Scientific Research Publishing

Journal of Intelligent Learning Systems and Applications, 2026, 18(3), 216-234
https://www.scirp.org/journal/jilsa
ISSN Online: 2150-8410
ISSN Print: 2150-8402

Enhancing Cross-Site Scripting (XSS) Attacks Detection through Modern Transformer Architecture Optimizations

Emil Wangilisasi, Judith Leo, Anael Sam

School of Computational and Communication Science and Engineering, Nelson Mandela African Institution of Science and Technology (NM-AIST), Arusha, Tanzania
Email: emily.wangilisasi@nm-aist.ac.tz, judith.leo@nm-aist.ac.tz, anael.sam@nm-aist.ac.tz

How to cite this paper: Wangilisasi, E., Leo, J. and Sam, A. (2026) Enhancing Cross-Site Scripting (XSS) Attacks Detection through Modern Transformer Architecture Optimizations. *Journal of Intelligent Learning Systems and Applications*, **18**, 216-234.
https://doi.org/10.4236/jilsa.2026.183014

Received: May 16, 2026
Accepted: June 28, 2026
Published: July 1, 2026

Copyright © 2026 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

Cross-Site Scripting (XSS) remains a widespread and damaging threats to web applications, as highlighted by the OWASP Top 10. While various detection methods exist, they often struggle to keep pace with the sophistication of attack vectors and obfuscation techniques. This paper implements an approach for enhancing XSS detection by leveraging modern optimizations within the transformer architecture. Our methodology uses a custom transformer encoder model trained on an aggregated dataset of nearly 100,000 samples, including newly collected XSS payloads. To enhance performance and efficiency, we integrate two key architectural improvements: Rotary Positional Embeddings (RoPE) to achieve a superior contextual understanding of HTTP payloads, and Flash Attention to significantly accelerate training and inference speeds while reducing memory consumption. Experimental results show good performance of our model that achieves an accuracy of 99.38 with high recall and precision. An ablation study demonstrates that the integrated optimizations improve detection accuracy by 0.11 percentage points, while reducing training time by approximately 32% and peak GPU memory usage by approximately 30% relative to standard Transformer configurations. Comparative evaluation against a Random Forest baseline further reveals a clear contextual understanding advantage over traditional frequency-based approaches, justifying the architectural complexity. The proposed model effectively captures structural patterns in sophisticated payloads that typically evade classical methods. This work presents an efficient, and accurate solution in real-time XSS threat detection.

Keywords

Cross-Site Scripting (XSS), Transformer-Based Detection, Rotary Positional

Embeddings, Flash Attention, Deep Learning

1. Introduction

The rapid increase of web applications, mobile applications and online services has made safeguarding data and ensuring secure user interactions a critical priority. Among many threats in the cyber landscape, Cross-Site Scripting (XSS) vulnerabilities stand out as one of the most pervasive and damaging. According to the Open Web Application Security Project (OWASP) [1], XSS attacks rank among the top ten most common web application security risks. OWASP is a global community and nonprofit organization focused on improving software security. These vulnerabilities can expose sensitive information, compromise user privacy, and enable malicious actors to execute arbitrary code on web applications, potentially leading to catastrophic consequences. Figure 1 shows the OWASP top ten vulnerabilities for the year 2025. This list is updated after every four years.

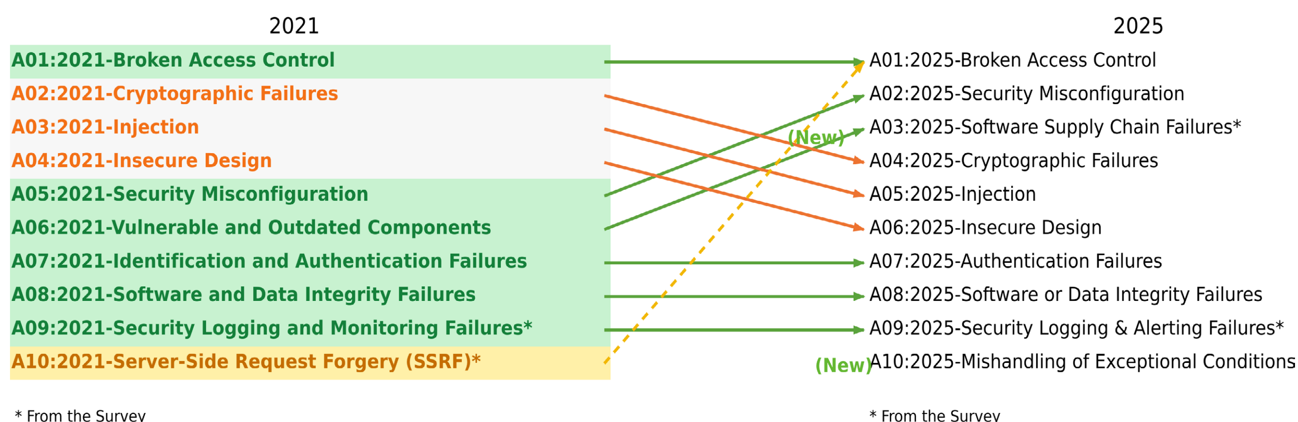

Figure 1. OWASP top 10 web application vulnerabilities [1].

XSS attacks occur when scripts with malicious intent are inserted into pages of a web application that are viewed by users. These scripts can be crafted to steal sensitive user data, deface websites and perform ill-intended actions on behalf of users without the user's consent. The consequences of XSS attacks are severe, leading to compromised user accounts, data breaches, and damaged reputations for affected organizations. Over the years, researchers have delved deep into different categories of XSS attacks, such as reflected, stored and DOM-based XSS [2]. They have also explored mitigation techniques ranging from validation of the inputs to coding practices that emphasize security and content security policies. Understanding the evolution of XSS attacks and the countermeasures devised to mitigate them is crucial for the ongoing efforts to secure web applications against this persistent threat. Figure 2 shows how an XSS attack occurs.

There have been many security mechanisms to detect and prevent XSS attacks.

