Journal of Theoretical and Applied Information Technology
30th September 2024. Vol.102. No. 18
© Little Lion Scientific

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A NOVEL MODEL BASED ON DEEP TRANSFER LEARNING FOR DETECTING MALICIOUS JAVASCRIPT CODE

XUAN DAU HOANG[1], THI THU TRANG NINH[2], HOANG DUY PHAM[3]

[1,2,3]Cybersecurity Lab, Posts and Telecommunications Institute of Technology, Hanoi, Vietnam

E-mail: [1]dauhx@ptit.edu.vn, [2]trangntt2@ptit.edu.vn, [3]duyph@ptit.edu.vn

ID 55470 Submission	Editorial Screening	Conditional Acceptance	Final Revision Acceptance
01-09-24	02-09-2024	21-09-2024	24-09-2024

ABSTRACT

The escalating prevalence of cyber threats and malware attacks across multiple platforms in recent years has highlighted the need for automated machine learning defense mechanisms. Numerous studies have focused on leveraging deep learning to identify malicious JavaScript code, showing promise in improving cybersecurity measures. Moreover, advancements in large language models (LLM), particularly generative pre-trained transformer-based models like GPT-2/3, have also created opportunities for more effective cyber threat prevention. Overall, these developments point to the significant potential of deep learning techniques for the efficient training of models and effective detection of threats within JavaScript code. This paper proposes a novel deep transfer learning-based model for detecting malicious JavaScript code using CodeBERT to improve the detection performance and minimize manual data engineering tasks. Since CodeBERT can be fine-tuned to adapt to different downstream tasks, we formulate different approaches based on CodeBERT to explore possible scenarios. We then evaluate our approaches on various datasets, and compare the performance of our models with previous researches, as well as baseline models, including both deep learning and traditional machine learning methods. Experimental results confirm that our CodeBERT-based model can detect malicious JavaScript code efficiently on various experimental datasets with the F1-score of 99.3%, which is better or comparable with results of the state-of-the-art proposals.

Keywords: *XSS detection model, malicious JavaScript detection, XSS detection based on deep transfer learning, CodeBERT-based XSS detection model*

1. INTRODUCTION

In this section, we first briefly introduce about malicious JavaScript code, next is a review of some related works and then is paper contributions and organization.

1.1. Overview of Malicious JavaScript Code

Malicious code poses a significant threat to network security. According to the 2019 Internet Security Threat Report, there are an average of 142 million malicious code cyber-attacks daily across more than 157 countries and regions worldwide [43]. Malicious code is defined as software or code designed to inflict harm on entities both online and offline, including businesses, individuals, and organizations. This paper will concentrate on a particular type of malicious code, specifically malicious JavaScript (JS), which is commonly embedded in webpages. Due to its widespread use, JavaScript is a frequent target for malicious activities and can be categorized into various types:

– Drive-by-Download attacks: refers to the download of malware to a computer stealthily without the user's knowledge or consent. After that, the malware may be executed or installed by itself, or by the user unknowingly. Websites that contain these kinds of malware can either be legitimate websites, or pages that are set up specifically to distribute malware. They may contain exploit kits that are used to attack vulnerabilities.
– Cross-site scripting (XSS): This is made possible by the lack of user input validation and sanitization, usually through forms and comments. The malicious content is usually put in place by a malicious user (attacker), and when other users (victims) visit the page, the malicious code will be executed on their browser. The code can be used to steal cookies, session tokens, or other sensitive information, or to perform other malicious actions. An example of XSS attack is shown in Figure 1.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

- Cross-Site Request Forgery (CSRF): Instead of stealing information like XSS, CSRF tricks the users into performing unwanted actions, which include changing the user's email address, password, or even transferring funds from their bank accounts.
- Heap spraying attacks: This technique involves the attacker filling the memory heap with malicious code, and then using a vulnerability to execute the code. This technique is often used together with other attacks, such as XSS.

There are also other types of malicious attacks that involve JavaScript, such as hidden iframes which load JavaScript malware, or "maladvertising" [49], which leverages advertising to distribute various forms of malware.

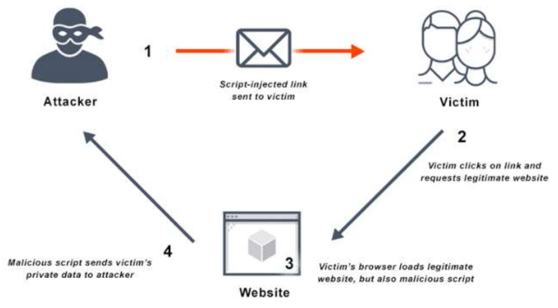
Attacker
1
Script-injected link sent to victim
Victim
2
Victim clicks on link and requests legitimate website
3
Victim's browser loads legitimate website, but also malicious script
Website
4
Malicious script sends victim's private data to attacker

Figure 1: The process of a reflected XSS attack [2]

As malicious code technology has developed, so too has technology for combating it. The primary defense against malicious code is its detection, which aims to analyze software or code features to identify its malicious nature. The accuracy of this detection is crucial for mitigating the damage caused by malicious code. Two primary methods for malicious code detection are static and dynamic analysis. Static analysis examines software without running it, usually by analyzing its source or binary code. Dynamic analysis involves executing the software in a controlled setting to monitor its behavior. Each method has its pros and cons: static analysis is faster but may yield false positives, while dynamic analysis is more accurate but time-consuming. Often, both are used together for improved detection. Still, there is a need for a new method that can achieve high accuracy while maintaining a low time cost. With the advance of artificial intelligence (AI), there have been many attempts to apply machine learning to malicious code detection. Following this trend of applying in security fields as well as to improve the detection performance and minimize manual data engineering tasks, this paper proposes a novel model based on CodeBERT deep transfer learning for detecting malicious JavaScript code embedded in webpages.

1.2. Related Works

Due to the importance of malicious code detection, there have been many attempts to apply machine learning to this problem from network security scholars and experts. The approaches outlined in these works often use static or dynamic analysis method, or a combination of both, and many of them also use machine learning and deep learning models to aid the detection.

Dynamic analysis is a common approach used for malicious code detection. Detection systems usually extract behavioral features during the execution of code, and then use honeypots to simulate the browser environment to execute JavaScript code, which produce information that can be recorded and analyzed to provide details and conclusion regarding malicious intention of the code. A honeypot serves as a specialized computing resource designed for the detection of unauthorized activities and cyber-attacks. Its primary function is to facilitate the study and analysis of the methods and objectives of potential attackers. Many studies have proposed the usage of honeypot for malware detection [28], such as using low and high interaction client honeypots [1]. Since code execution is part of dynamic analysis, these approaches can be effective against zero-day attacks. However, they can also be susceptible to infiltration by attackers, which is a major downside of honeypot implementations, and can be resource-intensive.

Besides dynamic analysis, static analysis is a common approach used for malicious code detection. Traditionally, security systems use rule-based approaches and pattern matching [39] to prevent malicious attacks, however, these approaches have been shown to be ineffective against zero-day attacks, as they tend to raise false negatives with new classes of malware. On the other hand, these methods can also be very lightweight and fast, making it possible to integrate them directly in the browser [8].

