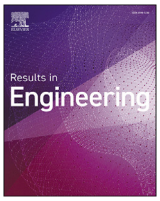
Contents lists available at ScienceDirect

Results in Engineering

journal homepage: www.sciencedirect.com/journal/results-in-engineering



Research paper

XSShield: A novel dataset and lightweight hybrid deep learning model for XSS attack detection

Gia-Huy Luu [a,b], Minh-Khang Duong [a,b], Trong-Phuc Pham-Ngo [a,b], Thanh-Sang Ngo [a,b], Dat-Thinh Nguyen [a,b], Xuan-Ha Nguyen [a,b], Kim-Hung Le [a,b,*]

[a] University of Information Technology, Ho Chi Minh City, Viet Nam
[b] Vietnam National University, Ho Chi Minh City, Viet Nam

ARTICLE INFO

Keywords:
XSS detection
Deep learning
CNN-LSTM model
Web security

ABSTRACT

With the proliferation of web applications, cross-site scripting (XSS) attacks have increased significantly and now pose a significant threat to users' information security and privacy. To enhance the efficiency of XSS attack detection, the adoption of machine learning (ML) and deep learning (DL) techniques offers promising solutions, but their effectiveness is limited by the lack of comprehensive and diverse datasets. Moreover, existing approaches often prioritize detection accuracy over real-time processing capabilities, which are essential for effective defense. To address these challenges, in this paper, we propose a novel framework that automatically collects web resources, efficiently extracts informative features, and constructs an up-to-date XSS attack dataset, which is then used to train a machine learning-based XSS detection model. Using this framework, we created and published a well-structured dataset over 100,000 samples for the research community. Furthermore, we present a hybrid detection model that leverages the strengths of both Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks. Extensive evaluations of our dataset demonstrate that the proposed model outperforms other baseline ML models across various metrics, including processing rate. Notably, our model achieves an accuracy of 99.27% while maintaining a low false positive rate of 0.06% and high processing rate of exceeding 1000 samples per second. These results highlight its high accuracy and robustness in detecting XSS, and suitability for real-time applications. Our work presents a comprehensive solution for enhancing web application security by providing a diverse dataset and a high-accuracy detection model with low latency.

1. Introduction

1.1. The significance of cross-site scripting (XSS) attacks

Cross-site scripting (XSS) attacks, a type of injection attack, pose a significant cybersecurity threat to web applications. In an XSS attack, attackers can exploit XSS vulnerabilities by injecting malicious scripts into HTML webpages. When users visit these compromised pages, the browser executes the malicious code, enabling attackers to compromise the user's web session. Recently, the XSS threat has a widespread impact and earned growing concern in both the industry and academia.

The severity of the XSS threat is evident in many industry reports. For example, the Open Web Application Security Project (OWASP) consistently ranks XSS among the top 10 global web application security risks, with XSS placing third in their 2021 report [1]. Recent cases of advanced threats, such as the Samy worm and other exploited XSS vulnerabilities, have led to critical data breaches (unauthorized access to sensitive data) and disruption of web services [2]. Moreover, Academic research also reflects the increasing concern over XSS threats. A study by Jasleen Kaur et al. [3] highlighted the evolution and sophistication of XSS attacks, pointing out that over 37.2% of web-based attacks come from XSS. Besides, Hannousse and Yahiouche [4] indicated that XSS vulnerabilities are now a focal point in cybersecurity research, with numerous studies dedicated to developing more effective detection and mitigation strategies. Therefore, the increasing severity of XSS vulnerabilities highlights the need for robust solutions to mitigate their impact.

* Corresponding author at: University of Information Technology, Vietnam National University, Ho Chi Minh City, Viet Nam.
E-mail addresses: 21520916@gm.uit.edu.vn (G.-H. Luu), 21520957@gm.uit.edu.vn (M.-K. Duong), 21520402@gm.uit.edu.vn (T.-P. Pham-Ngo), 21522543@gm.uit.edu.vn (T.-S. Ngo), 19520982@gm.uit.edu.vn (D.-T. Nguyen), hanx@uit.edu.vn (X.-H. Nguyen), hunglk@uit.edu.vn (K.-H. Le).

https://doi.org/10.1016/j.rineng.2024.103363
Received 30 April 2024; Received in revised form 8 October 2024; Accepted 6 November 2024

Available online 13 November 2024

1.2. The current gaps in XSS detection methodology

Various approaches have been proposed to detect and prevent XSS attacks. Conventional methods, which rely on rule-based systems or the expertise of security professionals, face significant drawbacks. First, they struggle to detect increasingly sophisticated XSS payloads, resulting in reduced detection accuracy and higher false-positive rates. These systems are limited by predefined security rules, making them unable to adapt to the evolving of attack techniques and leading to higher resource consumption. Second, they depend heavily on known attack patterns labeled by security experts, making them labor-intensive and challenging to deal with novel attacks. This reliance on predefined rules and expert-labeled patterns reduces their ability to respond to novel and complex attacks [5]. To overcome these shortcomings, recent approaches have focused on developing machine learning-based solutions, which use advanced algorithms to learn and detect new attack patterns effectively. These solutions improve detection accuracy and offer ease of deployment without expert supervision.

However, machine learning-based solutions have faced two key challenges: training data preparation and processing speed. The first challenge stems from the lack of high-quality training datasets, as existing datasets are often imbalanced, containing a low number of malicious samples, and inaccurate labels [6]. These datasets are unpublished and lack practical feature extraction modules, limiting their ability to adapt to the evolving nature of XSS payloads (as further detailed in Table 1). Additionally, with the rapid increase of XSS attacks, the existing datasets may quickly become outdated and can not reflect the latest attack techniques, reducing their effectiveness in attack detection. Furthermore, some datasets provide isolated XSS payloads without the surrounding context of the web application, making it challenging to develop models that can understand the full context of an attack [7,8]. The second challenge lies in the prioritization of detection accuracy over processing speed in recent studies. This focus may lead to missing the critical requirement for real-time or near-real-time processing capabilities, essential for a robust defense against XSS attacks. In more detail, the authors of [9–11] prioritize enhancing the accuracy of attack detection without considering the complexity of the model, which is necessary for rapid detection and response systems to effectively counteract XSS attacks effectively. These challenges prevent the practical applicability of machine learning and deep learning (ML/DL) models for XSS attack detection. To bridge this gap, there is a need for XSS detection solutions that achieve high accuracy while maintaining low latency operation.

1.3. Research motivation and key contributions

Motivated by the aforementioned gap, in this paper, we present (1) a novel XSS detection framework, named XSShield, that automatically collects web resources and efficiently extracts informative features; (2) constructs a comprehensive and up-to-date XSS attack dataset, which covers a wide range of both XSS attacks and normal patterns; and (3) the proposal of a hybrid CNN-LSTM XSS detection model, which leverages the strengths of both Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) networks.

In detail, to address the challenge of limited datasets, XSShield provides a comprehensive feature extraction algorithm that gathers rich features from web resources, including HTML, JavaScript, and URLs. Then, using this extraction, it crawls reliable web resources to create a comprehensive, up-to-date dataset for the research community. This dataset, with its rich insights into evolving XSS attack patterns, facilitates the training of the ML/DL-based detection system.

To balance high accuracy with low latency operation, it inspires us to propose a novel XSS attack detection system that leverages a combination of convolutional neural networks and long short-term memory models that could accurately detect XSS attacks at low latency. The CNN layers act as feature detectors, analyzing extracted data to identify characteristic patterns associated with XSS attacks. Unlike CNNs, LSTM layers analyze the features in sequence, enhancing the ability to understand the context and relationships between them, which may be missed by CNNs. For example, LSTM can identify a specific sequence of function calls often used in XSS attacks (malicious code is often in a specific order), even if they are spread across different parts of the extracted data.

Through our extensive experiments, we demonstrate the superior performance of the proposed detection model compared to other machine-learning approaches. Furthermore, we have made the dataset publicly available to support future research on XSS detection. In summary, the key contributions of this research are highlighted below:

- We introduce a novel framework with a comprehensive feature extraction module designed to automatically extract relevant features from web resources. This framework serves as a preceding component in collecting and feeding the deep learning model with a realistic and up-to-date dataset that captures the dynamic behavior of XSS attacks.
- We propose a novel XSS detection model that combines Convolutional Neural Network layers for extracting valuable insights and Long Short-Term Memory layers to boost detection accuracy. This hybrid architecture leverages the strengths of both CNN and LSTM to enhance the model's ability to identify XSS attacks effectively.
- We construct an extensive and up-to-date XSS dataset containing over 100,000 samples. This dataset includes a wide range of XSS attack patterns and benign samples, ensuring a comprehensive representation of real-world scenarios. To facilitate further research and development in the field of XSS detection, we make our dataset publicly available, serving as a valuable resource for developing and benchmarking XSS detection techniques.[1]

The remaining of this paper is organized as follows: In Section 2, we describe recent XSS detection methods using machine learning. Then, in Section 3, we present the proposed framework that includes the crawling model, feature extraction, and attack detection model. In Section 4, we briefly describe the experimental design, evaluation mechanism, experiment results, and discussion. Finally, we conclude the paper with recommendations for potential directions of this research area in Section 5.

2. Related works

2.1. Background on XSS attacks

Cross-site scripting (XSS) attacks are a type of security vulnerability found in web applications, where malicious scripts are injected into otherwise benign and trusted websites. The basic concept of XSS involves exploiting a vulnerability that allows an attacker to inject harmful code, typically in the form of JavaScript, which is then executed by the victim's browser. This attack usually initiates a cyber-chain, which leads to a range of harmful outcomes, including the theft of sensitive information, session hijacking, defacement of websites, and the spread of malware. As followed by research [3], XSS attacks can be grouped into four different categories based on their natural characteristic.

- ***Stored XSS attack***: occurs when a malicious script is permanently stored on the target server, such as in a database, comment field, or forum post. Whenever a user visits the page containing the script, it gets executed, potentially compromising the user's data.

[1] Dataset available at: http://clouds.iec-uit.com/publicdataset/xss_detection_2021.csv.

Table 1
The summary of related works on detecting XSS attack.