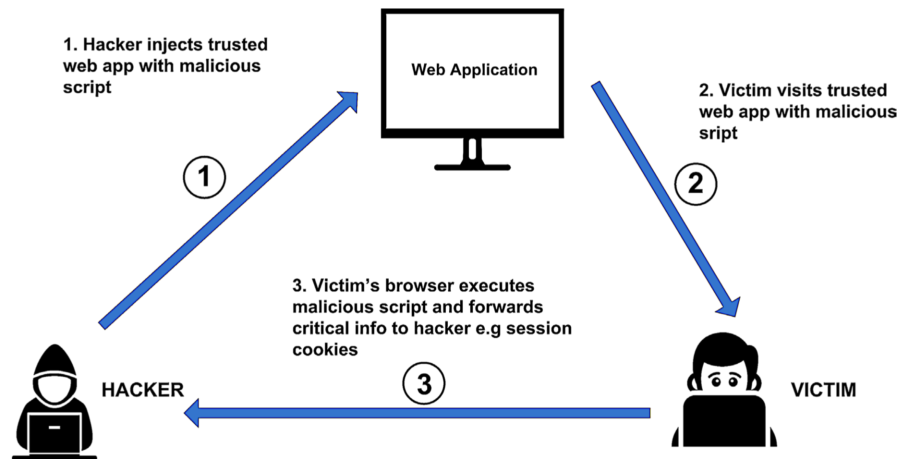

Figure 2. Cross site scripting attack (XSS).

Such mechanisms include dynamic analysis, static analysis hybrid analysis, traditional machine learning approaches, and approaches based on deep learning. Despite the relative success of these methods in mitigating the threat of XSS attacks, they have failed to completely counter the attacks due to the evolving sophistication of XSS attacks. Attackers have continually devised new evasion techniques and obfuscation methods to bypass these defenses, making it imperative for the cybersecurity community to explore novel, intelligent, and adaptive solutions.

The objective of this work is to apply recent advances in Natural Language Processing (NLP), particularly Transformer-based techniques, to improve the detection of XSS attacks across multiple performance metrics. The main contributions of this study are twofold. First, recent XSS data was collected to expand the available datasets used for training and testing the proposed model. Second, state-of-the-art techniques, including **Flash Attention** for improved processing speed and **RoPE positional encoding** for better contextual understanding of HTTP payloads, were incorporated to enhance the classification of malicious and benign traffic.

2. Related Work

Several research methods have been introduced in recent years to detect cross-site scripting vulnerabilities. Fang *et al.* [3] developed DeepXSS, a tool based on deep learning for detecting XSS attacks. Their approach applied the word2vec algorithm to extract features that capture word-order information from XSS payloads, after which each payload was represented as a corresponding feature vector. The detection model was then trained and evaluated using Long Short-Term Memory (LSTM) which is a type of recurrent neural networks (RNNs). For future work, the authors suggested collecting additional XSS attack datasets and experimenting with other deep learning algorithms.

Yan *et al.* [4] introduced a model based on convolutional neural networks (CNN) for detecting XSS attacks, incorporating a modified ResNet block to improve detection performance. Their key contribution was a URL preprocessing

method designed around the syntactic and semantic features of encoded XSS attack scripts. They also enhanced the residual module of ResNet to extract features from three perspectives and replaced the layer that is fully connected with a structure that uses 1 × 1 convolution features. For future work, the authors suggested exploring the use of their model in other web vulnerability detection and mining tasks, such as cross-site request forgery, buffer overflow and SQL injection.

In [5], traditional machine learning techniques were applied to detect XSS attacks in web applications. The authors analyzed various algorithms, such as support vector machines, decision trees, Naive Bayes, and logistic regression. For future work, they suggested addressing the dataset imbalance issue by using generative methods to create a more balanced dataset.

Gated Recurrent Units (GRU) were used to detect malicious Uniform Resource Locators (URLs) that may contain injection attacks such as XSS in [6]. Their study used characters as text classification features. For future work the authors propose further research in optimization of their model to reduce memory usage while maintaining the test results.

Zhou and Wang [7] developed an ensemble-based tool for detecting XSS attacks. Their method employed multiple Bayesian networks, each constructed using domain knowledge and threat intelligence to improve detection accuracy. They also developed a method to make their results explainable to end users. For future work, the authors aim to test their method with different datasets and scenarios that are more practical as well as integrating the method in an operational web application security risk assessment system. **Table 1** shows a summary of related studies.

Table 1. Related studies on XSS attacks detection.

Author	Year	Method	Future Work/Limitation
Fang *et al*	2018	Long Short Term Memory (LSTM)	Collect more XSS datasets and use other DL algorithms
Yan *et al.*	2022	CNN and modified ResNet	Expand method to other attacks such as cross-site request forgery and SQL injection
Kascheev and Olenchikova	2020	Classical Machine Learning	Use generative methods to generate a balanced dataset.
Yang *et al.*	2019	Gated Recurrent Units (GRUs)	Optimization to reduce memory usage while maintaining good test results
Zhou and Wang	2019	Machine learning ensemble-based approach	Test method with more datasets. Integrate method in areal web application security system

While existing methods effectively identify established XSS attack patterns, they often struggle to detect novel attack vectors. Our work presents an approach that analyzes the semantic structure of XSS patterns by leveraging recent advances in transformer architectures. This semantic understanding enables better detection of previously unseen attacks.

3. Materials and Methods

3.1. Dataset

In this study we have one dataset with 98,008 total samples, of which 49,532 are malicious and 48,476 are benign samples. This dataset was collected in five ways. The first dataset is a primary synthetic dataset that we created by employing the XSStrike tool against the intentionally vulnerable OWASP Juice Shop application, with the resulting logs captured using Burp Suite. The second dataset is secondary dataset obtained from Kaggle [8], containing 13,676 examples of which 6313 are benign traffic and 7363 are malicious XSS traffic. The third dataset was obtained from the official OWASP Github repository [9] which contains a list of up-to-date XSS payloads submitted by security researchers. The OWASP repository is up to date and well maintained. The fourth source was created by crawling the XSSed website [10] for malicious XSS samples while benign samples were collected by simulating normal browsing activity across legitimate websites and collecting the browsing traffic as normal payload. The fifth dataset consisted of live malicious traffic samples collected over a 90-day period using a T-Pot honeypot deployed on a Virtual Private Server (VPS). Snare and Tanner which are part of T-Pot were used to capture real interaction attempts and web-based attack traffic in an Internet-exposed environment. This approach provided realistic samples of live malicious activity beyond the static payloads in datasets 2 and 3, thereby exposing the model to attack patterns observed in operational settings. The five datasets were then combined and then the entire consolidated dataset was deduplicated before train-test-split to prevent the problem of data leakage which often occurs when identical samples appeared in both the training and evaluation sets [11]. The resulting single dataset was then used to train our model. **Figure 3** shows a snapshot of a sample of the dataset while **Table 2** summarizes the contribution of each source and collection route to the consolidated dataset.

