced datasets.

3. Model Training with MLP Classifier
The Multi-Layer Perceptron (MLP) Classifier is the core learning algorithm used in the model. It is trained on both benign and XSS-labeled URLs, allowing it to recognize patterns that differentiate between safe and harmful input. The MLP classifier has been optimized using the following techniques to improve both accuracy and efficiency:

- GridSearchCV for Hyperparameter Tuning: This process ensures that the optimal parameters (such as the number of hidden layers and learning rate) are selected for the model. GridSearchCV systematically searches through a range of values to identify the most effective configuration for accuracy.
- Multithreading/Batch Processing: To improve the efficiency of the system, the model uses multithreading and batch processing techniques. These enable the system to process multiple inputs simultaneously, reducing latency and ensuring that even large datasets are handled efficiently without affecting real-time performance.

4. Parallel Batch Processing
To handle the high volume of incoming URL requests efficiently, the system uses Parallel Batch Processing. This technique allows the model to process batches of URLs concurrently, which improves the throughput of the system, especially in high-traffic environments. By prefetching batches of URLs and applying multithreading, the system is able to manage larger datasets more effectively, contributing to faster detection times without compromising accuracy.

5. Model Output and Decision Threshold
The model produces a Probability Score for each input URL, indicating the likelihood of it being an XSS attack. A Decision Threshold is then applied to classify the URL as either malicious (XSS) or benign based on this score. The threshold (set to 0.95 in this system) is fine-tuned to balance the trade-off between precision and recall, ensuring that the model is sensitive enough to detect XSS attacks while minimizing false positives.

6. Output to Neo4j for Analysis

Once a URL has been classified by the model, the results are logged into the Neo4j graph database. This component does not directly improve detection accuracy but enhances the system's overall efficiency by providing an intuitive way to track, analyze, and visualize the relationships between clients, hosts, and detected XSS attacks. Storing the detection data in Neo4j also allows for easy querying and identification of recurring vulnerabilities, which supports long-term security improvements.

C. Neo4j Graph Database

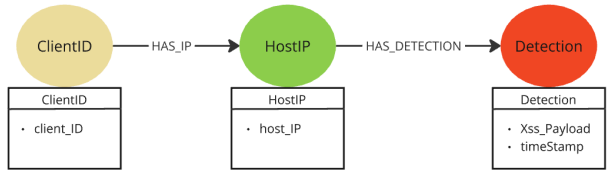

Fig. 2.Graph-based Model XSS Attack Detection.

This model features three interconnected entities as follows: *(1) ClientID* node represents the unique identifier for each client in the system. It links to one or more host IP addresses through the "HAS_IP" relationship, indicating the network nodes associated with client activities. *(2) HostIP* node denotes the IP address of a host, which is a server or a network interface where network activities are monitored. This entity is pivotal for localizing the network context of XSS detections. It is linked to the Detection entity through the "HAS_DETECTION" relationship, mapping where the XSS attack was detected within the network. *(3) Detection* node represents the XSS attack detection event, characterized by the Xss_Payload, which is the malicious script attempted to be injected, and timeStamp, marking the precise moment of detection. This entity is crucial for the timely and accurate reporting of security incidents. The model employs directed relationships to efficiently trace XSS attack incidents from the initiating client through the implicated host IPs to the resulting detections, thereby structuring the security analysis process.

IV. Methodology

1. Model Training Process: The first step in the XSS detection system is training the machine learning model. This involves loading, labeling, and preprocessing datasets of benign and malicious URLs. The system uses TF-IDF vectorization and custom features like URL length, special character count, and XSS keyword presence to create a comprehensive feature set.

The data is split into training and testing sets, with SMOTE applied to address class imbalance. The MLPClassifier is trained, and hyperparameters are optimized using GridSearchCV. The final model is saved for use in future detection tasks. Algorithm 1 presents the model training procedure.