Approaches that leverage machine learning and Deep Learning have been proposed, where Natural Language Processing (NLP) techniques are adopted into the processing of programming languages. There are three commonly recognized levels of interpretation of a language: lexical, syntactic and semantic [35]. In lexical analysis, a sequence of characters is transformed into tokens. Syntactic analysis then organizes these tokens into valid expressions in the given language, while semantic analysis correctly interprets these expressions to execute a specific algorithm. With this in mind, there are four main approaches to extract features from

JavaScript code for machine learning-based methods:

- Adopt natural language: The code snippets are considered as natural language text, and as such features can be represented as a collection of total number of lines, average string length, number of eval calls, share of encoded characters, file entropies, special functions count, number of special symbols, etc. This approach requires expert domain knowledge in the field of cyber security and an understanding of the programming language in question. Moreover, this approach is based on the assumption that malicious JavaScript will typically use obfuscation techniques to hide its true nature, and therefore will have different features than benign JavaScript [45]. Such an assumption may not always be true, as new types of malwares will always be developed, and they may not always use obfuscation techniques.
- Using lexical features: Following the lexical analysis angle, the code is tokenized into a sequence of lexical units, which can be used as features. The code will usually need to be cleaned by using regular expression rules to remove special characters and comments. Techniques, such as Doc2Vec and Word2Vec are often used to produce vector representation of the code, in conjunction with a classifier for detection [29][33].
- Using syntactic features: In order to associate meanings to the tokens to constitute valid expressions, other studies have also extracted syntactic unit sequence features of abstract syntax tree (AST) and used together with NLP featurization methods like Doc2Vec/Word2Vec to get a better representation of the code structure [11][29]. This approach has the benefit of having a faster preprocessing step, since with original JavaScript, we will need to do some pre-cleaning.
- Using semantic features: This approach leverages semantic information, which includes both AST-based features and other methods, such as Program Dependency Graph [42] to generate JavaScript semantic slices.

With the development of large language models (LLM), there have been studies that propose the use of pre-trained language models in malicious code detection, such as GPT-2 [9]. However, GPT-2 is primarily designed for natural language (NL) understanding and generation tasks, and it does not focus specifically on programming languages (PL). While the CodeBERT family of language models have not been found to be explored in other literatures regarding malicious code detection, they have been shown to be effective in other classification tasks, such as Code Search, which can be formulated as a text-code classification scenario [21]. Therefore, we are motivated to explore the usage of CodeBERT in malicious code detection.

Although there have been a number of previous proposals for malicious code detection using traditional machine learning and deep learning techniques, there are existing issues for further investigations:

- Approaches based on traditional machine learning techniques require the intensive manual definition and extraction of classification features from datasets. In order to maintain the detection performance, this task much be done on each type of datasets.
- Approaches based on deep learning techniques usually require large datasets for training to construct the detection models. The training process may require very high computing resources and take a long period of time because the detection models usually have to be built from the beginning.

In our approach, we explore the use of deep transfer learning techniques to construct our detection models. Specifically, we use CodeBERT that is a pre-trained model using large corpuses of natural languages and programming languages to build our malicious JavaScript code detection models. CodeBERT can reduce the training time as well as to reduce the requirements of computing resources because it is a pre-trained model and our training process is a type of incremental training on the top of CodeBERT. In addition, CodeBERT can eliminate the need of the manual definition and extraction of classification features from datasets because it can process code via some relatively simple preprocessing steps.

1.3. Paper Contributions and Organization

The contributions of our paper are as follows:

- Proposes a novel model based on Code-BERT deep transfer learning for detecting malicious JavaScript code embedded in webpages. Different fine-tuning strategies are investigated and used to enhance the performance of the proposed model;
- Implements, experiments and evaluates the Code-BERT based models against some baseline models and previous proposals. Experimental results confirm that the proposed Code-BERT model outperforms all baseline models in all

performance metrics and it has better or comparable detection results with previous studies.

The remaining of this paper is organized as follows: Section 2 provides background knowledge of traditional machine learning and deep learning algorithms and architectures used in this study. Section 3 describes datasets used in this study, the architecture of the proposed model, baseline models and used performance metrics. Section 4 present experimental setup, implementation, results and discussion. Finally, Section 5 gives the conclusion and future work of the paper.

2. BACKGROUND

2.1. Deep Transfer Learning with CodeBERT

2.1.1. Deep transfer learning

One key challenge in applying deep learning to practical problems is the lack of large-scale datasets. This is especially true for the malicious JS detection task, where the number of malicious code samples is limited. To address this problem, we propose to use deep transfer learning, which is a machine learning method that allows the knowledge transfer from one domain to another. Furthermore, transfer learning operates on the premise that the training data doesn't need to have the same distribution as the testing data, which helps address the issue of having limited training data [44].

The definition of deep transfer learning is as follows: given a learning task T_t based on D_t, and a related but different learning task T_s based on D_s, where D_s and D_t are the source and target domains, respectively, deep transfer learning aims to improve the learning of the target predictive function $f_t(\cdot)$ in T_t using the knowledge in D_s and T_s, where $D_s \neq D_t$ and/or $T_s \neq T_t$ [44]. Among the various deep transfer learning techniques, network-based deep transfer learning is the most widely used. It is based on the assumption that the source and target domains share the same feature space but have different marginal probability distributions [31]. Recently, pre-trained language models, such as BERT and GPT-3, with large amounts of unlabeled data and fine-tuning in downstream tasks have made a breakthrough in NLP domain [10].

Figure 2 illustrates the standard procedure for network-based deep transfer learning. Initially, a pre-trained deep learning model is acquired and set aside for fine-tuning, with its weights preserved. Subsequently, the final layers of this pre-trained model are isolated, and their weights are eliminated. It is commonly assumed that a pre-trained model consists of two segments: layers embedding general knowledge and layers designed for a specific upstream task, such as masked language models in text processing. The prevailing practice is to reset the weights of these task-specific classification layers, thereby maintaining the model's general knowledge while enabling its applicability to downstream tasks like text classification. Thirdly, the model undergoes fine-tuning by ingesting downstream training data, which is typically significantly smaller than the dataset used for the initial training by several orders of magnitude. During this phase, the model learns to perform downstream prediction task by leveraging knowledge from both the pre-trained model and the new training data. Finally, the fine-tuned model becomes ready for deployment in downstream predictive tasks.

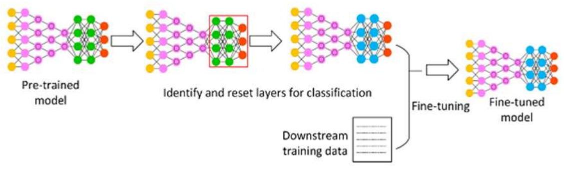

Figure 2: The typical process of network-based deep transfer learning.

2.1.2. BERT and CodeBERT

Language models can generally be divided into n-gram models and neural language models. Traditional neural approaches, such as Word2Vec [26], remain popular. BERT (Bidirectional Encoder Representations from Transformers) advances natural language pre-training through masked language modeling and a Transformer-based framework, leading to significant improvements in state-of-the-art performance across numerous natural language processing tasks. BERT is considered one of the top pre-training models for downstream applications, particularly since more advanced models, like GPT-3 [3], are neither open-source nor readily available.

RoBERTa [23] is a reproduction of BERT that retains BERT's architecture while introducing an enhanced pre-training approach. CodeBERT [13], based on the architecture of BERT and RoBERTa, specifically RoBERTa-large differs from them by taking both natural language and source code as input. Unlike BERT and RoBERTa, which focus on natural language, CodeBERT is trained on a vast dataset of code from several programming languages, such as Python, Java, JavaScript, and C++. This enables it to capture both the syntactic and semantic properties of programming languages. By fine-tuning and adding a classification layer, CodeBERT can be adapted for classification tasks.