3.1.1. Deduplication and Leakage Control

The consolidated dataset was deduplicated before splitting to prevent identical payloads from appearing in both training and evaluation subsets. Exact duplicate

19013	<input onmouseout="alert(1)">test</input>	1
14890	INTNAV=fNav:TermsOfUse	0
46944	pg=asdf<script>alert(1337)</script>&frame=1	1
60314	hl=en&ref_topic=3544742,2986333,	0
75736	cat=1"></title><script>alert(1337)</script>"...	1
88645	<title onpointerleave=alert(1)>XSS</title>	1
25240	<svg><strong onload=alert(1)>	1
72359	According to <a href="/wiki/CNN" title...	0

Figure 3. A snapshot of part of our XSS dataset.

Table 2. Final dataset composition by source and class label.

Source	Benign	Malicious	Total	Notes
XSStrike + OWASP Juice Shop via Burp Suite	10,000	15,507	25,507	Controlled synthetic attacks and benign application traffic
Kaggle XSS dataset	6313	7363	13,676	Public labeled dataset
OWASP GitHub repository	0	4500	4500	Curated XSS payloads
XSSed crawl	0	17,659	17,659	Crawled malicious samples
Benign browsing simulation	32,163	0	32,163	Normal browsing traffic
T-Pot/SNARE/Tanner honeypot	0	4503	4503	Live malicious web traffic
Total	**48,476**	**49,532**	**98,008**	**Composite dataset**

strings were removed after whitespace normalization, and encoded variants were normalized through safe decoding before duplicate checks where applicable. Although this procedure reduced direct leakage between partitions, the evaluation used a random stratified split of the merged dataset rather than a source-wise or external holdout. Therefore, transformed variants of the same base payload may still remain across partitions. Source-wise and external-holdout evaluation are identified as important directions for future validation.

The distribution of payload lengths in our dataset shows significant variability, as illustrated in Figure 4. Statistical analysis reveals a median payload length of 43 characters and a 75th percentile of 73 characters. These metrics are crucial for determining the optimal context length for our model architecture, as they indicate that the majority of payloads fall within this range.

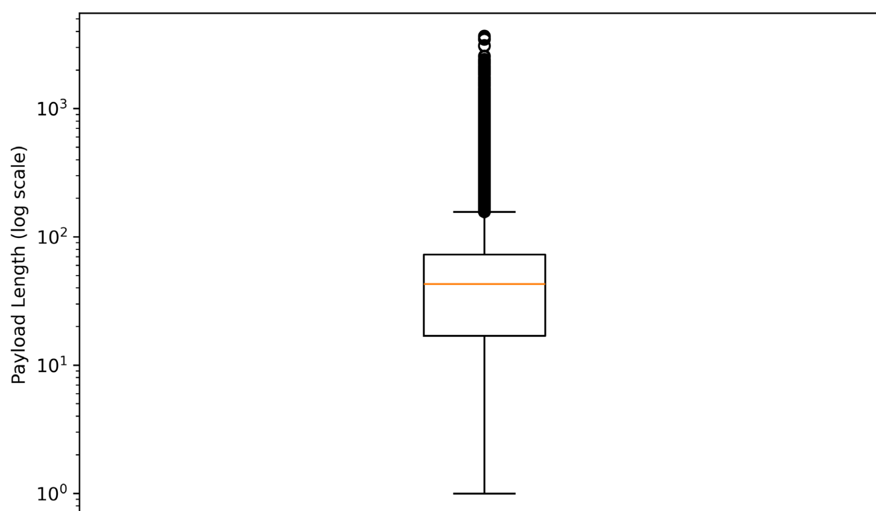

Figure 4. Distribution of payload lengths (in characters) across the dataset displayed on a logarithmic scale (n = 98,008).

To further validate the quality and discriminative nature of our dataset, we performed an analysis based on the t-distributed Stochastic Neighbor Embedding (t-

SNE) to visualize the high-dimensional feature space in two dimensions, as shown in Figure 5. For this analysis, we employed a stratified random sample of 20,000 instances from the complete dataset to ensure computational tractability while maintaining the original class distribution. Feature extraction was performed using Term Frequency-Inverse Document Frequency (TF-IDF) vectorization with character-level n-grams (n = 3 - 5), which effectively captures the syntactic patterns and special character sequences characteristic of XSS payloads. The t-SNE algorithm was configured with a perplexity of 30 and executed for 1000 iterations to achieve stable embeddings. The resulting visualization demonstrates some overlap in the feature space indicating challenging cases. This spatial distribution provides insights into the complexity of the classification task and validates the necessity of employing deep learning approaches capable of learning nuanced distinctions between legitimate HTTP traffic and XSS attack patterns.