Algorithm 1 Model Training

Input: `normal_path, xss_path`
function LOAD_DATA(normal_path, xss_path)
    Read payloads from `normal_path` and `xss_path`
    Label `normal_payloads` as 0 and `xss_payloads` as 1
    Return DataFrame with payloads and labels
end function
`data = LOAD_DATA('Train_NonXSS.txt', 'Train_XSS.txt')`
TF-IDF Vectorization:
Create a `TfidfVectorizer` instance with a limit of 5000 features,
`vectorizer = TfidfVectorizer(max_features=5000)`
Apply the vectorizer to transform the payload text data into numerical features,
`X_tfidf = vectorizer.fit_transform(data['payload']).toarray()`
Custom Feature Calculation:
Calculate custom features such as `payload_length`, `special_char_count`, and `keyword_presence` and add them to the data
Combine the `X_tfidf` array with the custom features to create `X_custom`
`y = data['label']`
Train-Test Split:
Split the data into training and testing sets,
`X_train, X_test, y_train, y_test = train_test_split(X_custom, y, test_size=0.3, random_state=42)`
Apply SMOTE:
Apply Synthetic Minority Oversampling Technique (SMOTE) to balance the classes in the training set,
`X_train_resampled, y_train_resampled = SMOTE().fit_resample(X_train, y_train)`
Hyperparameter Tuning:
Define the hyperparameter grid for `MLPClassifier`, including `hidden_layer_sizes`, `alpha`, and `max_iter`,
`param_grid = {'hidden_layer_sizes': [(100,), (128,)], 'alpha': [0.001], 'max_iter': [300]}`
Perform grid search with 2-fold cross-validation for hyperparameter tuning using `GridSearchCV`,
`grid_search = GridSearchCV(MLPClassifier(random_state=42, early_stopping=True), param_grid, cv=2, scoring='accuracy', verbose=10)`
Train the best model found by `GridSearchCV` on the resampled data
Prediction:
Predict probabilities for the test data,
`y_pred_proba = best_mlp_clf.predict_proba(X_test)[:, 1]`
Apply a custom decision threshold of 0.4 to classify predictions,
`y_pred = (y_pred_proba >= 0.4).astype(int)`
Model Saving:
Save the trained model and the vectorizer for future use,
`joblib.dump(best_mlp_clf, 'mlpc_xss_model_with_custom_features.pkl')`
`joblib.dump(vectorizer, 'tfidf_vectorizer_with_custom_features.pkl')`

2. XSS Detection Process Using RESTful Web Request:

After training, the model can detect XSS attacks in real-time using a RESTful API. When a web request is received via the REST API, the input field is preprocessed through TF-IDF vectorization and feature extraction, including: Input Length. Special Character Count and Keyword Presence. These features are combined and passed to the machine learning model, which outputs a probability score. If the score exceeds

a predefined threshold, the URL is flagged as a potential XSS attack. Algorithm 2 describes the detection procedure.

Algorithm 2 Detection with RESTful Request

```
Input: webRequest
function DETECT_XSS(webRequest)
    Load trained model and vectorizer
    Preprocess webRequest using:
    TF-IDF vectorization
    Calculate additional features:
    url_length
    special_char_count
    keyword_presence
    Combine TF-IDF and additional features
    Predict if webRequest is XSS or benign
    if prediction ≥ threshold then
        Classify as XSS
        Call Log_Detection_to_Neo4j
    else
        Classify as benign
    end if
    Return detection result
end function
```

If an XSS attack is detected, the system calls the detection logging module to log the details of the attack in a Neo4j.

3. Detection Logging : The system logs detected XSS attacks into a Neo4j graph database, which provides a flexible and efficient way to manage relationships between entities involved in the detection process. Algorithm 3 shows how the Neo4J adaptively update when the detection is done.

Algorithm 3 Log Detection to Neo4J

```
Input: detectionInfo (clientID, hostIP, timestamp, payload)
function LOG_DETECTION_TO_NEO4J(detectionInfo)
    Connect to Neo4j database
    Begin session
    if not CLIENT_NODE_EXISTS(clientID) then
        Call CREATE_CLIENT_NODE(clientID)
    end if
    if not HOST_NODE_EXISTS(hostIP) then
        Call CREATE_HOST_NODE(hostIP)
    end if
    Create Detection_Node with {timestamp, payload}
    Create relationships: {Client -> Host, Host -> Detection}
    Commit transaction to Neo4j
end function
function CREATE_CLIENT_NODE(clientID)
    if not CLIENTID_EXISTS(clientID) then
        Create Client node with clientID
    end if
end function
function CREATE_HOST_NODE(hostIP)
    if not HOSTIP_EXISTS(hostIP) then
        Create Host node with hostIP
    end if
end function
```

In Neo4J, the Client node is linked to the Host node to represent the client-server relationship. Also, the Host node is linked to the Detection node to capture the occurrence of the attack.