2.1.3. Training mechanism of CodeBERT

a. Masked Language Model (MLM)

Given a NL-PL pair datapoint ($x = w, c$) as input, where w is a sequence of NL and c is a sequence of PL, it first selects a random set of positions for both NL and PL to mask out (i.e. mw and mc, respectively), and then replaces the selected spots with a specific token (i.e. $[MASK]$). As mentioned in the CodeBERT paper [13], 15% of the tokens from x are masked out:

$$\begin{aligned} & m_i^w \sim \text{unif}\{1, \dots, |w|\} \text{ for } i = 1 \text{ to } |w| \\ & m_i^c \sim \text{unif}\{1, \dots, |c|\} \text{ for } i = 1 \text{ to } |c| \\ & w^{\text{masked}} = \text{REPLACE}(w, m^w, \text{<mask>}) \\ & x = w + c \end{aligned} \tag{1}$$

The objective of MLM is to predict the original tokens that were masked, and it can be represented as follows:

$$LOSS_{MLM}(\theta) = \sum_{i \in m^w \cup m^c} -\log p^{D_1}(x_i | w^{masked}, c^{masked}) \tag{2}$$

b. Replaced Token Detection (RTD)

The second objective of CodeBERT is replaced token detection as shown in Figgure 3, which further uses a large amount of unimodal data, such as codes without paired natural language texts [13]. Specifically, there are two data generators, a NL generator $p^{G_w}$and a PL generator p^{G_c}, both for generating plausible alternatives for the set of randomly masked positions.

$$\begin{aligned} & \overline{w_i} \sim p^{G_w}(w_i | w^{\text{masked}}) \text{ for } i \in m^w \\ & \overline{c_i} \sim p^{G_c}(c_i | c^{\text{masked}}) \text{ for } i \in m^c \\ & w^{\text{corrupt}} = \text{REPLACE}(w, m^w, \overline{w}) \\ & c^{\text{corrupt}} = \text{REPLACE}(c, m^c, \overline{c}) \\ & x^{\text{corrupt}} = w^{\text{corrupt}} + c^{\text{corrupt}} \end{aligned} \tag{3}$$

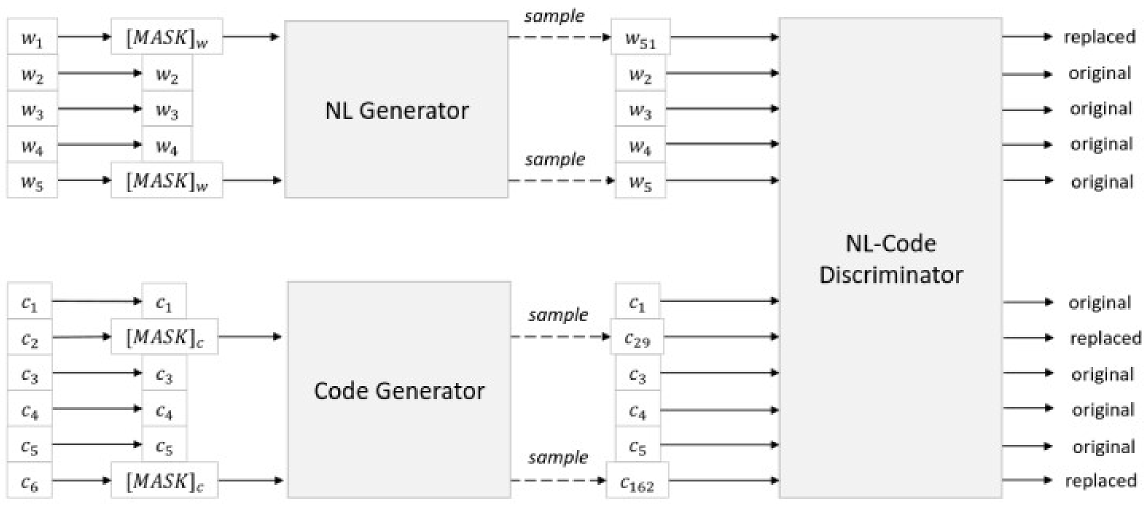

Figure 3: Illustration of replaced token detection [13].

The discriminator is trained to classify whether a given word is the original one or not, making it a binary classification task. The loss function for RTD, with respect to the discriminator parameterized by θ, is defined as follows:

$$LOSS_{RTD}(\theta) = \sum_{i=1}^{|w+|c|} \delta(i) \log p^{D_2}(x^{corrupt}, i) + [1 - \delta(i)][1 - logp^{D_2}([x^{corrupt}], i)] \tag{4}$$

$$\delta(i) = \begin{cases} 1 & if\ x_i^{corrupt} = x_i \\ 0 & otherwise \end{cases}$$

where $pD2$ is the discriminator that predicts the probability of the i-th word being original, and $\delta(i)$ is the indicator function.

There are several ways to implement the generators, and CodeBERT has implemented two n-gram language models with bidirectional contexts, one for NL and one for PL, and learn them from corresponding unimodal datapoints, respectively.

The final loss function is given by:

$$min_\theta \text{ LOSS}_{\text{MLM}}(\theta) + \text{LOSS}_{\text{RTD}}(\theta) \quad (5)$$

2.2. Baseline Models

In this section, we provide a brief overview of several baseline models used for comparing the performance of CodeBERT-based models. These baselines include models built on Bi-LSTM as well as traditional machine learning methods like Random Forest (RF), Support Vector Machine (SVM), and Logistic Regression (LR). Bi-LSTM, RF, SVM, and LR were chosen because they are among the most widely-used and effective models for text classification in natural language processing tasks [13][29][30].

2.2.1. Bi-LSTM

Long Short-Term Memory (LSTM) is a form of recurrent neural network (RNN) that excels at learning long-term dependencies. It is widely applied in natural language processing (NLP) tasks such as text classification, machine translation, and speech recognition. Bidirectional Long Short-Term Memory (Bi-LSTM) [37] is a enhanced version of the Long Short-Term Memory architecture [18].

LSTM is specifically designed to address the vanishing gradient issue found in traditional RNNs, which hinders their ability to learn long-term dependencies. This issue arises when the gradient of the loss function with respect to the network's parameters diminishes to near zero, causing the network to stop learning effectively. LSTM overcomes this challenge by introducing a memory cell capable of retaining information over extended periods. This memory cell consists of three gates: the input gate, output gate, and forget gate. The input gate regulates the information entering the memory cell, the output gate manages the information leaving the cell, and the forget gate determines what information should be discarded. These gates are controlled by sigmoid activation functions, which produce values between 0 and 1, allowing the LSTM to effectively manage long-term dependencies.

2.2.2. Random Forest

Random Forest [4] is an ensemble learning technique that creates numerous decision trees during training and produces a final output based on the majority class (in classification) or the average prediction (in regression) of the individual trees. It extends the bagging method by generating a large set of uncorrelated trees and then averaging their predictions. Random Forest is a highly accurate and versatile method suitable for both classification and regression tasks. This algorithm is widely used across various domains, including image classification, text classification, and speech recognition. The margin function in a random forest quantifies how much the average number of votes for the correct class at X, Y exceeds the average votes for any other class, and it is defined as follows:

$$mg(X, Y) = av_k I(h_k(X) = Y) - max_{j \neq Y}\, av_k I(h_k(X) = j) \quad (6)$$

where $I(\cdot)$ is the indicator function. A larger margin indicates greater confidence in the classification. The generalization error is expressed as:

$$PE^* = P_{X,Y}\,(mg(X, Y) < 0) \quad (7)$$

2.2.3. Support Vector Machine

Support Vector Machine (SVM) [7], introduced in 1995, is a supervised machine learning algorithm mainly used for classification, although it can also be adapted for regression tasks. Its primary objective is to identify a hyperplane that optimally divides data points from different classes within a feature space. Given a training dataset $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$, where $x_i \in \mathbb{R}^d$ is a feature vector and $y_i \in \{-1, 1\}$ is the corresponding label, the objective is to find the hyperplane defined as $w \cdot x + b = 0$ that maximizes the margin between classes. The optimization problem can be formally stated as:

$$minimize = \frac{1}{2}\|w\|^2 \quad (8)$$

$$\text{subject to } y_i(\mathrm{w} \cdot x_i + b) \geq 1, i = 1, \dots, n \quad (9)$$

The dual form of this problem introduces Lagrange multipliers α_i and leads to a quadratic programming problem. The decision function for a new data point x is $f(x) = \text{sign}(\sum_{i=1}^{n} \alpha_i y_i x_i \cdot x + b)$, where α_i are the solutions of the dual problem. Kernel functions can be employed to map data to a higher-dimensional space, enabling SVM to handle non-linearly separable data.