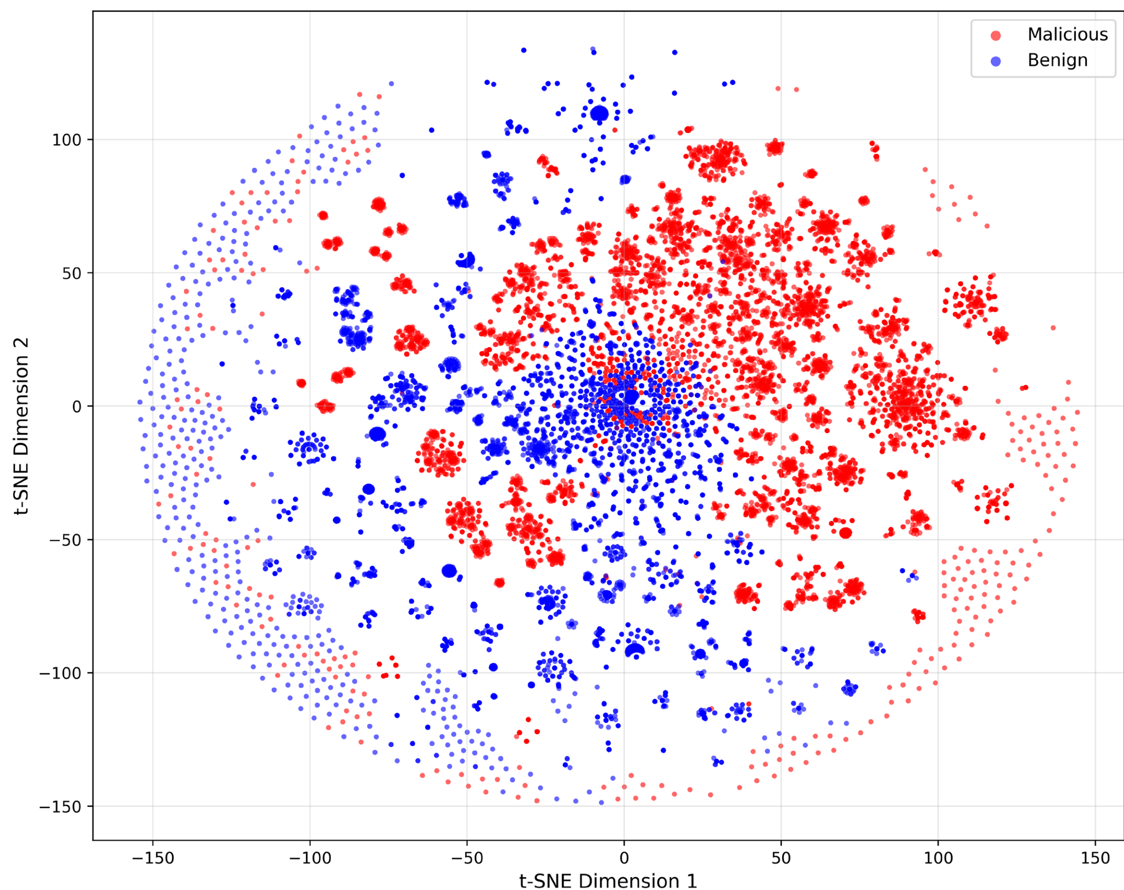

Figure 5. t-SNE visualization of malicious and benign samples based on character-level TF-IDF features.

3.1.2. Dataset Split and Model-Selection Protocol

After preprocessing and deduplication, the dataset was first split using an 80:20 train-test ratio, producing 78,406 development samples and 19,602 held-out test samples. The development portion was then divided further into training and validation subsets, with 10% reserved for validation. This produced an effective

72:8:20 train-validation-test protocol, consisting of 70,565 training samples, 7841 validation samples, and 19,602 test samples. All splits underwent stratification by class label to preserve the benign/malicious distribution, and a random seed that was fixed at 42 was used for reproducibility. The set for validation was used for model selection, hyperparameter tuning, and early stopping. During training, monitoring of validation loss was done with a patience of five epochs and an improvement threshold was set to a minumum value of 0.001. The held-out test set was used only for the final performance evaluation.

3.2. System Pipeline

A detailed overview of the proposed system pipeline is presented in this section, which uses deep neural networks built on the Transformer architecture. The proposed approach detects Cross-Site Scripting (XSS) attacks by applying deep learning techniques for text classification.

The input data is first processed through several preprocessing stages, including decoding and tokenization. The resulting tokenized sequences are then passed into the Transformer model, which is trained to classify each sample as either an XSS attack or a benign input. The overall architecture of the proposed method is depicted in Figure 6.

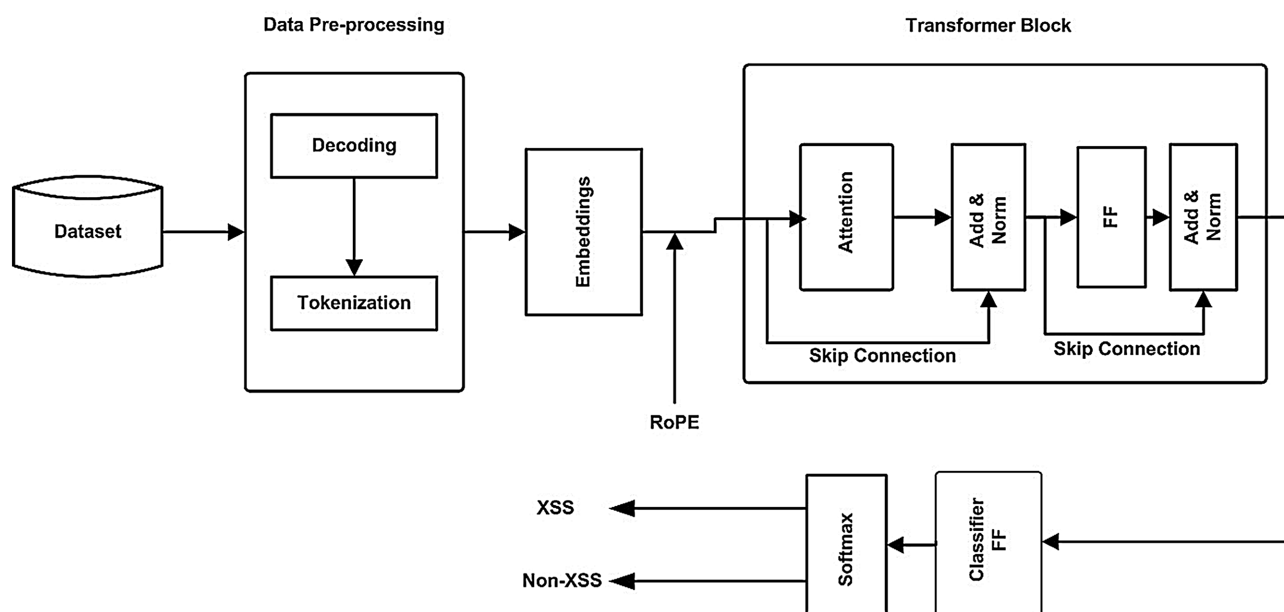

Figure 6. Overall system flow.

3.3. Data Preprocessing

One important step in preprocessing is payload decoding. Attackers frequently encode their malicious queries in a format such as URL, HTML or base64 encoding in order to bypass detecting mechanisms based on pattern matching [12]. This obfuscation can make even a straightforward payload appear harmless to simple filters. For instance, this cross-site scripting (XSS) payload, <DIV STYLE="back-

ground-image: url(javascript:alert('XSS'))">, that uses a CSS background-image style to execute JavaScript can be encoded in several ways as shown in **Table 3**.