V. EXPERIMENT

To evaluate the SMARTX effectiveness, accuracy tests were conducted, comparing it with Logistic Regression, Random Forest, SVM, and Naive Bayes. The models were tested on three datasets, and the results were measured by the percentage of XSS detected. The experiment was conducted on a system featuring an 8-core, 16-thread processor with 16 GB of RAM, with all tests executed using Python version 3.12.0.

A. XSS Detection Accuracy

We first measured the accuracy of our SMARTX system and other machine learning models by using two datasets as follows.

Dataset 1: XSS_20000_Line.txt

This dataset contains 20,000 lines of XSS attack data, designed to evaluate the models' ability to detect actual XSS attacks. Table 1 demonstrates the percentage of detected XSS payloads for each model, showing the effectiveness of their detection algorithms.

Table 1: XSS Detection Accuracy on Attack Data

Model	XSS Detected (Percentage)
Logistic Regression	80.45%
Random Forest	76.87%
SVM	90.23%
Naive Bayes	26.27%
SMARTX	98.43%

As presented in Table 1, our SMARTX Model achieved the highest detection rate at 98.43%, outperforming all the other models. Naive Bayes had the lowest performance at 26.27%, which could be due to its simplistic probabilistic approach that may not handle complex XSS attack patterns as effectively as other models.

Dataset 2: Non_XSS_180000.txt

This dataset contains 180,000 lines of non-XSS data and is used to evaluate the models' ability to avoid false positives (incorrectly flagging benign inputs as XSS). Table 2 presents the false positive rates of the detection of non-XSS cases.

Table 2: False Positive Rates on Non-XSS cases

Model	XSS Detected Percentage
Logistic Regression	0.93%
Random Forest	1.08%
SVM	0.95%
Naive Bayes	52.2%
SMARTX	1.83%

Our SMARTX maintains a low false-positive rate of 1.83%, showing that it can reliably distinguish non-XSS inputs. In contrast, Naive Bayes incorrectly detected 52.2% of non-XSS inputs as XSS, which highlights a major drawback of this model in environments with diverse or complex input.

B. Performance Evaluation

We conducted the experiments to measure the detection and query speed of our proposed system and four ML models including Logistic Regression, Random Forest, SVM, and Naive Bayes. Fig. 3 below visualizes the detection speed of each model across different file sizes. As shown, SVM exhibits the steepest increase in processing time, while Logistic Regression, Random Forest, and the SMARTX maintain relatively low processing times, even for the largest datasets.

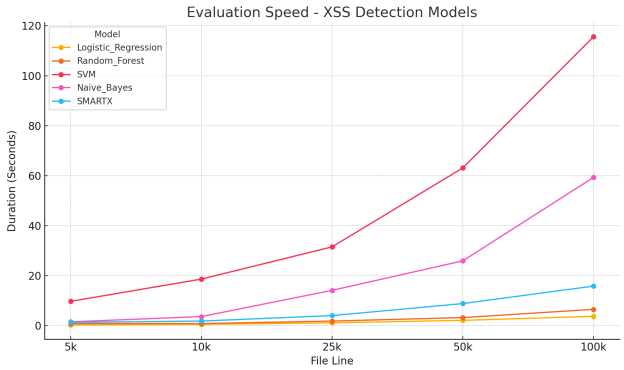

Fig. 3 Detection Speed Comparison by Number of Records

D .Query Speed

To evaluate the querying efficiency, a comparison between our proposed model using Neo4J and MySQL was conducted. The goal was to assess how quickly each database can execute queries as the dataset size increases from 10k to 2M records.

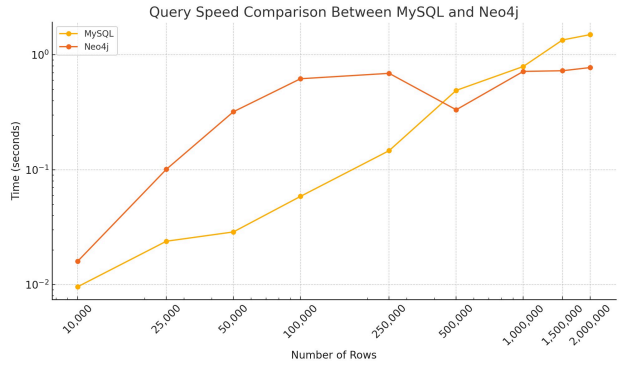

Fig.4 Query Speed Comparison

As shown in Fig. 4, for small datasets ranging from 10k to 250k records, MySQL slightly outperforms Neo4j, with query times of 0.0096 seconds versus 0.016 seconds, respectively, showing that both are efficient at this scale. However, as the dataset size increases to medium ranges (500k to 1M records), Neo4j begins to surpass MySQL, handling 500k records in 0.333 seconds compared to MySQL's 0.491 seconds. For large datasets (1.5M to 2M records), Neo4j consistently outperforms MySQL, completing queries for 2M records in 0.776 seconds, while MySQL requires 1.502 seconds, demonstrating Neo4j's superior efficiency with large, interconnected data.