ID	Paper	Release	Dataset						Model		Performance (%)
			Dataset	No. Fts	Type of Features	No. Samples	Source	Availability	Features Processing	Detection Model	
1	MNN-XSS: Modular Neural Network Based Approach for XSS Attack Detection [9]	2022	Custom	50	HTML, JS, URL attributes	11,000	Figshare	No	Pearson correlation method	MNN	ACC: 99.96 PPV: 99.95 TPR: 100.0 F1: 99.95
2	Detection and defending the XSS attack using novel hybrid stacking ensemble learning-based DNN approach [10]	2022	XSS Payload List	N/G	URLs	N/G	Github Kaggle	No	N/G	DNN	ACC: 99.59 PPV: 99.56 TPR: 99.78 F1: 99.67
3	A Hybrid Machine Learning Model to Detect Reflected XSS Attack [38]	2021	XSS Payload List	N/G	XSSed Payload strings	13,000	Github	No	N-Gram	CNN	ACC: 99.01 PPV: 98.72 TPR: 99.46
4	Detection of XSS in web applications using Machine Learning Classifiers [39]	2020	Custom	24	URLs, JS attributes	1,611	N/G	No	N/G	K-NN	ACC: 96.90 PPV: 97.00 TPR: 100.0 F1: 98.00
5	The Detecting Cross-Site Scripting (XSS) using Machine Learning methods [40]	2020	XSS Payload List	N/G	JS parameters	240,000	Github	No	N/G	Decision Tree	ACC: 98.81 PPV: 99.19 TPR: 93.70 F1: 95.90
6	Detecting web attacks with end-to-end deep learning [41]	2019	Custom	N/G	N/G	1,725	N/G	No	N/G	SVM	PPV: 72.80 TPR: 100.0 F1: 84.30
7	CODDLE: COde-injection Detection with Deep LEarning [32]	2019	XSS Payload List	N/G	Injected payload	N/G	Github	No	Multi-level Code Encapsulation	CNN	ACC: 90.20 PPV: 99.00 TPR: 90.00
8	An ensemble learning approach for XSS attack detection with domain knowledge and threat intelligence [42]	2019	XSS Payload List	30	JS attributes	151,658	Github	No	Entropy-based Discretization	Bayesian	ACC: 98.54
9	MLPXSS: An Integrated XSS-Based Attack Detection Scheme in Web Applications Using Multilayer Perceptron Technique [43]	2019	Custom	41	HTML, JS, URL attributes	138,569	XSSED	No	Dynamic Features Extraction	ANN	ACC: 99.32 PPV: 99.21 TPR: 98.35 F1: 98.77 FPR: 0.31
10	DeepXSS: Cross Site Scripting Detection Based on Deep Learning [44]	2018	Custom	N/G	XSSed Payload strings	40,637	XSSED DMOZ	No	word2vec	LSTM	PPV: 99.50 TPR: 97.90 F1: 98.70
11	XSSClassifier: An Efficient XSS Attack Detection Approach Based on Machine Learning Classifier on SNSs [45]	2017	Custom	25	URL, HTML, SNS attributes	1,000	DMOZ XSSED Weibo	No	N/G	ADTree	ACC: 97.10 PPV: 97.00 TPR: 97.20 F1: 97.10 FPR: 9.20
12	Our Research		Custom	80	HTML, JS, URL attributes	107,406	XSSED Cisco	Yes		CNN-LSTM	ACC: 99.27 PPV: 99.65 TPR: 95.61 F1: 97.59 FPR: 0.06

N/G: Not Given.

- ***Reflected XSS attack***: happens when the injected script is reflected off a web server, such as in a search result or error message. The script is executed immediately as part of the response, typically delivered via a link that the victim must click on.
- ***DOM-based XSS***: occurs within the Document Object Model (DOM) of the web page, rather than in the HTML or server-side scripts. The attacker manipulates the DOM environment directly, often by injecting or altering client-side JavaScript code to achieve malicious results.
- ***Mutation-based XSS***: is a more advanced form of attack where the injected script is altered or mutated by the web browser as it processes the page. This mutation can cause the script to bypass security filters or sanitation measures, making it particularly difficult to detect and prevent.

Nowadays, the increasing frequency and sophistication of Cross-Site Scripting attacks pose a critical threat to web application security. In 2021, XSS was re-labeled as a part of the Injection vulnerability that ranks 3rd in the top 10 web application security threats provided by OWASP [1] and continuously stayed in this list from 2004. XSS attack is also listed as the most frequent application layer attack according to EdgeScan report 2021 [12], which takes account of 37% reported attacks and twice as much with the second attacks. Moreover, the research in [13] also lists XSS as one of the most popular web attacks that threaten user privacy and app operation. Therefore, researching methods for detecting XSS vulnerabilities has become significant and gained attention from security researchers. The existing solutions in this field can be categorized into two main approaches: using traditional methods or employing machine learning.

2.2. The existing solutions on detecting XSS attacks

2.2.1. Conventional methods

Conventional methods are divided into two main approaches: static analysis and dynamic analysis. The static analysis method involves examining the source code of web applications to determine whether the web applications may have XSS vulnerabilities. In the paper [14], the authors use this method to predict whether a web application may have XSS vulnerabilities. But this approach faces a big challenge, it requires experts who directly review the source code, and sometimes humans may make a mistake, which leads to overlooking the critical vulnerabilities. Dynamic analysis involves generating inputs to simulate attacks and evaluating vulnerabilities by examining whether the output is related to the malicious input. It is similar to simulating the behavior of an attacker exploiting vulnerabilities to detect exploitable flaws in the application. The study [15] utilized this dynamic analysis method and designed a framework, *TT-XSS of DOM-XSS*, based on taint tracking.

2.2.2. Machine learning and deep learning-based methods

Besides conventional methods, the application of machine learning and deep learning (ML/DL) models is becoming widely popular nowadays. Many researches demonstrated that the application of ML/DL-based could result in higher accuracy and optimize cost-efficiency [16–18]. For example, the research [19] indicated that a machine learning model such as k-nearest neighbors, naive bays, or an ensemble of decision trees could detect XSS attacks with high accuracy. Similarly, the research [20] also provides insights into applying a machine learning model to detect XSS attacks from non-alphanumeric URLs. In the research [21], the authors proposed the Deep Q Learning (DQN) algorithm to transform the parameters of XSS malicious code to enhance learning effectiveness and create a tool to detect XSS automatically through payload analysis. Besides, the study [22] presents a solution based on an ensemble learning approach learned with domain knowledge and threat intelligence. This method combines several individual learners to create a more accurate classifier, which could increase the generalization ability of the model. The research [23] also proposed an attribute extraction method that we rely on and further develop in this work. The paper utilizes the BeautifulSoup library, combined with the html5lib parser, to extract raw HTML files and the Esprima JavaScript parser to extract the AST (abstract syntax tree) from JavaScript code. The goal is to identify and extract essential features related to XSS attacks from the collected raw data. Some novel algorithms proposed in research [24–29] show promise for future research in detecting XSS attacks.

The application of the deep learning model in detecting XSS attacks also gained attention from researchers [13]. In research [10], the author provides a novel hybrid stacking ensemble learning-based deep neural network method that can detect XSS URLs. Training with XSS payload from Github and Kaggle, this method achieves accuracy up to 99.5%. On the other hand, researchers have also gained interest in applying convolutional neural networks and long-short-term memory architecture to detect XSS attacks. For instance, research [30] proposed an XSS detection model based on CNN and archive accuracy of 98.62. In research [31], the authors proposed to use a LSTM-attention based model to detect XSS attacks from XSS payloads. Similarly, In the research [32], the author proposed an intrusion detection system (IDS) named COD-DLE. This IDS combines data preprocessing to eliminate randomness in data with a CNN model for higher accuracy classification. Research [33] proposed a hybrid framework based on CNN architecture and Cookie Analysis Engine (CAE) to protect web applications from XSS attacks. The model was trained with features extracted from HTTP request parameters and was evaluated on their custom large dataset. Moreover, the framework uses CAE to check valid incoming cookies. The system achieves a high accuracy of up to 98.74% on public datasets. Another approach proposed in research [34] showed that a model based on Natural Language Processing (NLP) and Programming Language Processing (PLP) could be used to detect XSS vulnerability in PHP and Node.js source codes. The models were trained with synthetic data from generated PHP and Node.js databases and outperformed other statistic analysis tools such as ProgPilot, Pixy, RIPS, and AppScan. Some novel deep learning solutions proposed in research [35–37] show promise in detecting XSS attacks.

In summary, most studies address XSS vulnerability detection at high accuracy through several approaches, such as static analysis, dynamic analysis, and the combined use of machine learning models. However, with the evolution of XSS attacks, the latest survey research [3,13] highlights several research gaps: the lack of standard XSS benchmarking datasets, limitations in feature extraction, and the need for low-latency detection models. Addressing these challenges, this research focuses on introducing a novel dataset and comprehensive features extraction algorithm that extracts valuable information from various web resources. Moreover, we propose a lightweight attack detection model powered by advanced deep-learning architecture, aiming to enhance the effectiveness and efficiency of XSS vulnerability detection.

3. XSShield

In this section, we provide a comprehensive overview of our proposed XSS detection system. We begin by presenting a high-level architecture of the system, outlining its main components and their interactions. Subsequently, each component is described in detail, focusing on its role and the techniques. Finally, we present the novel deep learning-based XSS detection model.

3.1. Overview

Cross-Site Scripting (XSS) attacks pose a significant threat to web application security, thus detecting and preventing XSS vulnerabilities is a critical challenge for information security experts and web application developers. To address this issue, we propose a novel framework aiming to detect accurately and timely XSS attacks. Our approach consists of three key stages, as illustrated in Fig. 1:

- The first stage involves preparing a diverse and representative dataset for training the detection model. We develop a web crawling module to collect data from various sources, including benign webpages and webpages with known XSS vulnerabilities. The collected data is then preprocessed and combined to form the raw dataset.
- The second stage focuses on extracting informative features from the raw dataset. We develop a dynamic feature extraction module that automatically extracts three key feature groups from webpage sources: HTML-based, JavaScript-based, and URL-based features. These feature groups are chosen to capture the essential characteristics of XSS vulnerabilities. The extracted features are then encoded into numerical representations to facilitate the training of the detection model.
- The final stage involves training a deep learning-based detection model to accurately identify webpages with XSS vulnerabilities. We propose a hybrid architecture that combines CNNs and LSTMs to effectively capture both spatial and temporal dependencies in the extracted features. To ensure optimal performance, we employ the Optuna framework for automated hyperparameter tuning, which helps in finding the best configuration of the model parameters based on the validation set performance.