Table 3. Encoding mechanisms.

Encoding	Obfuscated Payload
URL	%3CDIV%20STYLE%3D%22background-image%3A%20url(javascript%3Aalert(%27XSS%27))%22%3E
HTML	<DIV STYLE="background-image: url(javascript:alert('XSS')) ">
Base64	PERJViBTVFlMRT0iYmFja2dyb3VuZC1pbWFnZTogdXJsKGphdmFzY3JpcHQ6YWxlcnQoJ1hTUycpKSI+

In this work we make sure to decode all payloads before they are fed to the network for training. This step is important for ensuring the model can recognize the fundamental malicious pattern, regardless of the encoding scheme used to obfuscate it.

Another preprocessing stage used in this work is tokenization. Tokenization is a natural language processing technique that involves dividing text into smaller units known as tokens. Depending on the task and the algorithm applied, these tokens may represent complete words or sub-word units. In this study, several tokenization algorithms were tested, and the Byte Pair Encoding (BPE) algorithm [13] achieved the best results.

Decoding order, BPE vocabulary, and sequence length handling: Payload decoding was applied before tokenization. Multi-encoded payloads were decoded in a fixed order following the pattern of firstly URL decoding then HTML entity decoding, Unicode escape decoding, and Base64 decoding when the input matched a valid Base64 pattern. Decoding was applied iteratively for up to three passes and stopped earlier when the string no longer changed. This prevented single-pass decoding from missing nested encodings while avoiding unbounded transformation loops.

After decoding, payloads were tokenized using Byte Pair Encoding (BPE) with a vocabulary size of 16,000. Inputs longer than the maximum sequence length of 512 tokens were truncated, while shorter inputs were padded using a [PAD] token. Padding positions were masked so that they did not contribute to attention or loss computation.

3.4. Generating Token Embeddings

After tokenization, the tokens must be transformed into dense vector representations within a high-dimensional space. These embeddings are learnable parameters that can either be trained jointly with the model for a specific task or obtained from large text corpora using pre-trained methods such as Word2Vec or GloVe [14]. The quality of these embeddings plays an important role in the performance of NLP models, as they help the model generalize learned patterns to unseen text

by capturing relevant linguistic features. In this work, pre-trained embeddings are not used. Instead, the model relies on learned embeddings, where the embedding vectors are randomly initialized at the start of training and updated throughout the training process.

3.5. Training with the Transformer

The transformer architecture has revolutionized Natural Language Processing. It was introduced in 2017 in the landmark paper Attention is All You Need [15]. The transformer architecture dispenses with recurrence and convolutions and relies entirely on the attention mechanism. The Transformer architecture models the dependencies between the inputs and is more parallelizable and requires significantly less time to train. Previous sequence to sequence models like RNNs had the problem of vanishing gradients when running the backpropagation algorithm [16]. The Transformer unlike RNNs has no problem of vanishing gradients and can model longer sequences. In this work we use the encoder part of the original transformer and have incorporated modern optimizations of the transformer architecture that make the network more efficient in memory usage and faster both during training and inference.

Position Encoding: Unlike earlier architectures such as RNNs, which process words sequentially, the attention mechanism in Transformer models does not naturally capture word order within a sentence. Therefore, positional information must be explicitly added to the input sequences before they are passed to the Transformer's attention mechanism. In this work, we use Rotary Positional Embeddings (RoPE), a modern positional encoding technique [17]. RoPE introduces position-dependent rotations to the query and key vectors used in the attention mechanism. One key advantage of RoPE is its ability to generalize well to sequences that are longer than those encountered during training. As a result, models can be trained on shorter sequences while still supporting longer sequences during inference, reducing training time and computational cost [18].

To rotate a vector by an angle θ we use a rotation matrix R_θ. Each of k and q vectors are multiplied by this rotation matrix. The angle of rotation θ is dependent in the position of the token in the sequence.

$$R_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{1}$$

For a position t in the input sequence, RoPE rotates each pair of dimensions in the query and key vectors defined as:

$$q_t = \left[q_t^{(1)}, q_t^{(2)}, \cdots, q_t^{(d_q-1)}, q_t^{d_q} \right] \tag{2}$$

$$k_t = \left[k_t^{(1)}, k_t^{(2)}, \cdots, k_t^{(d_k-1)}, k_t^{d_k} \right] \tag{3}$$

d_k and d_q are the dimensionality of the query and key vectors. RoPE rotates pairs of dimensions indexed as $(2p, 2+1)$, where each pair's index p ranges from

0 to $d/2$.

Each pair p undergoes a rotation based on the token position t and a rotation frequency θ:

$$\text{RoPE}\left(q_t\left(p\right)\right)=\begin{bmatrix}\cos\left(\theta_p t\right) & -\sin\left(\theta_p t\right)\\ \sin\left(\theta_p t\right) & \cos\left(\theta_p t\right)\end{bmatrix}\begin{bmatrix}q_t^{(2p-1)}\\ q_t^{(2p)}\end{bmatrix} \tag{4}$$

θ_p is the rotation frequency for the pth pair. It is defined as:

$$\theta_p=\frac{1}{\Theta^{2p/d_q}} \tag{5}$$

Θ is a constant hyperparameter usually set to a 10,000. However larger values such as 500,000 have shown to boost performance of models to handle higher context lengths.

In this study, we use a value of Θ = 50,000 as this gives optimal results for our dataset.

Self-attention mechanism: This is one of the main innovations of the Transformer architecture. It enables tokens within a sequence to interact with one another, allowing each token representation to be generated based on its relationship with the other tokens in the same context window. As a result, each output representation is formed by combining information from all tokens in the sequence using learned attention weights. In the original Transformer model, this process connects different positions within a single sequence to produce a meaningful representation of that sequence [15].

The input $X\in\mathrm{R}^{(n\times d)}$ is first projected into queries, keys and values via $Q=XW^Q$, $K=XW^K$ and $V=XW^V$. The resulting representations are then used to compute the attention weights.

$$\text{Attention}\left(Q,K,V\right)=\text{softmax}\left(QK^{\mathrm{T}}\big/\sqrt{d_k}\right)V \tag{6}$$

The value d_k is a hyperparameter that represents the dimensionality of the key and query vectors.