2.2.4. Logistic Regression

Logistic Regression [19] is a supervised machine learning algorithm primarily used for binary classification tasks. Given a set of training data (x_1, y_1), (x_2, y_2), … , (x_n, y_n), where $x_i \in \mathbb{R}^d$ is a feature vector and $y_i \in \{0, 1\}$ is the binary label, the algorithm models the probability that $y_i = 1$ as a logistic function of a linear combination of the features. The model can be mathematically expressed as:

$$P(y_i = 1|x_i) = \frac{1}{1+e^{-(w\cdot x_i+b)}} \qquad (10)$$

where w $\in \mathbb{R}^d$ are the feature weights and $b \in \mathbb{R}$ is the bias term. The objective is to find w and b that minimize the log-likelihood:

$$minimize - \sum_{i=1}^{n}[y_i \log\big(P(y_i = 1|x_i)\big) + (1 - y_i) \log (1 - P(y_i = 1|x_i))] \qquad (11)$$

The model parameters are usually estimated through techniques such as maximum likelihood estimation (MLE), and regularization methods like L1 or L2 regularization can also be applied.

3. THE PROPOSED MODEL FOR DETECTING MALICIOUS JAVASCRIPT CODE

In this section, we first present the datasets and how they are established and processed. Next, we explain the overall architecture of the proposed model that is used for malicious JavaScript code detection.

3.1. Dataset Collection

Our datasets consist of benign and malicious JavaScript codes obtained from various external sources, totaling to 110,985 JavaScript code samples. First of all, we crawl the top 2,000 URLs ranked by Alexa and are able to collect 32,854 code samples from these URLs. We also cross-check these URLs with an URLs dataset from Kaggle [40] to make sure that the URLs are indeed non-malicious. These JavaScript codes are labelled as benign since they are extracted from top website URLs ranked by Alexa. After that, we collect data from various Github repositories, including the Petrak's dataset of 39,450 JavaScript malware samples [32], which have been used and cited in several studies [30]. We also collect data from other datasets, including [16], [47], [50] and [41]. The details of the collected datasets are shown in Table 1.

As seen in Table 1, the total size of the data is almost 8GB, which has proven to be a challenge for data processing and model training. Due to a limitation in time and hardware, we only use 20,515 for the experiments, which are randomly sampled from the original datasets. The distribution of the samples per class is shown in Table 2.

An example of malicious and benign code can be found in Table 3. As seen in this table, the malicious code attempts to embed an *iframe* from an external website into the current web page. While the code is not inherently malicious, the *iframe* is set to be extremely small (width="11" height="1"), which essentially makes it invisible or barely visible to the user. Moreover, the *iframe* source uses HTTP, which is less secure than HTTPS. This could make the content susceptible to man-in-the-middle attacks. In contrast, the benign JavaScript sample is a simple function that inverts the theme of the web page. The function takes in a theme as an argument and returns the inverted theme. The function is not malicious and does not contain any malicious code.

Table 1. Datasets used in this study

Source	No. of benign samples	No. of malicious samples	Total size
Top 2000 Alexa URLs	32,854	0	3.3 GB
Petrak, 2017 [32]	0	39,450	4.5 GB
He, 2018 [16]	1,292	381	3.1 MB
Tsukerman, 2019 [47]	1891	1476	19.3 MB
ZZN0508, 2022 [50]	8,079	8,500	919.1 MB
Singh, 2017 [41]	0	8,062	55.9 MB

Table 2. Distribution of classes of JavaScript snippets

Class	No. of samples	Pro-portion	Average length by character
Benign	9,551	46.5%	71,464
Malicious	10,964	53.5%	55,399

Table 3. Examples of malicious and benign JavaScript

```
(a) Benign code sample
```

```
export const themes = {
night: 'night',
default: 'default'
};
export const invertTheme = currentTheme
=> !currentTheme ||
currentTheme === themes.default ?
themes.night : themes.default;
```

```
(b) Malicious code sample
```

```
document.write(
'<center>' +
  '<iframe width="11" height="1" ' +
  'src="http://laghzesh.rzb.ir" ' +
  'style="border: 0px;" ' +
  'frameborder="0" ' + 'scrolling="auto">'+
  '</iframe>'
);
```

3.2. Preprocessing

Since the data is code snippets, minimal preprocessing is required. However, there are some steps that need to be taken to make the data suitable for input of the model. First of all, the collected data samples are checked to remove any duplication because they are gathered from multiple sources. We do this by calculating the MD5 hash of each code snippet and check for duplicates by setting the hash as the file name. After that, we use the Esprima package [17] to make sure that the code snippet is valid and can be parsed. This is done by checking if the code snippet can be parsed into an AST. If the code snippet cannot be parsed, it is removed from the dataset. Additionally, code comments are deleted, and any white spaces at the beginning and end of the source code are removed.

3.3. The Architecture of the Proposed Detection Model

Figure 4 illustrates the architecture of the proposed CodeBERT-based detection model, which follows a five-step process. In step A, different types of inputs (unimodal or bimodal) are selected, affecting the input data components. Step B involves tokenizing the source code into tokens according to the Backus Normal Form (BNF) grammar rules. In step C, the tokens are either truncated or further processed to prevent information loss from excessively long code. In step D, the data is input into the pre-trained CodeBERT model, which is then fine-tuned for the classification task. Lastly, step E involves evaluating the detection model using a separate test dataset. We explored various input strategies and post-tokenization processing methods, resulting in four fine-tuning approaches for CodeBERT, as detailed in Table 4.

Table 4. Four methods for fine-tuning CodeBERT used in the proposed model

Input strategies	Post-tokenization methods
Unimodal (PL)	Truncation
Unimodal (PL)	Segmentation + Average
Bimodal (NL-PL)	Truncation
Bimodal (NL-PL)	Segmentation + Average

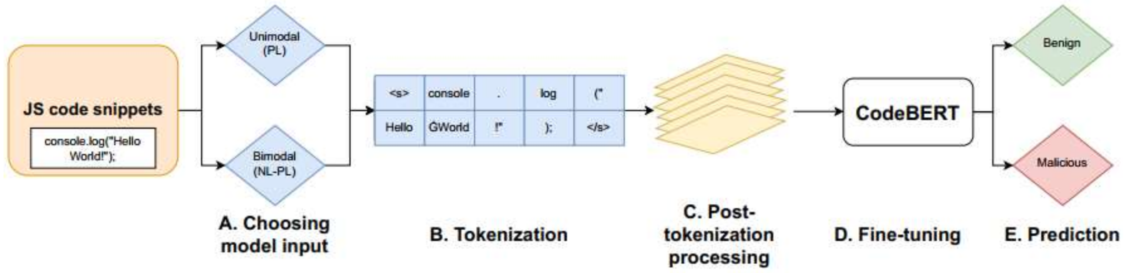

Figure 4: Architecture of the Proposed Detection Model Based on CodeBERT

3.3.1. Choosing model input

Since CodeBERT can be used for various downstream tasks, we can choose different strategies for prediction. Generally, we can classify downstream tasks based on the input and output, which can either be Natural Language (NL) or Programming Language (PL). For example, the input can be the JavaScript code snippet (PL), or a pair of a sentence and a code snippet (NL-PL). Since our task is to detect malicious JavaScript, we can choose two different strategies based on the types of inputs, as shown in Figure 5:

- Unimodal Input (PL only): This is the most straightforward strategy, where we take in code as input and predict 0 for benign and 1 for malicious.
- Bimodal Input (Input is NL-PL pair): Also called sentence-based strategy. We can frame the task as either a sentence-based matching or a keyword-based matching task. For this task, a trained model must determine if a code snippet responds to a specific natural language query, which can be framed as a binary classification problem [21]. For example, we can provide the model with a NL-PL pair of a sentence and a

code snippet, and the model will predict if the sentence matches the code. Similarly, we provide the model with a NL-PL pair of a list of keywords (“malicious”, “harmful”, “malware”) and a code snippet, and the model will predict if the keywords match the code. This means that the labels become 1 for a matching pair, and 0 for a non-matching pair. For this report, we will only focus on the sentence-based strategy. For this study, the following sentences are used as pairing with the codes and assigned to the code randomly with a 50% chance for each class:

+ To denote a malicious code sample: “javascript perform malicious actions to trick users, steal data from users, or otherwise cause harm”;
+ To denote a benign code sample: “javascript perform normal, non-harmful actions”.

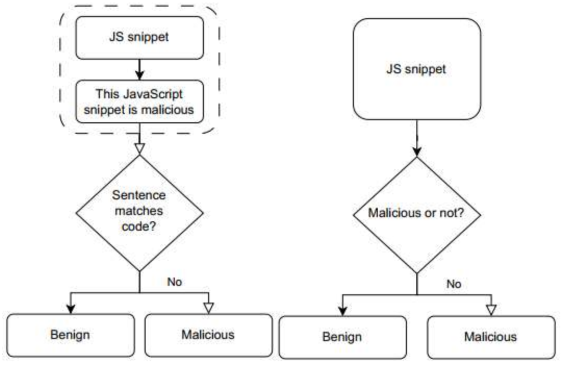

Figure 5:Code (PL) and Text+Code (NL-PL) approach

3.3.2. Tokenization

The CodeBERT tokenizer takes the input string (code or code+text) and convert it into a sequence of tokens that the model can understand. These tokens are subsequently mapped to unique integers (IDs), based on a pre-established vocabulary, and may undergo additional processing such as the addition of special tokens (e.g., [CLS], [SEP]). During tokenization, the tokenizer will split the input into smaller pieces, called tokens. These tokens could be as small as a single character or as long as a word. For example, considering the benign JavaScript example in Table 3.(a), the output of the tokenizer is:

```
['<s>', 'Ċ', 'export', 'Ġconst',
'Ġthemes', 'Ġ=', 'Ġ{', 'Ċ', 'Ġ',
'Ġnight', ':', "Ġ'", 'night', "',", 'Ċ',
'Ġ', 'Ġdefault', ':', "Ġ'", 'default',
"'", 'Ċ', '};', 'Ċ', 'Ċ', 'export',
'Ġconst', 'Ġin', 'vert', 'Theme', 'Ġ=',
'Ġcurrent', 'Theme', 'Ġ=>', 'Ċ', 'Ġ',
'Ġ!', 'current', 'Theme', 'Ġ||', 'Ċ',
'Ġ', 'Ġcurrent', 'Theme', 'Ġ===',
'Ġthemes', '.', 'default', 'Ġ?', 'Ċ',
'Ġ', 'Ġthemes', '.', 'night', 'Ġ:', 'Ċ',
'Ġ', 'Ġthemes', '.', 'default', ';', 'Ċ',
'</s>']
```

The tokens can be understood as follows:

– Special characters like `<s>` and `</s>`: These are start and end tokens, respectively. They indicate the beginning and end of a sequence.
– ‘Ċ’ and ‘Ġ’ Prefixes: Special characters used to indicate a new line (‘Ċ’) or a space (‘Ġ’). They capture the code’s formatting, which can be syntactically significant in some programming languages.
– Keywords and Variable Names: Elements like `export`, `const`, `themes`, `night`, `default`, `invertTheme`, and `currentTheme` are preserved as they are. These are recognizable constructs that are important for understanding the code’s semantics.
– Operators and Punctuation: Characters like ‘=’, ‘{’, ‘}’, ‘:’, ‘?’, etc., are also treated as individual tokens.
– Code Splicing: Some tokens, like `current` and `Theme`, are examples of the tokenizer breaking down composite identifiers into parts that are more common and easier for the model to generalize across different contexts.

If a token is missing from the vocabulary file, an unknown token (, ,) will be used.

This paper employs two preprocessing patterns, aligning with the two input strategies outlined in Section 3.3.1. The first preprocessing approach (for unimodal input) adds a marker at both the start and end of the code. The second approach (for bimodal input) includes an additional marker to distinguish between the source code and the declarative sentence. The code, once preprocessed, is then prepared for tokenization.

3.3.3. Post-tokenization processing

As shown in Section 3.1, the average length by character count of the code snippets is quite high, which would be a challenge for the CodeBERT to learn from, since CodeBERT only supports a maximum length of 512 tokens. Therefore, we need to process the data further to make it suitable for the model. We propose two methods to handle long code snippets:

– Truncation: Simply truncate the code snippet to 512 tokens. This is the simplest method, but it could lead to loss of information;
– Aggregation: Split the code snippet into smaller segments and feed them into CodeBERT to get

their representations. After that, we can aggregate the representations to get a global representation of the code snippet. This method is more complex, but it can preserve more information. The method is illustrated in Figure 6. Given a sequence of tokens $T(C) = \{t_1, t_2, \ldots, t_n\}$, with n being the number of tokens, we can perform the following steps:

+ Segmentation: The tokenized sequence $T(C)$ is split into K segments, each of size L or less. In this case, we choose $L = 512$ and $K = \lceil \frac{n}{L} \rceil$.

$$S_k = t_{(k-1)L+1}, t_{(k-1)L+2}, \ldots, t_{min(kL, N)}, \quad k = 1, 2, \ldots, K \quad (12)$$

+ CodeBERT Transformation: Each segment S_k is then fed into CodeBERT to get its representation $f(S_k)$.
+ Aggregation: Finally, all the segment representations are aggregated to get a global representation G. This is done by taking the average:

$$G = \frac{1}{K} \sum_{k=1}^{K} f(S_k) \quad (13)$$

For this study, the Truncation method and Segmentation by average method are used.

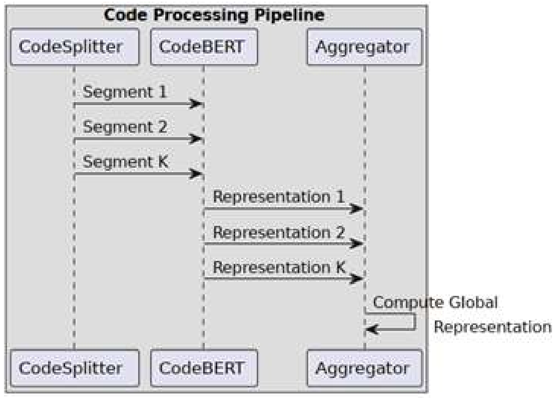

Figure 6: Process to handle long code snippets

3.3.4. Fine-tuning CodeBERT

Fine-tuning CodeBERT with the JavaScript dataset involves performing incremental training on the original CodeBERT framework. As outlined in Section 2.1.3, we can select either MLM or RTD as the fine-tuning objectives. For the purposes of this paper, we will use MLM. The data is split into three distinct subsets: training, validation, and testing. The validation set is utilized throughout training to assess the model's performance at each epoch.