3.2. Crawling module

In this phase of our research, we outline a systematic approach to automatically downloading web content from reliable sources for extracting features and constructing datasets (as described in Stage 1 in Fig. 1). To ensure the quality and relevance of collected data, it is crucial to select reliable sources and employ efficient crawling techniques, therefore, our crawling process involves two main tasks: surveying and

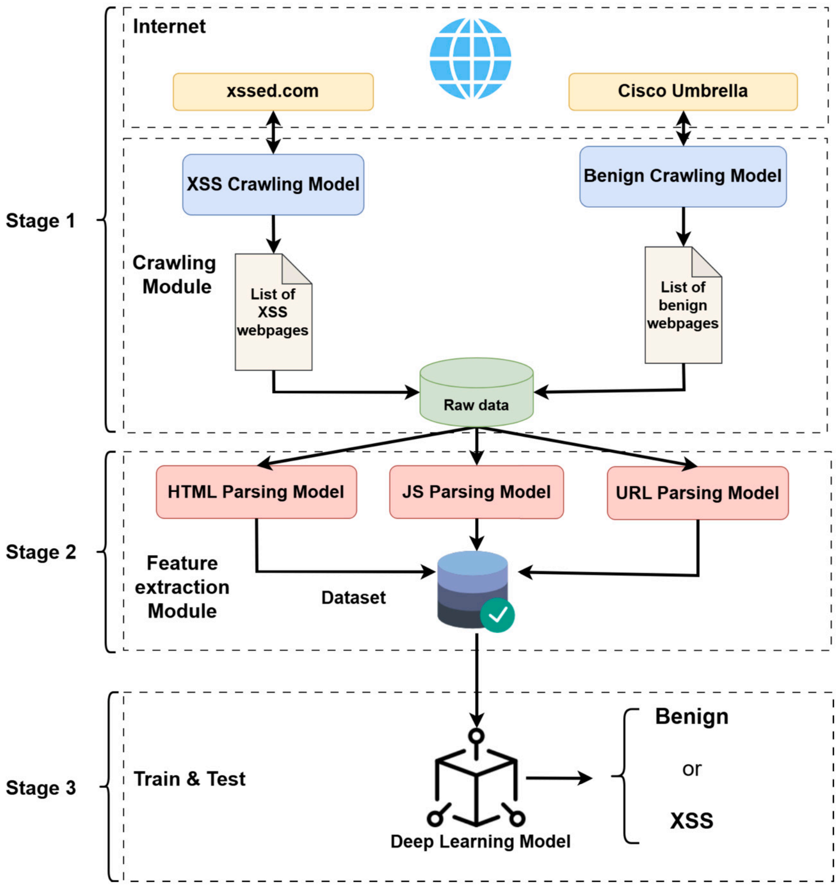


Fig. 1. The overview of the proposed system.

selecting reliable sources for collecting available URLs and crawling web page content.

- ***Surveying and selecting reliable databases of web URL***: we surveyed and selected two reliable sources for obtaining web URLs: the Umbrella Popularity List for benign web pages and xssed.com for malicious web pages. We select these data sources based on their reliability, diversity, and relevance to our research objectives in XSS detection. In detail, the Umbrella Popularity List, created by Cisco Umbrella [46] is a list of top-reaching domains based on their popularity among internet users. This list contains a wide range of popular domains across various categories, providing a representative sample of benign web content. For malicious URLs, xssed.com [12] is the largest specialized platform focusing on cross-site scripting vulnerabilities. In addition, each reported vulnerability passes through expert verification process before publishing, ensuring the integrity of our malicious samples. It also regular updates from the security community with emerging XSS attacks.
- ***Crawling webpage contents***: After obtaining the list of domains, we verified if they were still online and proceeded to crawl available benign and malicious web URLs. Once selected, the extracted URLs are then crawled using automated scripts built on Python libraries like requests and BeautifulSoup, respecting robots.txt guidelines and ensuring data quality through validation checks. After crawling, the content of active malicious and benign web then formed the raw datasets and was fed into the feature extraction module to extract valuable features. The labeling of our dataset was based on the source of each URL. URLs obtained from the Cisco Umbrella Popularity List are labeled as benign (0), and on the contrary, URLs sourced from XSSed.com were labeled as malicious (1). This binary labeling approach aligns with our goal of training a model to distinguish between benign web pages and those having XSS vulnerabilities.

By using reliable sources and crawling actual websites rather than generating synthetic data, our dataset represents real-world scenarios, including the complexities and nuances of both benign and malicious web content. Furthermore, our automated process allows for regular updates to the dataset, ensuring that it includes the latest benign web designs and XSS attack patterns.

After carefully crawling the data, we obtained a raw dataset with approximately 107,406 samples. This substantial size allows us to capture a wide diversity of XSS attack patterns and benign web content, thereby better representing the vast and varied nature of real-world web traffic. The large sample size enabled us to provide a substantial number of malicious samples for training. Moreover, it provides a large test set of over 21,000 samples, enhancing the validity of our evaluation by covering diverse scenarios.

3.3. Features extraction module

Following the crawling data stage, the feature extraction process is a comprehensive approach applied to all collected web resources during the data preparation stage. This process extracts informative features from three main components: HTML content, JavaScript content, and URL information. This feature set was designed to capture the several aspects of XSS attacks, from payload structure to execution methods and obfuscation techniques. For example, HTML features present elements that frequently exploited in XSS attacks, such as script tags and event attributes. URL features aim to identify malicious content in URL parameters. The extracted features form a set that serves as input to our detection model, both during the training and deployment phases. Our feature set comprises 80 features: 29 HTML features, 29 JavaScript features, and 22 URL features. This set is selected based on suggestions from domain expertise in XSS vulnerabilities, previous studies [9,43,45], and the need to balance the detection accuracy with the model complexity. HTML features focus on elements that are commonly exploited in XSS attacks. JavaScript features target DOM manipulation and suspicious code patterns that are often found in XSS payloads. URL features are designed to detect attempts to inject malicious content using the URL parameters. This comprehensive feature set surpasses most related works (described in Table 1), hence, ensuring our model can detect a wide range of XSS variants while minimising false positives. A detailed descri ption of each feature is provided in Table A.6.

To implement the proposed feature selection process, our proposed feature extraction module employed three sub-modules to extract informative features: the HTML parsing module, the JS parsing module, and the URL parsing module, corresponding to three main feature groups. Each sub-module is designed to extract specific features from the raw data described in Stage 2 of Fig. 1. After extracting, the extracted features are aggregated and constructed under an suitable format for the detection model. Specifically, the algorithms employed by each model are presented below:

3.3.1. HTML features extraction

Algorithm 1: Parsing HTML documents.

```
Input: Page Content
Output: HTML features
 1: $TG[\,] \longleftarrow$ list of HTML tags
 2: $AT[\,] \longleftarrow$ list of HTML attributes
 3: $EV[\,] \longleftarrow$ list of HTML events
 4: $H_{FV}\{\,\} \longleftarrow \emptyset$
 5: $P \longleftarrow$ BeautifulSoup(Page Content,'html5lib')
 6:
 7: for each $node \in P$ do
 8:     for each $t \in TG$, $a \in AV$, $e \in EV$ do
 9:         $H_{FV}[t]$ += ($t$ exist in $node$) ? 1 : 0
10:         $H_{FV}[a]$ += ($a$ exist in $node$) ? 1 : 0
11:         $H_{FV}[e]$ += ($e$ exist in $node$) ? 1 : 0
12:     end for
13: end for
14:
15: $H_{FV}$[html_length] = $len$(Page Content)
16:
17: return $H_{FV}$
```

HTML (Hypertext Markup Language) is the basis for constructing websites and web applications. Developers develop the structure and content of a webpage via a system of tags and attributes to define elements such as headings, paragraphs, links, images, forms, and more. Despite its essential role in building websites, HTML can also be abused by attacker to launch various types of attacks on end-users. For example, attackers can exploit HTML tags to execute XSS attacks (e.g., *div*, *script*, *iframe*, *frame*, ...) by injecting JavaScript code. Furthermore, attacker can leverage event triggering (*onclick*, *onmouseover*, *onload*, *onerror*, ...) to execute JS code through HTML attributes (*href*, *src*, *target*, *action*, ...). This means that when users perform seemingly normal actions, it becomes a point where malicious code is executed. Therefore, analyzing HTML source code is valuable in detecting XSS attacks.

We use the html5lib parser and the BeautifulSoup library in Python to extract features from tags, attributes, and events from raw HTML files. The html5lib parser allows us to parse pages similar to how web browsers do, ensuring accuracy and consistency in understanding and processing HTML5 documents. Meanwhile, the Python BeautifulSoup library is used to extract necessary data from HTML source code, such as tags, attributes, and events. It can work in parallel with other parsers, specifically html5lib, to provide a robust and flexible solution for exploring and extracting information from modern websites. In the HTML Parsing Module, we apply the algorithmic details presented in Algorithm 1 to extract a total of 29 features. More specifically, we analyze the frequency of appearance by counting the occurrences of tags, attributes, and events in the HTML code, and formed them as the corresponding features. Moreover, we also monitor the length of the HTML source code and add one feature to represent it.

3.3.2. JavaScript features extraction

Algorithm 2: Parsing JavaScript codes.

```
Input: Page Content
Output: JS features
 1: $DO[\,] \longleftarrow$ list of dom Object
 2: $JP[\,] \longleftarrow$ list of JS properties
 3: $JM[\,] \longleftarrow$ list of JS method
 4: $TG[\,] \longleftarrow$ list of HTML tags
 5: $AT[\,] \longleftarrow$ list of HTML attributes
 6: $EV[\,] \longleftarrow$ list of HTML events
 7: $JPP[\,] \longleftarrow$ list of JS Pseudo Protocol
 8: $JS_{FV}\{\,\} \longleftarrow \emptyset$
 9: $P \longleftarrow$ Page Content
10: JS_strings[ ] $\longleftarrow$ Extract JS code from Page Content
11:
12: $JS_{FV}$[js_define_function] = countFunctDeclaration($P$)
13: $JS_{FV}$[js_function_calls] = countFunctCalls($P$)
14:
15: tokens = Esprima.tokenize[JS_strings]
16: Stringlist[ ]
17: for each $token \in$ tokens do
18:     if token[type] == Identifier and token[value] exist in {DO, JP,
        JM} then
19:         $JS_{FV}$[token[value]] += 1
20:     else if token[type] == String then
21:         Stringlist.append(token[value])
22:     end if
23: end for
24:
25: if $Stringlist$ then
26:     $JS_{FV}$[js_min_length] = min(len(Stringlist))
27:     $JS_{FV}$[js_max_length] = max(len(Stringlist))
28: end if
29:
30: $JS_{FV}$[js_file] = any(js_file exist in $P$)
31: $JS_{FV}$[js_pseudo_protocol] = any($j \in$ JPP exist in $P$)
32:
33: return $JS_{FV}$
```

JavaScript is utilized in website design, creating captivating effects and fantastic features to serve users and improve server performance. Here, we represent JS as an Abstract Syntax Tree (AST). AST is beneficial for various purposes, and in this case, we use it to analyze and extract features of JS, helping us understand the structure and characteristics of the source code. This analysis is particularly crucial for identifying potential security vulnerabilities, specifically features closely related to the susceptibility to XSS attacks. Features in JS that impact the susceptibility to XSS attacks include the length of JS code, the number of function definitions, and function calls. Additionally, methods (such as *getElementsByTagName*, *write*, and *prompt*), Dom objects (such as *windows*, *location*), and properties (such as *cookies*, *innerHTML*) contribute to this analysis.