Flash Attention: This is an algorithm designed to make the attention mechanism faster and more memory-efficient, particularly for long sequences [19]. Standard attention mechanisms can be computationally intensive and memory-hungry due to the need to materialize and store the large $N\times N$ attention matrix, where N is the sequence length. Flash Attention addresses this bottleneck by restructuring the attention computation. It leverages tiling and recomputation techniques to compute the exact attention output without ever forming the full attention matrix in GPU High Bandwidth Memory (HBM). This approach significantly reduces the reading and writing to and from memory between GPU HBM and on-chip SRAM, leading to substantial speedups and a reduction in memory usage. The efficiency gains from Flash Attention are crucial for training and deploying large transformer models on longer context windows, making it a vital component in modern LLM architectures. In this work, we have incorporated flash attention via the Pytorch flash attention library, which has significantly sped up both train-

ing and inference of our detection model.

Feedforward Layer: The position-wise feedforward layer in the Transformer applies two linear transformations with a ReLU activation independently to each token's representation (Vaswani *et al.*, 2017) [15]. Given an input vector $x \in \mathrm{R}^d$, the layer computes

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{7}$$

where, $W_2 \in \mathrm{R}^{d_{ff} \times d}$, and b_1, b_2 are bias vectors. By acting independently on each position, this component both expands the model's capacity for capturing complex features and preserves parallelizability across sequence elements. In this work we set the intermediate dimension $d_{ff} = 2048$ in all feedforward layer, and observed that this choice balanced expressive power with training efficiency.

Dropout: Dropout is a technique for reqularization that is designed to reduce overfitting in neural networks [20]. During the training phase, for each forward pass, a subset of neurons within a given layer is stochastically deactivated, or "dropped", with a predefined probability p. This temporary removal of units prevents co-adaptations, wherein neurons become too reliant on the presence of specific other neurons, thereby encouraging the learning of more robust and independently useful features. Consequently, the network effectively trains an ensemble of numerous "thinned" networks with shared weights. At inference time, dropout is typically disabled, and the full network is utilized. In this work we used dropout with a rate of 0.2 to train our transformer. This value was obtained through experimentation.

Regularization/Weight Decay: Regularization techniques are essential for constraining model complexity and improving generalization by discouraging overly large parameter values [21]. A common approach is L2 regularization, or weight decay, which augments the loss function L with a penalty proportional to the squared norm of the weights:

$\mathcal{L}_{\text{total}} = \mathcal{L} + \lambda \sum_i w_i^2$ where λ is the weight decay coefficient. In the context of adaptive optimizers, we employ the AdamW algorithm [22], which decouples the weight decay term from the gradient-based update to avoid the unintended interaction between momentum and L_2 penalty. By applying weight decay throughout training, our model learns more robust feature representations and exhibits reduced overfitting, as evidenced by a narrower gap between training and validation losses. In this work, we used weight decay coefficient $\lambda = 0.01$.

Early Stopping: Early stopping is a technique for technique that is used to reduce overfitting and improve the generalization ability of deep learning models by monitoring performance during training on a validation set and stopping the process when validation performance no longer improves or begins to decline. This prevents the model from memorizing the training data and encourages it to learn patterns that generalize well to unseen data. Two key parameters used in early stopping are patience and min_delta. The patience parameter specifies the number of consecutive epochs training can continue without improvement before

stopping, while min_delta defines the minimum change in the monitored metric, such as validation loss or accuracy, required to be considered a meaningful improvement. If the validation metric improves by less than the specified min_delta value for the number of epochs defined by patience, training is stopped. In this study, the optimal values for patience and min_delta were empirically determined as 5 epochs and 0.001, respectively. **Table 4** summarizes the parameters and hyperparameters used in the proposed model.

Table 4. Summary of model parameters and hyperparameters.

Parameter/ Hyperparameter	Value	Description
d_model	256	The dimensionality of embeddings
depth	4	The number of identical transformer layers stacked together
num_heads	4	The number of parallel attention heads in multi-head attention
d_ff	1024	The dimensionality of the inner layer in the feed-forward network.
max_seq_length	512	The maximum input sequence length
warmup_steps	10,000	The number of initial steps for the learning rate warmup
batch_size	32	The number of sequences processed per training step
dropout_rate	0.2	The probability of dropping out units during training to prevent overfitting
weight_decay	0.01	The L2 regularization penalty applied by the optimizer.
learning_rate	0.0001	The base step size for the optimizer

Loss Function: The loss function which is also known as the objective function, is a key component of model training because it measures the difference between the model's predictions and the ground-truth labels [21]. Its value provides the main signal for the optimization algorithm, which updates the model's parameters, such as weights and biases, through backpropagation in order to minimize prediction error. Selecting a suitable loss function is essential and depends on the type of machine learning task being addressed. Binary Cross-Entropy (BCE) loss is a function that is commonly used for binary classification tasks, as it measures the difference between the true label distribution, where each label is either 0 or 1, and the model's predicted probability distribution. The BCE loss for a single prediction is defined by the formula:

$$\text{Loss} = -\left[y \log\left(\hat{y}\right) + \left(1 - y\right) \log\left(1 - \hat{y}\right) \right] \tag{8}$$

where y is the real label (0 or 1) and $\hat{y}$ is the predicted probability from the model of the class being 1. In this study, we employed the binary cross-entropy loss function to train the network, as our objective is to solve a binary classification problem of XSS attacks being either benign or malicious.

4. Results

To perform evaluation of the trained model, the test set was used to measure the model's capability to generalize to unseen data. The model was assessed using evaluation metrics such as precision, accuracy, F1-score and recall, which indicate how effectively it classified inputs as either XSS attacks or benign traffic. In addition, a confusion matrix and classification report were generated to provide a more detailed analysis of the results, including true positives, true negatives, false positives, and false negatives. The dataset was composed of 98,008 samples and was split using an 80:20 ratio, producing 78,406 training samples and 19,602 testing samples. The trained model was then evaluated on the testing set using the selected performance metrics.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{9}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{10}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{11}$$

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{12}$$

On the test set of 19,602 examples, the model achieved 9811 true positives and 9676 true negatives, with only 19 false positives and 96 false negatives. This corresponds to an accuracy of 0.9938, precision of 0.9902 for the negative class and 0.9981 for the positive class while the recall is 0.9980 for the negative class and 0.9903 for the positive class. This shows good performance of our model in detecting XSS attacks. Table 5 and Figure 7 depict the classification report and the confusion matrix respectively.