As outlined in Section 3.3.1, corresponding to the two input strategies, we have two fine-tuning methods, unimodal data (PL only) and bimodal data (NL-PL).

a. Unimodal input

A classification layer is added on top of the pre-trained CodeBERT model. Consider CodeBERT as a function F that maps an input sequence (the input IDs) to a latent space z, with being the model parameters, which can be denoted as follows:

$$z = F(x;\ \theta) \quad (14)$$

The resultant z contains the features learned by CodeBERT, encapsulated in a final layer of the transformer architecture. These features are then linearly mapped to a set of logits l via a classification layer, where W is the weight matrix b and is the bias term.

$$l = W_z + b \quad (15)$$

These logits l can be normalized using the softmax function σ to produce a probability distribution p over the two classes, benign and malicious.

$$p = \sigma(l) = \frac{e^{l}}{\sum_{c \in C} e_c^{l}} \quad (16)$$

Since the labeled data y is provided, the cross-entropy loss L can be calculated as follows:

$$L = \sum_{c \in C} y_c \log p_c \quad (17)$$

The loss L is then backpropagated through the network to update the model parameters θ and W, which would fine-tune CodeBERT to the JavaScript dataset for the malicious JavaScript detection task.

b. Bimodal input

For NL-PL input, the NL-PL pair is concatenated and goes through the CodeBERT encoder to extract meaningful representations from both code snippets and natural language descriptions. The combined feature vector of the NL-PL pair Z_{comb} is then passed through a Multi-Layer Perceptron (MLP) with parameter W and sigmoid activation function σ to map the combined latent space to the binary output space Y:

$$logit_i = MLP(Z_{comb_i};\ W) \quad (18)$$
$$P(Y_i = 1 \mid Z_{comb_i};\ W) = \sigma(logit_i) \quad (19)$$

The model is trained by minimizing the Binary Cross-Entropy (BCE) loss $\mathcal{L}$ between the predicted probabilities and the true labels y_i:

$$\mathcal{L} = -\frac{1}{n} \sum_{i=1}^{n} y_i \log(P(Y_i = 1 \,/\, Z_{comb_i};\ W)) + (1 - y_i) \log(1 - P(Y_i = 1 \,/\, Z_{comb_i};\ W)) \quad (20)$$

Finally, a thresholding operation is applied to the output probabilities to obtain binary predictions:

$$\hat{y}_i = \begin{cases} 1 & if\ P(Yi = 1 \mid Z_{comb_i};\ W) > 0.5 \\ 0 & otherwise \end{cases} \quad (21)$$

This approach leverages the powerful representation learning ability of the transformer encoder to map both code and natural language into a common latent space and learns a discriminative function in this latent space to perform the matching classification task.

3.4. Setting up Baseline Models

In this section, we present the process to set up the baseline models for comparison as presented in Section 2.2.

3.4.1. Extracting syntactic features

To process data for Bi-LSTM, we first extract syntactic features from the JavaScript code samples. An abstract syntax tree (AST) serves as a tree-based model of a program's abstract syntactic structure. It transforms source code written in a specific programming language into a tree, where each node corresponds to a statement from the source code. ASTs are widely used for tasks like code checking, analysis, and conversion. For instance, before executing JavaScript code, a browser converts it into an AST using a JavaScript parser. The AST plays a pivotal role in semantic analysis and is instrumental for static program analysis methods.

In this study, we employ Esprima and its Python port [17] to convert JavaScript source code into an AST. Esprima is a high-performance JavaScript parser that takes a program written as a string and outputs an AST. It generates 69 distinct node types, including Program, Statement, Expression, Declaration, and Pattern type nodes. Different code segments are mapped to various node types, also known as syntactic units. Using Esprima, we parse a given JavaScript code sample into an AST, which we then traverse depth-first to produce sequences of syntactic units. The sequences are then used as input to our deep learning model, particularly for Bi-LSTM architecture, since CodeBERT doesn't require sequences of syntactic units as the input.

Consider the following JavaScript code snippet that can be converted to an AST tree as shown in Figure 7:

```
function init() {
   console.log('hello world');
}
init();
```

After isolating the sequences of syntactic units, these sequences are classified using text classification techniques. To be compatible with our model, the sequence undergoes a conversion into word vectors. Given that JavaScript syntactic units of the same category, such as declaration class and expression class, share similar affixes, it is believed that the internal semantic structure of these units should be considered during word vector training. Therefore, this paper proposes to use the FastText model [3] for training sequences of syntactic units. FastText serves as an advanced extension of the Word2Vec model [26]. Unlike Word2Vec, which overlooks the internal structure of words, FastText incorporates sub-word information. It represents each word as a bag of character-level n-grams, and the word vector for a given word is linked with each of its constituent n-grams.

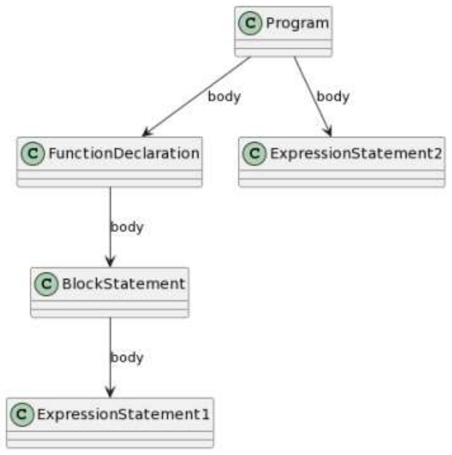

Figure 7: Example of an AST tree

3.4.2. Extracting natural language features

To set up the baseline models using traditional machine learning algorithms, we treat JavaScript code as natural language text and build features based on the text as suggested in previous literatures [47]. The details of the features are shown in Table 5.

Table 5. Features used for the machine learning models

Index	Description
1	Length of the code snippet
2	Count of spaces
3	Combined count of open and close parenthesis
4	Count of slash characters ("/")
5	Count of plus characters ("+")
6	Count of dot characters (".")
7	Count of comma characters (",")
8	Count of semicolon characters (";")
9	Count of alphanumeric characters

10	Count of numeric characters
11	Ratio of spaces to code length
12	Ratio of alphanumeric characters to code length
13	Ratio of numeric characters to code length
14	Ratio of parenthesis to code length
15	Ratio of slash characters to code length
16	Ratio of plus characters to code length
17	Ratio of dot characters to code length
18	Ratio of comma characters to code length
19	Ratio of semicolon characters to code length
20	Count of specific string operations (substring, charAt, split, concat, slice, substr)
21	Ratio of specific string operations to code length.
22	Count of specific encoding operations (escape, unescape, string, fromCharCode)
23	Ratio of specific encoding operations to code length
24	Count of URL redirection functions (setTimeout, location.reload, location.replace, document.URL, document.location, document.referrer)
25	Ratio of URL redirection functions to code length
26	Count of specific functions (eval, setTime, setInterval, ActiveXObject, createElement, document.write, document.writeln, document.replaceChildren)
27	Ratio of specific functions to code length

3.5. Evaluation Metrics

Since the task of malicious code detection belongs to the class of binary classification problems, the four metrics, including Accuracy, Precision, Recall and F1-score are used to evaluate the performance of the detection models:

- Accuracy (Acc): the ratio of the number of correctly classified samples to the total number of samples. It is defined as:

$$Acc = \frac{TP+TN}{TP+TN+FP+F} \quad (22)$$

- Precision (Pre): the ratio of the number of malicious code samples correctly classified as malicious to total number of samples classified as malicious. It is defined as:

$$Pre = \frac{TP}{TP+FP} \quad (23)$$

- Recall (Rec): the ratio of the number of malicious code samples correctly classified as malicious to the total number of malicious code samples. It is defined as:

$$Rec = \frac{TP}{TP+FN} \quad (24)$$

- F1-score (F1): the harmonic mean of precision and recall. It is defined as:

$$F1 = \frac{2TP}{2TP+FP+FN} \quad (25)$$

where, TP, FP, FN and TN are elements of the confusion matrix given in Table 6.