In the JS Parsing Module, we extract a total of 29 features by applying the algorithm described in Algorithm 2. First, the JavaScript code of the webpage is extracted and segmented into identifiable units (tokens). Each token is then evaluated to determine if it represents a DOM object, property, or method. The number of occurrences of each token is assigned as a feature value. Subsequently, regular expressions are used to count the number of function definitions and function calls. Finally, the existence of pseudo-JavaScript protocols is identified.

3.3.3. URL features extraction

Algorithm 3: Parsing URL addresses.

```
Input: Page URL
Output: URL features
 1: $TG[\,] \longleftarrow$ list of HTML tags
 2: $AT[\,] \longleftarrow$ list of HTML attributes
 3: $EV[\,] \longleftarrow$ list of HTML events
 4: $URD[\,] \longleftarrow$ list of URL redirection
 5: $UFD[\,] \longleftarrow$ list of URL frequent parameters
 6: $KE[\,] \longleftarrow$ list of keywords evil
 7: $SP[\,] \longleftarrow$ list of special characters
 8: $U_{FV}\{\,\} \longleftarrow \emptyset$
 9: $Ustr \longleftarrow$ url_decode($URL$)
10:
11: $U_{FV}$[url_length] = $len$(URL)
12: $U_{FV}$[url_duplicated_characters] = IsDuplicated($Ustr$)
13: $U_{FV}$[url_special_characters] = any($sp \in SP$ exist in $Ustr$)
14: $U_{FV}$[tag] = any($tag \in TG$ exist in $Ustr$)
15: $U_{FV}$[attribute] = any($attribute \in AT$ exist in $Ustr$)
16: $U_{FV}$[event] = any($event \in EV$ exist in $Ustr$)
17: $U_{FV}$[url_redirection] = any($urd \in URD$ exist in $Ustr$)
18:
19: for each $p \in UFP$ do
20:     $U_{FV}$[url_keywords_param] += ($p$ exist in $Ustr$) ? 1 : 0
21: end for
22:
23: for each $k \in KE$ do
24:     $U_{FV}$[url_keywords_evil] += ($k$ exist in $Ustr$) ? 1 : 0
25: end for
26:
27: $U_{FV}$[url_number_domain] = countDomains($Ustr$)
28: $U_{FV}$[url_number_ip] = countIPs($Ustr$)
29: $U_{FV}$[url_cookie] = any($document.Cookie$ exist in $Ustr$)
30:
31: return $U_{FV}$
```

The structure and parameters of a URL play a critical role, as attackers can leverage them for XSS attacks. Attackers may encode the URL to bypass testing tools and filters designed to screen for malicious URLs based on unusual keywords, attempting to conceal malicious code to execute undesirable actions on users. To address this issue, we propose a comprehensive URL analysis algorithm to collect malicious features that directly impact whether a webpage is susceptible to XSS attacks. As outlined in Algorithm 3, our method begins by decomposing the URL into its constituent parts, including HTML tags, URL direction, and URL parameters. Then, it extracts various features from these components, including the length of the URL, the presence of specific characters, and the occurrence of keywords. Finally, The algorithm then outputs a comprehensive list including 22 features that can be used to assess the potential risk of XSS attacks. Some of the features under consideration include:

- URL redirection: can be utilized to redirect users to malicious resources managed by attackers (*e.g., top.location, self.location, location, window.open, window.history, document.URLUnencoded, document.baseURI* ...).
- Properties, Tags, and Events of HTML: their presence in the URL is considered suspicious.
- Additionally, URL length: when a URL contains malicious code, it tends to be much longer than usual.
- Special characters, Keyword parameters, IP Address, and Number of domains appearing in the URL are also significant for XSS attacks.
- Evil keywords: dangerous keywords that frequently appear in URLs subjected to XSS attacks.

3.4. Proposed detection model

Deep Learning has affirmed its value in various fields, from image processing and object recognition to autonomous management systems and, recently, in cyber security. Among the numerous architectures that have been developed, the combination of convolutional neural networks (CNN) and long short-term memory (LSTM) networks have gained much attention from researchers. CNNs excel at capturing spatial patterns and local dependencies [47], while LSTMs are designed to identify long-range dependencies in temporal sequences [48]. By integrating CNNs and LSTMs, the model can learn the complex data structure.

In this research, we proposed an XSS detection model based on the CNN-LSTM architecture. Fig. 2 depicts our model, comprising two primary components: feature extraction component (which is constructed from CNN layers) and classification component (which is constructed from LSTM layers and Dense layer). Our proposed detection model operates on the extracted feature set obtained from the feature extraction module. It uses these features as input to detect XSS attacks and does not perform feature extraction from raw webpage content itself. When receiving input, the CNN layers can effectively capture key patterns associated with XSS attacks while reducing input dimensionality. The LSTM layer then captures sequential dependencies that are crucial for understanding the context of potential XSS payloads. The subsequent dense layers allow the model to learn nonlinear relationships between these high-level features, then providing the final classification decision. This architecture enables the detection model to understand complex relationships between input features in both spatial and temporal patterns and achieve higher accuracy in detecting XSS attacks. We implemented this CNN-LSTM model using the robust TensorFlow-Keras deep learning framework [49,50].

Feature extraction: The first component of our model, CNN, is trained to extract the learnable relationships among the URL features. The input vector x, which represents features of a URL, is passed through a number of convolutional blocks. Each block consists of a convolutional layer followed by a $ReLU$ activation layer, and a max-pooling layer. The convolutional layer, $Conv1D$, extracts the spatial relationship of features via a convolutional filter parameterized by kernel size, stride, and padding. The extracted spatial matrix f_1 is then forwarded to the subsequent $Maxpooling1D$ layer.

$$f_1 = Conv1D(x) \tag{1}$$

$$f_2 = MaxPooling1D(f_1) \tag{2}$$

The $Maxpooling1D$ layer reduces the size of the spatial matrix while retaining important features activated by $ReLU$. The mathematics presentation of $Conv1D$ and $Maxpooling1D$ layer is depicted in Equation (3) and (4):

$$f_1[i] = \sum_{j=0}^{k-1} W[j] \cdot x[i + j - \lfloor \frac{k}{2} \rfloor] \tag{3}$$

$$f_2[i] = \max_{j=i\cdot s}^{i\cdot s+p-1} x[j] \tag{4}$$

Where $f_1[i]$ and $f_2[i]$ depicts the output at index i after the operation. W is the convolution filter (also known as the kernel), and k is the size of the filter/kernel. s is the stride of the pooling operation, and p is the size of the pooling window.

Classification: The second component of our model, LSTM, is trained to learn the temporal dependencies of the data over a long sequence. Following the feature extraction steps, the matrix f_2 is passed through a Flatten layer to transform it into a vector $p_1 \in \mathbb{R}^k$, where $k = \frac{T}{2} \times F$, with T and F representing the time steps determined by the padding and strides of the Conv1D layer and the number of filters from the convolutional layers. Subsequently, p_1 serves as the input for the LSTM model to perform data augmentation and handle the long-term information loss issue in the data. This enables the model to remember important infor-

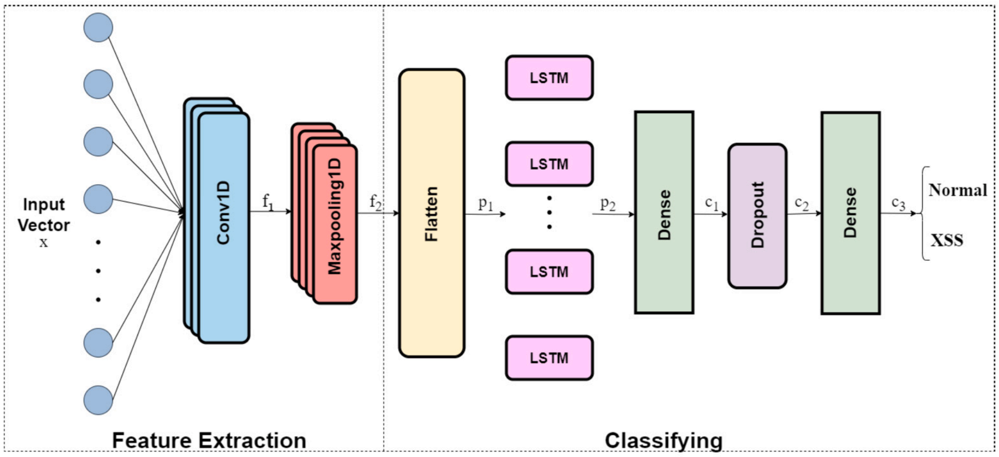


Fig. 2. The overview of the proposed detection model architecture.

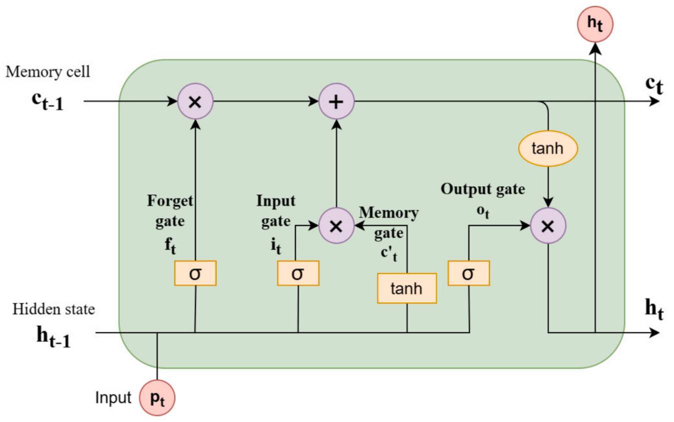


Fig. 3. The illustration of a long short-term memory cell design.

mation from the past and use it for prediction purposes in the future, effectively mitigating Gradient Vanishing/Exploding issues that conventional models may struggle with.

$$p_1 = Flatten(f_2) \tag{5}$$

$$p_2 = LSTM(p_1) \tag{6}$$

The vector input p_1 would be processed through three steps of a long short-term memory cell depicted in Figs. 3 with the mathematics presentation presented at Equation (7). In which:

- The forget gate plays a role in determining which information needs to be kept or discarded from the previous cell state c_{t-1}. Typically, a distribution function such as a logistic sigmoid is employed in this gate and takes the previous cell state c_{t-1} and current input p_t to make decisions. The output f_t is a value range between 0 and 1, which determines the proportion of previous memory to forget.
- The input gate determines which information from the current input x_t and the previous hidden state h_{t-1} to update into the cell state c_t. It is designed with two operations: the input gate i_t and the memory cell state c'_t. The input gate uses the sigmoid function, and the cell state c'_t is calculated by the *tanh* function. After two operations, the cell state c_t is updated by the forget gate output f_t, the input gate i_t, and the memory cell state c'_t as in Equation (7).
- The output gate calculates and returns the output of LSTM cell o_t. It decides if information from the current cell state c_t to output as hidden state h_t.

$$\begin{aligned}
i_t &= \sigma(W_{pi} \cdot p_t + W_{hi} \cdot h_{t-1} + b_i)\\
f_t &= \sigma(W_{pf} \cdot p_t + W_{hf} \cdot h_{t-1} + b_f)\\
c'_t &= \tanh(W_{pc} \cdot p_t + W_{hg} \cdot h_{t-1} + b_c)\\
o_t &= \sigma(W_{po} \cdot p_t + W_{ho} \cdot h_{t-1} + b_o)\\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t\\
h_t &= o_t \odot \tanh(c_t)
\end{aligned} \tag{7}$$

In the final layer of the classification, our model summarizes the data and provides classification results for XSS identification. The input vector p_2 is fed into a dense layer with a sigmoid activation function and then sent to a dropout layer to randomly drop some neurons to minimize the risk of overfitting the model. The model then performs a binary prediction step in the final Dense layer and gives a result that determines whether XSS or Normal.

$$c_1 = Dense(p_2) \tag{8}$$

$$c_2 = Dropout(c_1) \tag{9}$$

$$c_3 = Dense(c_2) \tag{10}$$

As observed, a neural network model comprises many layers with different functionalities. Each layer has its own set of parameters with a wide and diverse range of values. The challenge here is: How can we optimize and adjust these parameters and the number of layers to obtain the most efficient and effective CNN-LSTM neural network structure? To address this challenge, we used a Hyperparameter Optimization Framework called Optuna [51]. Optuna is chosen for its flexibility in defining hyperparameter search spaces, support for advanced optimization algorithms (e.g., Tree-structured Parzen Estimator (TPE), Covariance Matrix Adaptation Evolution Strategy (CMA-ES), and Random Search), and seamless integration with our TensorFlow-Keras implementation. This flexibility allows us to select the most appropriate algorithm based on their specific problem and computational resources. Moreover, many research [52–54] indicated that Optuna consistently outperforms other frameworks such as Hyperopt and Scikit-optimize in terms of both accuracy and efficiency when integrated with deep learning architecture.

To optimize our proposed model, we applied Optuna to determine the configuration of layers and parameters to return the model with the highest performance. Fig. 4 illustrates the workflow of Optuna applied to our CNN-LSTM model. It iteratively runs training trials on the training and validation sets with parameters for the model architecture computed to be the most optimal by the hyperparameter tuner. After the trials, Optuna determines the best-performing model by comparing the accuracy of each model on the test set and returns the values of the corresponding parameters for that model's architecture. The final parameter configuration is detailed in Table 2.

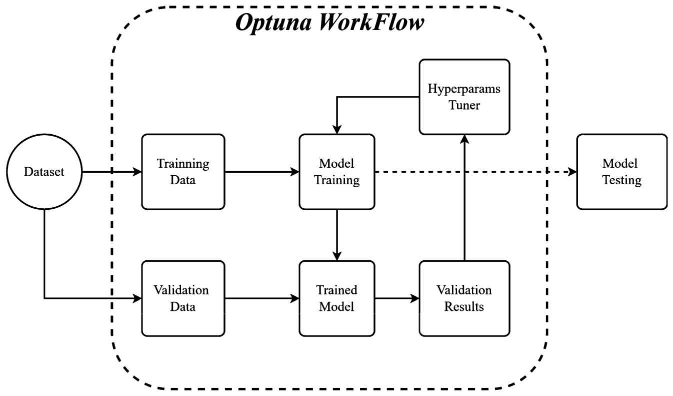


Fig. 4. The Optuna workflow to tune the model parameters.

Table 2
The detail parameter of the proposed model.

No	Layer Name	Layer Type	Filters - Kernel size - stride
1	Input	Local Binary and Numerical Features	-
2	Covolution1D_1	Convolution	362 - 3 x 1 - 1 x 1
3	ReLU_1	ReLU	-
4	MaxPooling1D_1	MaxPooling	2 x 1 - 2
5	Covolution1D_2	Convolution	86 - 3 x 1 - 1 x 1
6	ReLU_2	ReLU	-
7	MaxPooling1D_2	MaxPooling	2 x 1 - 2
8	Covolution1D_3	Convolution	205 - 5 x 1 - 1 x 1
9	ReLU_3	ReLU	-
10	MaxPooling1D_3	MaxPooling	2 x 1 - 2
11	Covolution1D_4	Convolution	371 - 3 x 1 - 1 x 1
12	ReLU_4	ReLU	-
13	MaxPooling1D_4	MaxPooling	2 x 1 - 2
14	LSTM	LSTM	160 neurons
15	ReLU_5	ReLU	-
16	Fc_1	Fully Connected	107 neurons
17	ReLU_6	ReLU	-
18	Drop_1	Dropout	30% dropout
19	Fc_2	Fully Connected	147 neurons
20	ReLU_7	ReLU	-
21	Drop_2	Dropout	30% dropout
22	Fc_3	Fully Connected	132 neurons
23	ReLU_8	ReLU	-
24	Drop_3	Dropout	30% dropout
25	Fc_4	Fully Connected	66 neurons
26	ReLU_9	ReLU	-
27	Drop_4	Dropout	30% dropout
28	Fc_5	Fully Connected	1 neurons
29	Sigmoid	Sigmoid	-
30	Output		

4. Experimental environment and results

In this section, we present a comprehensive evaluation of the proposed detection model, focusing on its detection accuracy, speed, and performance on resource-constrained devices. We compare our model's effectiveness against existing machine learning models, including both traditional machine learning approaches and deep learning-based techniques. This is due to the lack of publicly available source code and datasets, highlighted in Table 1. To demonstrate the lightweight and robustness of our model, we benchmark its inference time on a Raspberry Pi 4, a popular single-board computer representative of network edge devices.

4.1. Dataset

In this research, we evaluated the accuracy of our detection system using our dataset described in Section 3. The final dataset, obtained after collecting and extracting features, comprises 107,406 samples of normal and malicious instances. The class distribution ratio is approximately 5:1 for normal and malicious samples, respectively. This dataset has 80 features, which were extracted from our algorithm as detailed in Section 3.3. These features cover three main components of a website: HTML, JS, and URL. Specifically, 29 features are extracted from each of the HTML and JS components, while 22 features are derived from URL information. The features extracting algorithm prioritized extracting the most valuable features that can help the detection model reach high accuracy. The Table A.6 detailed the features set with its description. To ensure the validation of the experiment, we divided the datasets into training/testing subsets with a ratio of 8:2 respectively. The data distribution on each subset is described in Table 3.

Table 3
The data distribution of the experimental dataset.

	Benign	Malicious	Total
Training	72,698	13,226	85,924
Testing	18,220	3,262	21,482
Total	90,918	16,488	107,406

4.2. Evaluation metrics

In the binary classification problem, the confusion matrix is a popular metric for understanding the model's performance. Based on the confusion matrix elements, we use a range of evaluation metrics, starting with True Negative (TN) - indicating that cases labeled as 0 (benign) have been accurately predicted. Next, False Positive (FP) reveals instances where cases labeled as 1 (XSS attack) have been incorrectly predicted. False Negative (FN) signifies cases of XSS attacks mistakenly identified as benign (label 0), while True Positive (TP) denotes cases of XSS attacks accurately identified as an attack (label 1). Additionally, we calculated other metrics for a comprehensive evaluation, including precision, precision, recall, F1 score, misclassification rate, and AUC-ROC. The formulation for each metrics is provided below:

- Accuracy (ACC):

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}$$

- Precision (PPV):

$$PPV = \frac{TP}{TP + FP}$$

- Recall - True Positive Rate (TPR):

$$TPR = \frac{TP}{TP + FN}$$

- False Positive Rate (FPR):

$$FPR = \frac{FP}{FP + TN}$$

- F1-score:

$$\text{F1-score} = \frac{2 * Recall * Precision}{Recall + Precision}$$

- Misclassification rate:

$$\text{Misclassification rate} = 100 - ACC$$

4.3. Experiment environment configuration

In this research, we conducted experiments on two different environments to ensure the ability of the proposed system to work across various devices. The first environment is the default Colab environment provided by Google, while the second environment is an edge device.

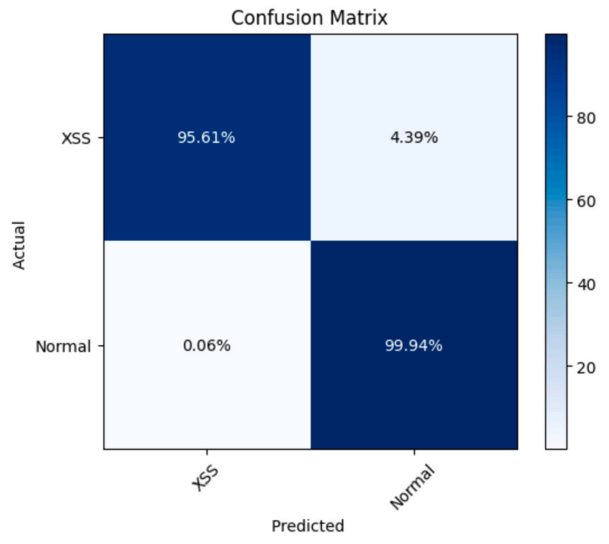


Fig. 5. The confusion matrix of our proposed XSS detection model, evaluated on the experimental dataset.

- The Edge gateway device used is a Raspberry PI 4B, a single-board computer using a Quad-core Cortex-A72 processor with 8 GB of RAM.
- The default Colab environment provided by Google uses Intel(R) Xeon(R) CPU @ 2.20 GHz processor and has 12 GB RAM.

Both environments were configured with Python version 3.10 and TensorFlow version 2.15. In addition, the model was tested without using GPU devices.

4.4. Experiment results

4.4.1. Our proposed model detection performance

The detailed detection performance of our proposed model is illustrated in Fig. 5. Overall, our approach delivers high true positive and true negative rates, whereas the rate of false positives and false negatives is minimal. Particularly, our model successfully identifies 95.61% of 3,262 XSS attacks in our experiment, whereas only 4.39% of these attacks are misclassified. On the other hand, our approach achieves a promising true negative rate of 99.94% and a negligible false positive rate of 0.06%. These results indicate the potential of our approach in detecting XSS attacks.