Error analysis: A manual review of the misclassified samples was conducted to characterize the remaining false negatives and false positives. Based on the confusion matrix, the test set contained 19 false positives and 96 false negatives. Table 6 shows a summary of the errors and their characterization.

False positives were primarily benign inputs containing JavaScript-like or HTML-like syntax, such as developer documentation, search queries, or form

Table 5. Classification report.

	Precision	Recall	F1-Score	Support
0	0.9902	0.9980	0.9941	9695
1	0.9981	0.9903	0.9942	9907
Accuracy			0.9938	19,602
Macro Avg	0.9941	0.9942	0.9941	19,602
Weighted Avg	0.9942	0.9941	0.9941	19,602

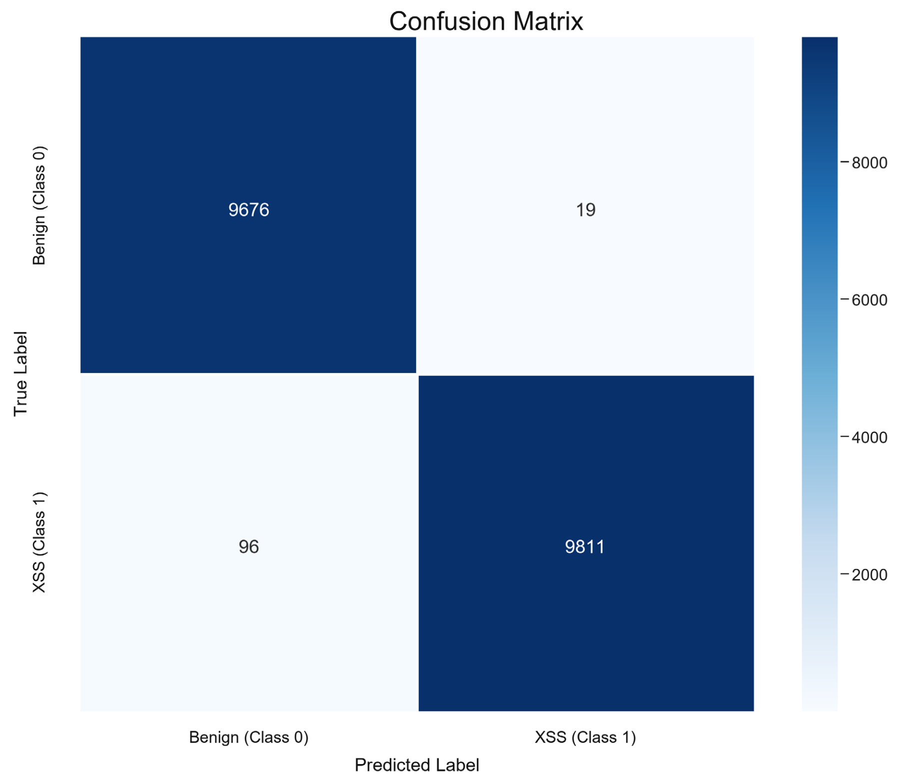

Figure 7. Confusion matrix.

Table 6. Summary of error analysis.

Error Type	Count	Observed pattern
False Positives	19	Benign JavaScript/HTML snippets, technical queries containing <, >, script, event-handler-like strings, or special characters in normal form submissions
False Negatives	96	Heavily encoded payloads, short payloads with few lexical markers, DOM-based XSS requiring application context, or uncommon obfuscation patterns

values with special characters. False negatives were mainly heavily encoded or context-dependent XSS payloads whose malicious behavior depended on browser or application execution context rather than obvious lexical markers. These errors suggest that future improvements should combine payload-level detection with contextual request and response analysis.

4.1. Ablation Study

An ablation study was also done to measure the effects of RoPE and Flash Attention on the detection of XSS injection. The baseline transformer (without RoPE or Flash Attention) achieved 99.30% accuracy with an average training time 74.48 ms. Replacing the learned position embeddings with RoPE raised accuracy to 99.38%. Incorporating Flash Attention reduced the average training time to 50.34 ms and lowered peak GPU memory usage on an NVIDIA A100 from 12.4 GB to

8.7 GB (−30%). Table 7 gives a summary of the results of the ablation study. When combined, RoPE and Flash Attention delivered both accuracy and efficiency. Incorporating RoPE enhanced contextual understanding of HTTP payloads, and Flash Attention delivered speed and memory gains.

Table 7. Summary of results of ablation study.

Model Variant	RoPE	Flash Attention	Accuracy (%)	Peak GPU Time (ms)	Peak GPU Memory
Baseline Transformer	✗	✗	99.27	74.48	12.4
+RoPE	✓	✗	99.38	74.48	12.4
+Flash Attention	✗	✓	99.27	50.34	8.7
RoPE + Flash Attention	✓	✓	99.38	50.34	8.7

Timing metric and inference latency: The timing values in the ablation study report the average training step time per batch, measured in milliseconds on an NVIDIA A100 GPU with a batch size of 32. The reported reduction from 74.48 ms to 50.34 ms therefore reflects training-step acceleration rather than end-to-end inference latency.

Model-only inference latency was also measured under the same hardware setup using batch size 1 to approximate single-request deployment behavior. These latency values represent model-only GPU inference after preprocessing and tokenization, end-to-end deployment latency may be higher due to decoding, tokenization, data transfer, and application-level overhead. Table 8 summarizes the latency values.

Table 8. Summary of inference latency values (model-only).

Model variant	Batch size	Mean (ms)	P95 (ms)	Hardware
baseline Transformer	1	6.40	8.32	NVIDIA A100
+ RoPE	1	7.10	8.65	NVIDIA A100
+ Flash Attention	1	4.76	5.91	NVIDIA A100
RoPE + Flash Attention	1	4.98	6.18	NVIDIA A100

4.2. Comparison with a Baseline Classifier

To evaluate the relative performance gain of our architecture, we implemented a Random Forest (RF) classifier as a baseline on the same dataset. Random Forest is a widely used ensemble learning method in Machine Learning due to its ability to handle data of high dimensions. For this baseline, payloads were vectorized using TF-IDF (Term Frequency-Inverse Document Frequency) with character-level n-grams (n = 3 to 5). The RF model was configured with 100 trees and a maximum depth of 20 to ensure a competitive baseline. This comparison justifies the use of our approach which is more complex than classical ML solutions. Clas-

sical models such as Random Forest rely largely on statistical frequency features, whereas Transformers employ self-attention to capture semantic relationships and sequential structure within payloads. This enables improved detection of advanced, non-repetitive obfuscation patterns that typically bypass frequency-based classifiers. Table 9 shows the comparison between Random Forest and our approach.