Table 6. TP, FP, FN and TN in the confusion matrix

		Actual Class	
		Malicious	*Legitimate*
Predicted Class	*Malicious*	TP (True Positives)	FP (False Positives)
	Legitimate	FN (False Negatives)	TN (True Negatives)

4. EXPERIMENTS AND RESULTS

4.1. Experiment Setup

The experiments for both CodeBERT-based approaches and the baseline models are run on a machine running Ubuntu 20.04.2 LTS with Python 3.10.11 and CUDA 11.7.

The raw datasets are saved in the file system as separate code snippets and organized into directories based on their labels (“benign” or “malicious”). However, to make it easier to work with, we also save the result after sampling and preprocessing into parquet format, which is much smaller in terms of data size.

To fine-tune CodeBERT, the same hyperparameters are used across all methods. The models are trained with 20 epochs. For the baseline models, 5-fold cross-validation is used to evaluate the models. The results are averaged across the 5 folds.

4.2. Experimental Results

In this section, the results of the experiments are presented. The experiments are conducted on the datasets described in Section 3.1. First, results produced by the proposed model based on CodeBERT are shown and comparison between different input and post-tokenization methods are analyzed. Then, results generated by baseline models are presented and compared with those produced by proposed model based on CodeBERT results.

Table 7 shows the results of detection models based on CodeBERT using different fine-tuning strategies of input and post-tokenization methods. Generally, all strategies perform well over all evaluation metrics. The best performing strategy is bimodal input with truncation, followed by unimodal input with truncation, with a difference of ± ~6%. Notably, the Segmentation + Average approaches with both types of input have a significant drop off compared to the truncation method, which is not in line with the previous hypothesis that the segmentation method would improve the results.

Table 8 shows the performance comparison of baseline models for malicious JavaScript detection. Among the baseline models, the best performing model is the model based on Random Forest with NL features, which has an accuracy of 98.2% and F1 score of 95.1%. The deep learning approach with Bi-LSTM and syntactic features using ASTs performs worse than Random Forest and SVM by about 1% in terms of accuracy, however it has the highest precision, recall and F1-score among all baseline models. The worst performing model is the model based on Logistic Regression with NL features, which has an accuracy of 92.5% and F1 score of 92.7%.

Table 9 compares the detection performance of previous studies with this study. In general, the approach proposed in this study has better or better comparable detection results with previous studies. Specifically, our proposed detection model outperforms proposals in [9], [29], [33] and [42] in all performance metrics. Our model also has comparable detection results with proposals in [11], [30] and [45]. Among these, the approach based on Boosted Decision Tree and natural language features [45] perform better with a reported F1-score of 99.7%, which is slightly higher than the best result of this study by 0.4%.

Table 7. Comparison of malicious JavaScript detection results of models based on CodeBERT using different fine-tuning strategies

Input strategies	**Post-tokenization method**	**Acc (%)**	**Pre (%)**	**Rec (%)**	**F1 (%)**
Bimodal (NL-PL)	*Truncation*	*99.3*	*99.1*	*99.5*	*99.3*
Unimodal (PL)	Truncation	93.1	93.6	93.4	93.1
Bimodal (NL-PL)	Segmentation + Average	86.3	87.1	84.4	85.7
Unimodal (PL)	Segmentation + Average	86.4	86.8	86.7	86.7

Table 8. Performance comparison of baseline models for malicious JavaScript detection

Model	**Featurization method**	**Acc (%)**	**Pre (%)**	**Rec (%)**	**F1 (%)**
Random Forest	*NL features*	*98.2*	*97.3*	*93.0*	*95.1*
SVM	NL features	98.0	97.3	92.0	94.6
Bi-LSTM	Syntactic features	97.1	97.4	97.1	97.2
Logistic Regression	NL features	92.5	92.9	92.5	92.7

Table 9. Performance comparison of the proposed model with other studies

Models	**Papers**	**Acc (%)**	**Pre (%)**	**Rec (%)**	**F1 (%)**
Fine-tuned CodeBERT	*This study (Best result)*	*99.3*	*99.1*	*99.5*	*99.3*
BDT + NL	Tellenbach et. al. (2016) [45]		99.6	98.7	99.7
Bi-LSTM-Attention+ syntactic	Fang et. al. (2020) [11]	99.3	99.4	99.3	99.3
SVM + syntactic	Ndichu et. al. (2019) [30]		99.0	99.0	99.0
Bi-LSTM + semantic	Song et. al. (2020) [42]	97.7	98.6	97.9	98.2
Bi-LSTM-CNN + lexical	Phung et. al. (2021) [33]	97.1	97.4	97.1	97.2
SVM + lexical	Ndichu et. al. (2018) [29]		89.0	90.0	89.0
Fine-tuned GPT-2	Demirci et. al. (2022) [9]	85.4	82.6	89.7	86.0

4.3. Discussion

From the results give in Tables 7, 8 and 9, there are several notable points of discussions:

– Between the two strategies for model input, which are bimodal and unimodal, bimodal input (NL-PL) perform better than unimodal input (PL) by approximate 6%. This can be explained as bimodal input containing more information than unimodal input, while CodeBERT is also pre-trained with a large corpus of natural language input and can understand the concept of malicious versus benign code.

– The results from the *Segmentation + Average* approaches used in the post-tokenization step (Accuracy about 86%) are significantly worse than the truncation method, which has an accuracy from 96% to 99%. This goes against the hypothesis outlined in Section 3.3.3, where it is expected that the segmentation method would be able to retain the global representation of the JavaScript code and avoid information loss from truncation. The reason for this drop-off can potentially come from the complicated implementation of the method, as this implementation is experimental in nature, since this has not been explored in other literature.

– CodeBERT-based models perform better than the baseline models, including both deep learning and traditional machine learning algorithms. This is expected, since CodeBERT is a state-of-the-art model for code representation learning, and it is able to capture the semantic and syntactic information from the code. The results from the baseline models are comparable with other studies, which shows that the dataset used in this study is representative of the problem.

The proposed best CodeBERT-based model performs better or comparable with previous studies, which shows that the CodeBERT deep transfer learning is the appropriate selection for constructing models to detect malicious JavaScript code embedded in webpages.

5. CONCLUSION

This paper proposes a novel model based on Code-BERT deep transfer learning for detecting malicious JavaScript code embedded in webpages. The proposed Code-BERT model outperforms all baseline models in all performance metrics, including models based on traditional machine learning and deep learning techniques. Furthermore, proposed model has better or comparable detection results with previous studies. Specifically, the best proposed Code-BERT-based model has F1-score of 99.3%, which is among the top performance of the state-of-the-art proposals for the detection of malicious JavaScript code. In addition, because Code-BERT a pre-trained model, the Code-BERT-based detection model has less inference time due to its minimal need for preprocessing. This points to the potential of using CodeBERT-based model for malicious JavaScript code detection in real production environment.

Although our CodeBERT-based detection model outperforms many previous studies, it is noted that the proposed model has some limitations as follows: (1) CodeBERT only accept 512 tokens as its input so more preprocessing tasks need to be done for post-tokenization step and (2) there are other pre-trained models, such as GraphCodeBERT and uniXcoder that may perform better than CodeBERT. These issues are future tasks to be carried out:

- Data preprocessing: For the post-tokenization step, it is possible to get the global representation by processing the JavaScript code through a Recurrent Neural Network (RNN). An RNN is a useful choice for aggregating code segments into a global representation since it can capture the sequence information essential for understanding code, as code segments are not independent of each other, while also maintaining contextual information from earlier segments, creating a more comprehensive global representation. Moreover, the current method used in this study (Segmentation + Average) has shown to be not effective, and we hope to improve this further in the future.
- Selection of pre-trained model: Other pre-trained language models that focus on programming language besides Code-BERT have been released and can be used for fine-tuning for malicious JavaScript code detection, such as GraphCodeBERT [14] and uniXcoder [15]. Particularly, uniXcoder considers multi-modal contents like code comment and AST to enhance code representation. This can potentially help with cutting down training and inference time, as well as improving the performance of the model. Moreover, only MLM objective is used in this paper, and as such other objectives like RTD can be explored.