In Fig. 6, we evaluate our detection performance in the absence of hyperparameter tuning with the Optuna framework. In general, although the overall detection performance of our model with and without using Optuna does not differ significantly, the absence of hyperparameter tuning does result in a drop in recall - which is critical in attack detection. In particular, the accuracy without Optuna has an insignificant drop of 0.1%, whereas the precision remains unchanged. On the contrary, the recall of our model without Optuna degrades by 0.64%, which also leads to a decrease in the f1-score by 0.33%.

4.4.2. Baseline comparison

As highlighted in Table 1, comparing our proposed model with other existing research on XSS attack detection is challenging due to the lack of publicly available source code and datasets. Several studies, including those by Alqarni et al. [9] (used a custom dataset from figshare.com), Krishnan et al. [10] and Beraat et al. [38] (used XSS Payload Lists from Github and Kaggle), employ custom datasets that are unavailable for public use. Similarly, other research (Banerjee et al. [39], Kascheev and Olenchikova [40], Pan et al. [41], etc.) use private datasets without making them publicly available. Furthermore, the authors also did not provide the source code and hyperparameter configuration, preventing us from reproducing their models on our proposed datasets. Hence, it is important to note that the performance metrics of previous studies presented in Table 1 are based on their respective datasets, which differ from the dataset used in our experiments. To ensure a fair comparison, we implemented and evaluated the experiment with various machine learning and deep learning models, including random forest (RF), light gradient-boosting machine (LightGBM), extreme gradient boosting (XGBoost), deep neural network (DNN), convolutional neural network (CNN) and convolutional neural network + bi-direct long short term memory (CNN+BiLSTM).

Table 4 presents the comparative evaluation results between our proposed model and baseline models. We can see that the proposed model (Ours) achieves the highest scores across all evaluation metrics. For example, our model obtains 99.27% on the accuracy, surpassing the nearest competitor, CNN-BiLSTM, at 99.13%. For precision and recall metrics, it again takes the lead with 99.65% and 95.61%, respectively. Additionally, our proposed model also reaches the highest F1-score and AUC-ROC with 97.59% and 99.5%, respectively. This high accuracy ensures that legitimate web content is not misclassified as malicious and that XSS web content is not ignored. Thereby, it maintains the functionality and user experience of web applications.

Besides the high accuracy, in the context of web security, False positives in XSS detection can block legitimate web content, causing disruptions to normal website operations and damaging user experience. Therefore, our proposed model surpasses other baseline models when it archives the lowest FP rate of 0.06%, which means that only six out of 10,000 benign samples are incorrectly flagged as malicious, significantly reducing unnecessary alerts. This helps limit the effect on user experience. In summary, the evaluation results demonstrate that our approach is the most accurate, precise, and robust method for XSS detection based on the comprehensive comparative analysis presented in Table 4.

4.4.3. Our proposed model runtime resource consumption

Table 5, along with Figs. 7 and 8, presents a comprehensive analysis of the runtime performance of our proposed model in resource-constrained device environments. As shown in Table 5, the model's compact architecture, with only 264,878 parameters and a computational cost of 9.94e-05 GFLOPS, ensures its high processing rate with minimal resource consumption. The proposed model occupies only 15 MB of memory while achieving processing speeds of over 1020 samples per second in the Colab environment and 576 samples per second on edge devices. Additionally, Fig. 7 illustrates that the model utilizes approximately 85% of the CPU and requires over 100 MB of RAM in the Colab environment. Similarly, Fig. 8 shows the model's resource consumption on edge devices, using around 80% of the CPU and slightly over 10 MB of RAM. These experimental results confirm that the proposed detection system is lightweight and does not introduce performance bottlenecks, thereby maintaining the high responsiveness of web applications. Consequently, the model is well-suited for deployment across various device environments, including those with limited resources.

With the lightweight and high processing of our proposed model, it can be applied in real-world web security scenarios in various contexts. For instance, it can be integrated into Web Application Firewalls (WAFs) to provide an intelligent, real-time XSS detection layer that adapts to evolving attack patterns. Additionally, deployment at the Content Delivery Network (CDN) level allows for scanning and filtering content for XSS attacks before they reach the servers, offering a distributed security solution. On the server side, it can enhance input validation processes and analyze incoming requests in real-time. Moreover, efficient operation on edge devices enables client-side protection through browser extensions.

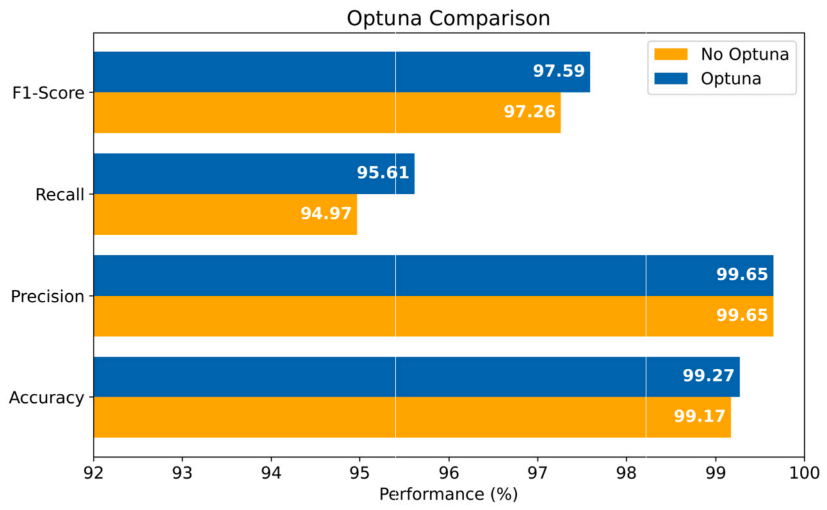


Fig. 6. Comparing the detection performance (%) of our proposed model with and without using Optuna..

Table 4
The comparison between our detection model with others baseline models(%).

Algorithm	Accuracy	Precision	Recall	F1-Score	FP-rate	Misclassification rate	AUC-ROC
RF	98.94	98.28	94.82	96.52	0.3	1.06	99.09
LightGBM	99.1	99.56	94.61	97.02	0.08	0.9	97.27
XGBoost	99.08	98.48	95.51	96.97	0.27	0.92	98.97
DNN	98.94	98.61	94.49	96.51	0.24	1.06	99.19
CNN	99.02	99.55	94.1	96.75	0.08	0.98	99.49
CNN-BiLSTM	99.13	99.49	94.88	97.13	0.09	0.87	98.73
Ours	**99.27**	**99.65**	**95.61**	**97.59**	**0.06**	**0.73**	**99.5**

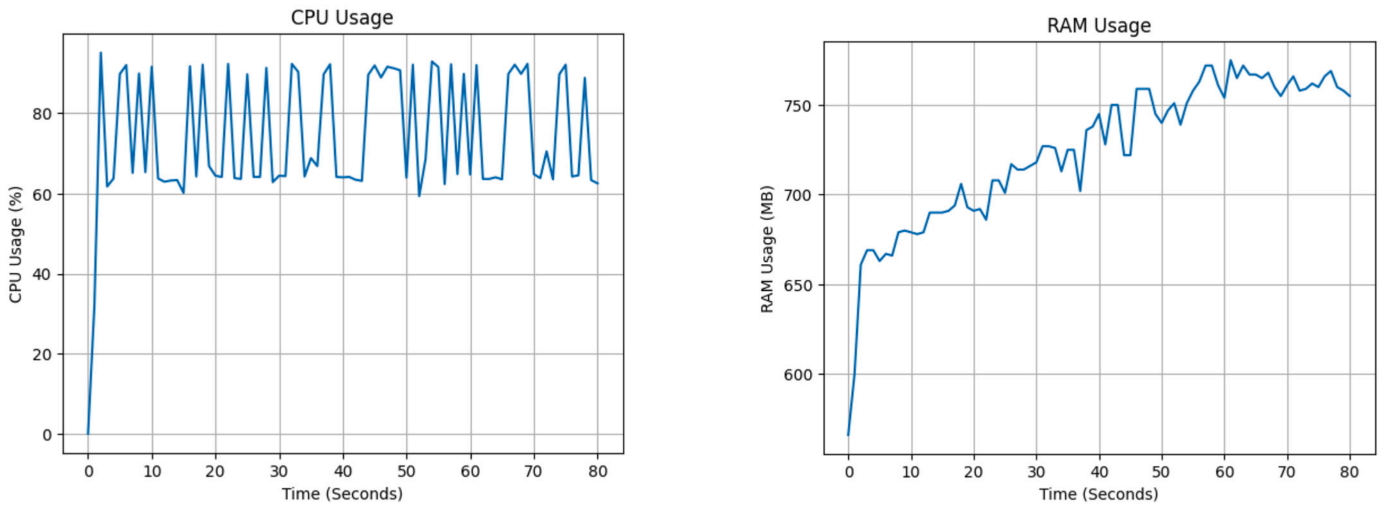


Fig. 7. The resource consumption of the proposed model on Google Colab environment.

4.5. *Limitation and future research*

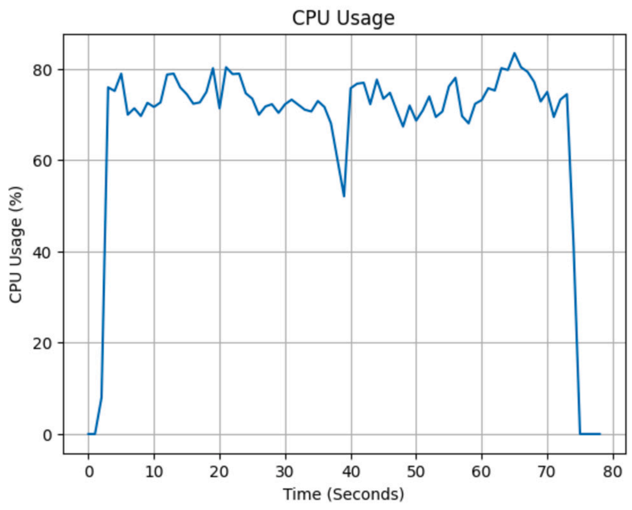
While demonstrating high accuracy and efficiency, we acknowledge that our current framework and detection model still have certain limitations that warrant further research:

- ***Real-world Performance***: While our model shows promising results in controlled experiments, its performance in real-world, highly dynamic web environments with diverse traffic patterns and attack variations needs further evaluation.
- ***Model Interpretability***: The use of deep learning techniques, while effective, can make it challenging to interpret why specific decisions are made by the model, which may be a concern in some security contexts.

Table 5
The runtime performance of the proposed model.