VI. Conclusion

This paper presented the development and implementation of an advanced machine learning-based system for detecting and preventing Cross-Site Scripting (XSS) attacks in real-time. By utilizing a Multi-Layer Perceptron (MLP) classifier trained with TF-IDF vectorization and custom features, the system offers an efficient and scalable approach to identifying XSS vulnerabilities in web applications. Additionally, the use of multiprocessing ensures that the detection process remains fast, even under heavy traffic conditions. The system's integration with a Neo4j graph database improves the system's ability to track recurring attacks but also aids in understanding complex attack patterns and trends. Through extensive testing, the system has shown high accuracy in detecting various XSS payloads, outperforming traditional methods. Future improvements could focus on further optimizing the model for even faster detection speeds and expanding the system to detect other types of web-based vulnerabilities.

References

[1] A. Morais, B. Rolim, and J. Neto, "Analysis of Cross-Site Scripting (XSS) Threat Vectors," *Computer Networks*, vol. 212, p. 108131, 2022.

[2] G. Forkan, S. Debnath, and G. Roy, "Machine Learning Approaches to XSS Detection," *Future Internet*, vol. 11, no. 8, p. 177, 2019.

[3] Y. Li, Z. Zhao, and L. Cao, "Cross-Site Scripting Detection using Neural Networks," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 927-936, 2020.

[4] J. Patel, L. Walker, and H. Turner, "Mitigating Web Vulnerabilities through Machine Learning Approaches," *IEEE Access*, 2023.

[5] P. Iyer and R. Gupta, "Secure Web Applications using Real-Time XSS Detection Techniques," *IEEE Trans. Cloud Comput.*, 2021.

[6] D. Ali and H. Hassan, "Modern Approaches for Cross-Site Scripting Detection," *Information*, vol. 15, no. 7, p. 420, 2023.

[7] A. Murugan and S. Rajan, "AI-Based Detection of Web Threats: A Comprehensive Study," *Arab. J. Sci. Eng.*, vol. 49, pp. 9140-9152, 2023.

[8] X. Zhang, Y. Zhou, S. Pei, J. Zhuge and J. Chen, "Adversarial Examples Detection for XSS Attacks Based on Generative Adversarial Networks," in *IEEE Access*, vol. 8, pp. 10989-10996, 2020

[9] P. Zheng and Y. Zhang, "Enhancing the Security of Web Applications Using AI," *IEEE Access*, 2021.

[10] A. Fadli, M. Noor, and R. Rahman, "A Neural Network-Based Approach to XSS Detection," *IEEE Trans. Cloud Comput.*, 2023.

[11] K. Tamamura, S. Sakai, K. Watarai, S. Okada and T. Mitsunaga, "Detection of XSS Attacks with One Class SVM Using TF-IDF and Devising a Vectorized Vocabulary," *2023 IEEE International Conference on Computing (ICOCO)*, Langkawi, Malaysia, 2023, pp. 35-40.

[12] T. Suzuki and K. Arita, "Implementing a Real-Time XSS Detection System Using Deep Learning," *J. Biomed. Inform.*, 2023.

[13] M. Gill and N. Patel, "Machine and Deep Learning-based XSS Detection Approaches: A Systematic Review," *IEEE Trans. Cybern.*, 2023.

[14] B. Peng, X. Xiao and J. Wang, "Cross-Site Scripting Attack Detection Method Based on Transformer," *2022 IEEE 8th International Conference on*

Computer and Communications (ICCC), Chengdu, China, 2022, pp. 1651-1655
[15] D. Patterson and J. Doyle, "AI-Based Web Security Solutions for XSS Attacks," *Appl. Sci.*, 2023.
[16] N. Palak and M. Hart, "Data-Driven Approaches to Enhancing Web Application Security," *Int. J. Data Sci.*, 2023.
[17] S. Pillai and V. Verma, "A Novel Web Attack Detection Mechanism Using Maximal-Munch With Torrent Deep Network," in *IEEE Transactions on Cloud Computing*, vol. 11, no. 4, pp. 3591-3600, Oct.-Dec. 2023,.