Table 9. Comparison with baseline classifier.

Model	Accuracy	Precision	Recall	F1-Score
Random Forest	96.42	0.958	0.961	0.959
Transformer	99.38	0.994	0.994	0.994

5. Conclusion and Future Work

An optimized Transformer-based approach was presented in paper for the detection of Cross-Site Scripting (XSS) attacks. By integrating Rotary Positional Embeddings (RoPE) and Flash Attention, we achieved an accuracy of 99.38% while reducing the computational overhead. Our use of a multi-source dataset, including live network captures, prepares the model to be resilient against real-world, obfuscated payloads and ready for high-throughput network environments.

While this study focuses on the detection of XSS, we identify two primary directions for future research. The first one is to expand the model dataset to detect other common injection vulnerabilities highlighted in the OWASP Top 10, specifically SQL Injection (SQLi) and Command Injection. Thes second one is to explore the use autonomous mitigation via Agentic AI. Our future research will focus on integrating this detection engine into an Agentic AI framework such that once an attack is identified with high confidence, an autonomous agent will be triggered to perform real-time mitigation tasks. This includes dynamically updating firewall policies, isolating compromised sessions, and performing automated root-cause analysis. Such a system would reduce the time to remediate from minutes to milliseconds, moving toward a fully self-healing web security architecture.

Acknowledgements

The authors of this paper would like to express their thanks to the Higher Education for Economic Transformation (HEET) project for sponsoring the first author's studies.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] OWASP (2025) Introduction-OWASP Top 10. https://owasp.org/Top10/2025/0x00_2025-Introduction

[2] Stency, V.S. and Mohanasundaram, N. (2021) A Study on XSS Attacks: Intelligent Detection Methods. *Journal of Physics*: *Conference Series*, **1767**, Article ID: 012047. https://doi.org/10.1088/1742-6596/1767/1/012047

[3] Fang, Y., Li, Y., Liu, L. and Huang, C. (2018) DeepXSS: Cross Site Scripting Detection Based on Deep Learning. *Proceedings of the* 2018 *International Conference on Computing and Artificial Intelligence*, Chengdu, 12-14 March 2018, 47-51. https://doi.org/10.1145/3194452.3194469

[4] Yan, H., Feng, L., Yu, Y., Liao, W., Feng, L., Zhang, J., *et al*. (2022) Cross-Site Scripting Attack Detection Based on a Modified Convolution Neural Network. *Frontiers in Computational Neuroscience*, **16**, Article 981739. https://doi.org/10.3389/fncom.2022.981739

[5] Kascheev, S. and Olenchikova, T. (2020) The Detecting Cross-Site Scripting (XSS) Using Machine Learning Methods. 2020 *Global Smart Industry Conference* (*GloSIC*), Chelyabinsk, 17-19 November 2020, 265-270. https://doi.org/10.1109/glosic50886.2020.9267866

[6] Yang, W., Zuo, W. and Cui, B. (2019) Detecting Malicious URLs via a Keyword-Based Convolutional Gated-Recurrent-Unit Neural Network. *IEEE Access*, **7**, 29891-29900. https://doi.org/10.1109/access.2019.2895751

[7] Zhou, Y. and Wang, P. (2019) An Ensemble Learning Approach for XSS Attack Detection with Domain Knowledge and Threat Intelligence. *Computers & Security*, **82**, 261-269. https://doi.org/10.1016/j.cose.2018.12.016

[8] Kaggle (2020) Cross Site Scripting XSS Dataset for Deep Learning. https://www.kaggle.com/datasets/syedsaqlainhussain/cross-site-scripting-xss-dataset-for-deep-learning

[9] OWASP (2025) Cross Site Scripting Prevention Cheat Sheet. OWASP Cheat Sheet Series. https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md

[10] XSSED (2015) XSSed | Cross Site Scripting (XSS) Attacks Information and Archive. http://www.xssed.com

[11] Huyen, C. (2022) Designing Machine Learning Systems. O'Reilly Media, Inc.

[12] PortSwigger (2024) Input Data Encoding and Obfuscation. https://portswigger.net/web-security/essential-skills/obfuscating-attacks-using-encodings

[13] Sennrich, R., Haddow, B. and Birch, A. (2016) Neural Machine Translation of Rare Words with Subword Units. *Proceedings of the* 54*th Annual Meeting of the Association for Computational Linguistics* (*Volume* 1: *Long Papers*), Berlin, 7-12 August 2016, 1715-1725. https://doi.org/10.18653/v1/p16-1162

[14] Pennington, J., Socher, R. and Manning, C. (2014) Glove: Global Vectors for Word Representation. *Proceedings of the* 2014 *Conference on Empirical Methods in Natural Language Processing* (*EMNLP*), Doha, 25-29 October 2014, 1532-1543. https://doi.org/10.3115/v1/d14-1162

[15] Vaswani, A., *et al*. (2017) Attention Is All You Need. arXiv: 1706.03762.

[16] Noh, S. (2021) Analysis of Gradient Vanishing of RNNs and Performance Comparison. *Information*, **12**, Article 442. https://doi.org/10.3390/info12110442

[17] Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W. and Liu, Y. (2024) Roformer: Enhanced Transformer with Rotary Position Embedding. *Neurocomputing*, **568**, Article ID: 127063. https://doi.org/10.1016/j.neucom.2023.127063

[18] Burkov, A. (2025) The Hundred-Page Language Models Book: Hands-On with PyTorch. True Positive Inc.

[19] Dao, T., Ermon, S., Fu, D., Ré, C. and Rudra, A. (2022) Flashattention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *Advances in Neural Information Processing Systems* 35, New Orleans, 28 November-9 December 2022, 16344-16359. https://doi.org/10.52202/068431-1189

[20] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014) Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, **15**, 1929-1958.

[21] Goodfellow, I., Bengio, Y. and Courville, A. (2016) Deep Learning. MIT Press.

[22] Loshchilov, I. and Hutter, F. (2019) Decoupled Weight Decay Regularization. arXiv: 1711.05101.