Model size	Total of params	GFLOPs	Processing rate on Colab	Processing rate on Edge
15 MB	264878	9.94e-05	1020 sample/s	576 sample/s

Building on the previous discussion, we propose some key areas for future research to further enhance the effectiveness and adaptability of



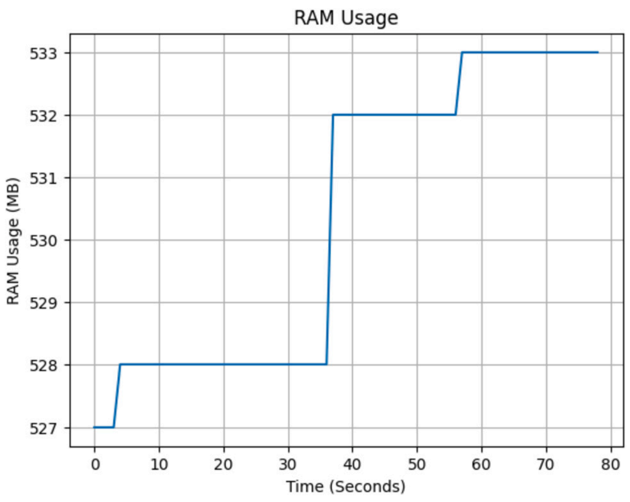


Fig. 8. The resource consumption of the proposed model on Raspberry Pi 4.

our XSS attack detection model. First, we suggest integrating adaptive learning techniques with our detection model. This would enable the model to continuously update its knowledge base with new data without requiring full retraining, thereby enhancing its capability to detect novel attacks even without labeled data. Second, we could expand the feature set with other web components like HTTP headers, cookies, and web storage to detect more complex and sophisticated XSS attacks. Third, with the growing relevance of adversarial attacks in cybersecurity, integrating adversarial detection techniques could enhance the model's resilience against such threats, making this a promising direction for further research. Lastly, in real-world scenarios, security administrators need more explanation from models than a prediction for the type of attack. Therefore, future research should focus on integrating explainable AI techniques, which would provide more interpretable results, thereby assisting security analysts in understanding and validating the model's outputs.

5. Conclusions

In this research, we have proposed an advanced solution to protect web-based services from Cross-Site Scripting (XSS) attacks. Our approach includes a novel framework and a hybrid CNN-LSTM detection model to significantly enhance the accuracy and efficiency of XSS attack detection. The proposed framework could automatically collect web resources and extract informative features from web pages, including HTML, JavaScript, and URL components. The detection model leverages the strengths of CNN and LSTM architectures to accurately identify the presence of XSS attacks. Furthermore, we construct a comprehensive benchmark dataset collected from reliable sources, containing more than 100,000 samples of both benign and XSS samples. We publicise this dataset along with our feature extraction algorithm to support further research in the XSS detection domain with a standardized comparison. Besides, we conducted extensive experiments on our detection model using the proposed dataset and compared it with other baseline detection models. The experimental results demonstrate the superior performance of our detection model, achieving a remarkable accuracy of 99.27%, surpassing all baseline machine learning and deep learning models across various evaluation metrics. Furthermore, our model has a low false positive rate of 0.06% and maintains a high processing speed, exceeding 1000 samples per second. These results highlight the practical capability of our approach in real-world scenarios, where efficient and accurate detection is crucial to protect web applications from XSS attacks.

CRediT authorship contribution statement

Gia-Huy Luu: Writing – original draft, Validation, Software, Methodology, Formal analysis, Conceptualization. **Minh-Khang Duong:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Conceptualization. **Trong-Phuc Pham-Ngo:** Visualization, Validation, Software. **Thanh-Sang Ngo:** Writing – original draft, Visualization, Validation, Software. **Dat-Thinh Nguyen:** Writing – review & editing, Validation, Software. **Xuan-Ha Nguyen:** Writing – review & editing, Visualization, Validation. **Kim-Hung Le:** Writing – review & editing, Supervision, Project administration, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This research is funded by Vietnam National University HoChiMinh City (VNU-HCM) under grant number DS2023-26-03.

Appendix A. The list of feature set

Table A.6
The detailed description of the features set of our proposed datasets.

Index	Feature	From	Description
1	html_tag_main	HTML Parsing	No. main tags
2	html_tag_section	HTML Parsing	No. section tags
3	html_tag_script	HTML Parsing	No. script tags
4	html_tag_iframe	HTML Parsing	No. iframe tags
5	html_tag_meta	HTML Parsing	No. meta tags
6	html_tag_link	HTML Parsing	No. link tags
7	html_tag_svg	HTML Parsing	No. svg tags
8	html_tag_form	HTML Parsing	No. form tags
9	html_tag_div	HTML Parsing	No. div tags
10	html_tag_style	HTML Parsing	No. style tags
11	html_tag_img	HTML Parsing	No. img tags
12	html_tag_input	HTML Parsing	No. input tags
13	html_tag_textarea	HTML Parsing	No. textarea tags
14	html_attr_selected	HTML Parsing	No. occurrences of the selected attribute
15	html_attr_target	HTML Parsing	No. occurrences of the target attribute
16	html_attr_class	HTML Parsing	No. occurrences of the class attribute

Table A.6 (*continued*)

Index	Feature	From	Description
17	html_attr_action	HTML Parsing	No. occurrences of the action attribute
18	html_attr_background	HTML Parsing	No. occurrences of the background attribute
19	html_attr_href	HTML Parsing	No. occurrences of the href attribute
20	html_attr_src	HTML Parsing	No. occurrences of the src attribute
21	html_attr_http-equiv	HTML Parsing	No. times using http-equiv for HTTP header parameters
22	html_event_onchange	HTML Parsing	No. times using onchange for element changes
23	html_event_onclick	HTML Parsing	No. times using onclick for user clicks
24	html_event_onfocus	HTML Parsing	No. times using onfocus for element focus
25	html_event_onload	HTML Parsing	No. times using onload for page or element loading
26	html_event_onmouse-out	HTML Parsing	No. times using onmouseover for mouse leaving
27	html_event_on-mouseover	HTML Parsing	No. times using onmouseout for mouse entering
28	html_event_onsubmit	HTML Parsing	No. times using onsubmit for form submission
29	html_length	HTML Parsing	Length of the website's source code
30	js_dom_document	JS Parsing	No. times using the document object to interact with webpage structure
31	js_dom_window	JS Parsing	No. times using the window object to interact with window and frames
32	js_dom_navigator	JS Parsing	No. times using the navigator object for browser and user system info
33	js_dom_location	JS Parsing	No. times using the location object for URL manipulation
34	js_dom_localStorage	JS Parsing	No. times using localStorage to store client-side data
35	js_dom_sessionStorage	JS Parsing	No. times using sessionStorage for session-specific data
36	js_dom_history	JS Parsing	No. times using the history object to manage the browser history
37	js_dom_console	JS Parsing	No. times using the console object for logging messages
38	js_dom_alert	JS Parsing	No. times using the alert property to display alert dialogs to users
39	js_prop_cookie	JS Parsing	No. times using the cookie property for handling webpage cookies
40	js_prop_referrer	JS Parsing	No. times using the referrer property for previous webpage URL
41	js_prop_innerHTML	JS Parsing	No. times using innerHTML for accessing HTML content
42	js_prop_textContent	JS Parsing	No. times using textContent for non-HTML text
43	js_prop_value	JS Parsing	No. times using value property for input element values
44	js_prop_href	JS Parsing	No. times using the href property containing the linked URL
45	js_prop_src	JS Parsing	No. times using the src property containing the resource to be loaded
46	js_prop_classList	JS Parsing	No. times using the classList property to manipulate CSS classes of elements
47	js_method_getAt-tribute	JS Parsing	No. times accessing the value of an attribute of an HTML element
48	js_method_setAt-tribute	JS Parsing	No. times setting the value for an attribute of an HTML element
49	js_method_write	JS Parsing	No. times writing HTML or text into the document
50	js_method_getEle-mentsByTagName	JS Parsing	No. times accessing all elements with a specific HTML tag
51	js_method_getEle-mentById	JS Parsing	No. times accessing an element by ID
52	js_method_fromChar-Code	JS Parsing	No. times converting Unicode to characters
53	js_min_length	JS Parsing	Minimum length of a string used in JavaScript
54	js_max_length	JS Parsing	Maximum length of a string used in JavaScript
55	js_define_function	JS Parsing	No. defined functions
56	js_function_calls	JS Parsing	No. times functions are called
57	js_file	JS Parsing	Check the existence of importing JavaScript files
58	js_pseudo_protocol	JS Parsing	Check the existence of Pseudo-protocols in JavaScript code
59	url_length	URL Parsing	Length of the URL
60	url_tag_script	URL Parsing	Check the existence of HTML script tag in the URL
61	url_tag_iframe	URL Parsing	Check the existence of HTML iframe tag in the URL
62	url_tag_link	URL Parsing	Check the existence of HTML link tag in the URL
63	url_tag_frame	URL Parsing	Check the existence of HTML frame tag in the URL
64	url_tag_form	URL Parsing	Check the existence of HTML form tag in the URL
65	url_tag_style	URL Parsing	Check the existence of HTML style tag in the URL
66	url_tag_video	URL Parsing	Check the existence of HTML video tag in the URL
67	url_tag_img	URL Parsing	Check the existence of HTML img tag in the URL
68	url_tag_main	URL Parsing	Check the existence of HTML main tag in the URL
69	url_tag_section	URL Parsing	Check the existence of HTML section tag in the URL
70	url_tag_article	URL Parsing	Check the existence of HTML article tag in the URL
71	url_attr_action	URL Parsing	Check the existence of HTML attribute action in the URL
72	url_attr_data	URL Parsing	Check the existence of HTML attribute data in the URL
73	url_attr_src	URL Parsing	Check the existence of HTML attribute src in the URL
74	url_event_onerror	URL Parsing	Check the existence of HTML event onerror in the URL
75	url_event_onload	URL Parsing	Check the existence of HTML event onload in the URL
76	url_event_on-mouseover	URL Parsing	Check the existence of HTML event onmouseover in the URL
77	url_number_key-words_param	URL Parsing	Check the existence of specific parameters in the URL
78	url_number_key-words_evil	URL Parsing	Malicious keywords frequency in URLs of websites
79	url_cookie	URL Parsing	Check if the URL contains document.Cookie
80	url_number_domain	URL Parsing	No. domains appearing in the URL

Data availability

Data will be made available on request.

References

[1] T.T. OWASP, Top 10 owasp in 2021, https://owasp.org/Top10/A03_2021-Injection/, 2021.
[2] Z. Liu, Y. Fang, C. Huang, J. Han, Graphxss: an efficient xss payload detection approach based on graph convolutional network, Comput. Secur. 114 (2022) 102597.
[3] J. Kaur, U. Garg, G. Bathla, Detection of cross-site scripting (xss) attacks using machine learning techniques: a review, Artif. Intell. Rev. (2023) 1–45.
[4] A. Hannousse, S. Yahiouche, M. Nait-Hamoud, Twenty-two years since revealing cross-site scripting attacks: a systematic mapping and a comprehensive survey, corr 2022, arXiv preprint, arXiv:2205.08425.
[5] M. Liu, B. Zhang, W. Chen, X. Zhang, A survey of exploitation and detection methods of xss vulnerabilities, IEEE Access 7 (2019) 182004–182016.
[6] F.M.M. Mokbal, W. Dan, W. Xiaoxi, Z. Wenbin, F. Lihua, Xgbxss: an extreme gradient boosting detection framework for cross-site scripting attacks based on hybrid feature selection approach and parameters optimization, J. Inf. Secur. Appl. 58 (2021) 102813.
[7] J.R. Tadhani, V. Vekariya, V. Sorathiya, S. Alshathri, W. El-Shafai, Securing web applications against xss and sqli attacks using a novel deep learning approach, Sci. Rep. 14 (2024) 1803.
[8] R. Alhamyani, M. Alshammari, Machine learning-driven detection of cross-site scripting attacks, Information 15 (2024) 420.

[9] A.A. Alqarni, N. Alsharif, N.A. Khan, L. Georgieva, E. Pardede, M.Y. Alzahrani, Mnn-xss: modular neural network based approach for xss attack detection, 2022.
[10] M. Krishnan, Y. Lim, S. Perumal, G. Palanisamy, Detection and defending the xss attack using novel hybrid stacking ensemble learning-based dnn approach, Digit. Commun. Netw. (2022).
[11] B. Beraat, B. Gülçiçek, Ş. Bahtiyar, A hybrid machine learning model to detect reflected xss attack, Balkan J. Electr. Comput. Eng. 9 (2021) 235–241.
[12] KF, DP, Download xss webpages at xssed.com, http://xssed.com/, 2007.
[13] I.K.T. Thajeel, K. Samsudin, S.J. Hashim, F. Hashim, Machine and deep learning-based xss detection approaches: a systematic literature review, J. King Saud Univ, Comput. Inf. Sci. (2023) 101628.
[14] G. Usha, S. Kannimuthu, P. Mahendiran, A.K. Shanker, D. Venugopal, Static analysis method for detecting cross site scripting vulnerabilities, Int. J. Comput. Sci. Inf. Secur. 13 (2020) 32–47.
[15] R. Wang, G. Xu, X. Zeng, X. Li, Z. Feng, Tt-xss: a novel taint tracking based dynamic detection framework for dom cross-site scripting, J. Parallel Distrib. Comput. 118 (2018) 100–106.
[16] P. Garrad, S. Unnikrishnan, Reinforcement learning in vanet penetration testing, Results Eng. 17 (2023) 100970.
[17] G. Lazrek, K. Chetioui, Y. Balboul, S. Mazer, et al., An rfe/ridge-ml/dl based anomaly intrusion detection approach for securing iomt system, Results Eng. (2024) 102659.
[18] M. Annabi, A. Zeroual, N. Messai, Towards zero trust security in connected vehicles: a comprehensive survey, Comput. Secur. (2024) 104018.
[19] Q.A. Al-Haija, Cost-effective detection system of cross-site scripting attacks using hybrid learning approach, Results Eng. 19 (2023) 101266.
[20] K. Santithanmanan, K. Kirimasthong, T. Boongoen, Machine learning based xss attacks detection method, in: UK Workshop on Computational Intelligence, Springer, 2023, pp. 418–429.
[21] L. Li, L. Wei, Automatic xss detection and automatic anti-anti-virus payload generation, in: 2019 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2019, pp. 71–76.
[22] Y. Zhou, P. Wang, An ensemble learning approach for xss attack detection with domain knowledge and threat intelligence, Comput. Secur. 82 (2019) 261–269, https://doi.org/10.1016/j.cose.2018.12.016, https://www.sciencedirect.com/science/article/pii/S0167404818306370.
[23] F.M.M. Mokbal, W. Dan, A. Imran, L. Jiuchuan, F. Akhtar, W. Xiaoxi, Mlpxss: an integrated xss-based attack detection scheme in web applications using multilayer perceptron technique, IEEE Access 7 (2019) 100567–100580, https://doi.org/10.1109/ACCESS.2019.2927417.
[24] M. Ghasemi, M. Zare, A. Zahedi, M.-A. Akbari, S. Mirjalili, L. Abualigah, Geyser inspired algorithm: a new geological-inspired meta-heuristic for real-parameter and constrained engineering optimization, J. Bionics Eng. 21 (2024) 374–408.
[25] M. Ghasemi, M. Zare, A. Zahedi, P. Trojovskỳ, L. Abualigah, E. Trojovská, Optimization based on performance of lungs in body: lungs performance-based optimization (lpo), Comput. Methods Appl. Mech. Eng. 419 (2024) 116582.
[26] G. Hu, Y. Guo, G. Wei, L. Abualigah, Genghis khan shark optimizer: a novel nature-inspired algorithm for engineering optimization, Adv. Eng. Inform. 58 (2023) 102210.
[27] A.E. Ezugwu, J.O. Agushaka, L. Abualigah, S. Mirjalili, A.H. Gandomi, Prairie dog optimization algorithm, Neural Comput. Appl. 34 (2022) 20017–20065.
[28] J.O. Agushaka, A.E. Ezugwu, L. Abualigah, Gazelle optimization algorithm: a novel nature-inspired metaheuristic optimizer, Neural Comput. Appl. 35 (2023) 4099–4131.
[29] A. Zeroual, F. Harrou, Y. Sun, Predicting road traffic density using a machine learning-driven approach, in: 2021 International Conference on Electrical, Computer and Energy Technologies (ICECET), IEEE, 2021, pp. 1–6.
[30] J. Kumar, A. Santhanavijayan, B. Rajendran, Cross site scripting attacks classification using convolutional neural network, in: 2022 International Conference on Computer Communication and Informatics (ICCCI), IEEE, 2022, pp. 1–6.
[31] L. Lei, M. Chen, C. He, D. Li, Xss detection technology based on lstm-attention, in: 2020 5th International Conference on Control, Robotics and Cybernetics (CRC), IEEE, 2020, pp. 175–180.
[32] S. Abaimov, G. Bianchi, Coddle: code-injection detection with deep learning, IEEE Access 7 (2019) 128617–128627.
[33] W.B. Shahid, B. Aslam, H. Abbas, S.B. Khalid, H. Afzal, An enhanced deep learning based framework for web attacks detection, mitigation and attacker profiling, J. Netw. Comput. Appl. 198 (2022) 103270.
[34] H. Maurel, S. Vidal, T. Rezk, Statically identifying xss using deep learning, Sci. Comput. Program. 219 (2022) 102810.
[35] F. Harrou, A. Zeroual, F. Kadri, Y. Sun, Enhancing road traffic flow prediction with improved deep learning using wavelet transforms, Results Eng. (2024) 102342.
[36] F. Harrou, A. Zeroual, M.M. Hittawe, Y. Sun, Chapter 6 - recurrent and convolutional neural networks for traffic management, in: F. Harrou, A. Zeroual, M.M. Hittawe, Y. Sun (Eds.), Road Traffic Modeling and Management, Elsevier, 2022, pp. 197–246.
[37] N. Tendikov, L. Rzayeva, B. Saoud, I. Shayea, M.H. Azmi, A. Myrzatay, M. Alnakhli, Security information event management data acquisition and analysis methods with machine learning principles, Results Eng. 22 (2024) 102254.
[38] B. Buz, B. Gülçiçek, Ş. Bahtiyar, A hybrid machine learning model to detect reflected xss attack, Balkan J. Electr. Comput. Eng. 9 (2021) 235–241.
[39] R. Banerjee, A. Baksi, N. Singh, S.K. Bishnu, Detection of xss in web applications using machine learning classifiers, in: 2020 4th International Conference on Electronics, Materials Engineering & Nano-Technology (IEMENTech), IEEE, 2020, pp. 1–5.
[40] S. Kascheev, T. Olenchikova, The detecting cross-site scripting (xss) using machine learning methods, in: 2020 Global Smart Industry Conference (GloSIC), IEEE, 2020, pp. 265–270.
[41] Y. Pan, F. Sun, Z. Teng, J. White, D.C. Schmidt, J. Staples, L. Krause, Detecting web attacks with end-to-end deep learning, J. Internet Serv. Appl. 10 (2019) 1–22.
[42] Y. Zhou, P. Wang, An ensemble learning approach for xss attack detection with domain knowledge and threat intelligence, Comput. Secur. 82 (2019) 261–269.
[43] F.M.M. Mokbal, W. Dan, A. Imran, L. Jiuchuan, F. Akhtar, W. Xiaoxi, Mlpxss: an integrated xss-based attack detection scheme in web applications using multilayer perceptron technique, IEEE Access 7 (2019) 100567–100580.
[44] Y. Fang, Y. Li, L. Liu, C. Huang, Deepxss: cross site scripting detection based on deep learning, in: Proceedings of the 2018 International Conference on Computing and Artificial Intelligence, 2018, pp. 47–51.
[45] S. Rathore, P.K. Sharma, J.H. Park, Xssclassifier: an efficient xss attack detection approach based on machine learning classifier on snss, J. Inf. Process. Syst. 13 (2017) 1014–1028.
[46] C. Umbrella, Download begin webpages at umbrella popularity list, https://s3-us-west-1.amazonaws.com/umbrella-static/index.html, 2016.
[47] A. Farahat, F. Effenberger, M. Vinck, A novel feature-scrambling approach reveals the capacity of convolutional neural networks to learn spatial relations, Neural Netw. 167 (2023) 400–414.
[48] T. Trinh, A. Dai, T. Luong, Q. Le, Learning longer-term dependencies in rnns with auxiliary losses, in: International Conference on Machine Learning, PMLR, 2018, pp. 4965–4974.
[49] T. Developers, Tensorflow, Zenodo, 2022.
[50] A. Géron, Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow, O'Reilly Media, Inc., 2022.
[51] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, Optuna: a next-generation hyperparameter optimization framework, in: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2019, pp. 2623–2631.
[52] S. Shekhar, A. Bansode, A. Salim, A comparative study of hyper-parameter optimization tools, in: 2021 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE), IEEE, 2021, pp. 1–6.
[53] P. Pokhrel, A comparison of AutoML hyperparameter optimization tools for tabular data, Ph.D. thesis, 2023.
[54] S. Hanifi, A. Cammarono, H. Zare-Behtash, Advanced hyperparameter optimization of deep learning models for wind power prediction, Renew. Energy 221 (2024) 119700.