ACKNOWLEDGMENT

The authors deeply thank the Cyber Security Lab, Posts and Telecommunications Institute of Technology, Hanoi, Vietnam for the great assistance in fulfilling this project.

REFRENCES

[1] OWASP Project, https://owasp.org, last accessed 2020/09/20.

[2] Alosefer, Y., Rana, O. (2010). Honeyware: A web-based low interaction client honeypot. 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, Paris, France, 2010, pp. 410-417, doi: 10.1109/ICSTW.2010.41.

[3] AVI Networks (2024), Cross Site Scripting, https://avinetworks.com/glossary/cross-site-scripting/.

[4] Bojanowski, P., Grave, E., Joulin, A., Mikolov, T. (2017). Enriching word vectors with subword information. Transactions of the Association for Computational Linguistics 5:135–146. https://arxiv.org/abs/1607.04606.

[5] Breiman, L. (2001). Random forests. Machine Learning 45(1):5–32. https://doi.org/10.1023/A:1010933404324.

[6] Brown, T., Mann, B., Ryder, N., et al. (2020). Language models are few-shot learners. NIPS'20: Proceedings of the 34th International Conference on Neural Information Processing Systems. In: Advances in neural information processing systems 33:1877–1901.

[7] Choi, Y., Kim, T., Choi, S., Lee, C. (2009). Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis. In: Lee, Yh., Kim, Th.,

Fang, Wc., Ślęzak, D. (eds) Future Generation Information Technology. FGIT 2009. Lecture Notes in Computer Science, vol 5899. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-10509-8_19.

[8] Cortes, C., Vapnik, V. (1995). Support-vector networks. Machine Learning 20:273–297. https://doi.org/10.1007/BF00994018.

[9] Curtsinger, C., Livshits, B., Zorn, B., Seifert, C. (2011). ZOZZLE: Fast and precise in-browser JavaScript malware detection. 20th USENIX Security Symposium. https://www.usenix.org/conference/usenix-security-11/zozzle-fast-and-precise-browser-javascript-malware-detection.

[10] Demirci, D., Sahin, N., Sirlanci, M., Acarturk, C. (2022). Static malware detection using stacked BiLSTM and GPT-2. IEEE Access 10:1–1. https://doi.org/10.1109/ACCESS.2022.3179384.

[11] Devlin, J., Chang, M.W., Lee, K., Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. https://arxiv.org/abs/1810.04805.

[12] Fang, Y., Huang, C., Su, Y., Qiu, Y. (2020). Detecting malicious JavaScript code based on semantic analysis. Computers & Security 93:101764. https://doi.org/10.1016/j.cose.2020.101764

[13] Fass, A., Krawczyk, R.P., Backes, M., Stock, B. (2018). JAST: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In: Giuffrida, C., Bardin, S., Blanc, G. (eds) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2018. Lecture Notes in Computer Science(), vol 10885. Springer, Cham. https://doi.org/10.1007/978-3-319-93411-2_14.

[14] Feng, Z., Guo, D., Tang, D., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. Findings of the Association for Computational Linguistics: EMNLP 2020. https://doi.org/10.18653/v1/2020.findings-emnlp.139.

[15] Guo, D., Ren, S., Lu, S., et al. (2021). GraphCodeBERT: Pre-training code representations with data flow. https://arxiv.org/abs/2009.08366.

[16] Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J. (2022). UniXcoder: Unified cross-modal pre-training for code representation. https://arxiv.org/abs/2203.03850.

[17] He, X., Xu, L., Cha, C. (2018). Malicious JavaScript code detection based on hybrid analysis. In: 2018 25th Asia-pacific software engineering conference (APSEC). pp 365–374. https://doi.org/10.1109/APSEC.2018.00051.

[18] Hidayat, A. (2018). ECMAScript parsing infrastructure for multipurpose analysis. https://esprima.org/doc/.

[19] Hochreiter, S., Schmidhuber, J. (1997). Long short-term memory. Neural computation 9(8):1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735.

[20] Hosmer, Jr. D.W., Lemeshow, S., Sturdivant, R.X. (2013). Applied logistic regression. John Wiley & Sons. DOI:10.1002/9781118548387.

[21] Huang, J., Tang, D., Shou, L., et al. (2021). CoSQA: 20,000+ web queries for code search and question answering. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing. DOI: 10.18653/v1/2021.acl-long.442.

[22] Husain, H., Wu, H.H., Gazit, T., Allamanis, M., Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. https://arxiv.org/abs/1909.09436.

[23] Kim, H.G., Kim, D.J., Cho, S.J., Park, M.J., Park, M.Y. (2012). Efficient detection of malicious web pages using high-interaction client honeypots. Journal of Information Science and Engineering 28:911–924. https://doi.org/10.6688/JISE.2012.28.5.6.

[24] Liu, Y., Ott, M., Goyal, N., et al. (2019). Roberta: A robustly optimized bert pretraining approach. https://arxiv.org/abs/1907.11692.

[25] Lu, S., Guo, D., Ren, S., et al. (2021). CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. https://arxiv.org/abs/2102.04664.

[26] Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. Linux Journal, Vol. 2014. Issue 239. https://dl.acm.org/doi/10.5555/2600239.2600241.

[27] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J. (2013). Distributed representations of words and phrases and their compositionality. NIPS'13: Proceedings of the 26th International Conference on Neural Information Processing Systems. https://arxiv.org/abs/1310.4546.

[28] Nagarjun, P., Ahamad, S.S. (2020). Cross-site scripting research: A review. International Journal of Advanced Computer Science and Applications(IJACSA), Volume 11, Issue 4, 2020. https://doi.org/10.14569/IJACSA.2020.0110481.

[29] Nawrocki, M., Wählisch, M., Schmidt, T.C., Keil, C., Schönfelder, J. (2016). A survey on honeypot software and data analysis. https://arxiv.org/abs/1608.06249.

[30] Ndichu, S., Ozawa, S., Misu, T., Okada, K. (2018). A machine learning approach to malicious JavaScript detection using fixed length vector representation. In: 2018 international joint conference on neural networks (IJCNN). pp 1–8. https://doi.org/10.1109/IJCNN.2018.8489414.

[31] Ndichu, S., Kim, S., Ozawa, S., Misu, T., Makishima, K. (2019). A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors. Applied Soft Computing 84:105721. https://doi.org/10.1016/j.asoc.2019.105721.

[32] Pan, S.J., Yang, Q. (2010). A survey on transfer learning. IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 10, pp. 1345-1359, Oct. 2010, doi: 10.1109/TKDE.2009.191.

[33] Petrak, H. (2017). Javascript malware collection. https://github.com/HynekPetrak/javascript-malware-collection.

[34] Phung, N.M., Mimura, M. (2021). Detection of malicious javascript on an imbalanced dataset. Internet of Things 13:100357. https://doi.org/10.1016/j.iot.2021.100357.

[35] Ramirez, S. (2018). FastAPI. https://github.com/tiangolo.

[36] Rivoira, S., Torasso, P. (1982). The lexical, syntactic and semantic processing of a speech recognition system. International Journal of Man-Machine Studies 16(1):39–63. https://doi.org/10.1016/S0020-7373(82) 80071-0.

[37] Schölkopf, B., Smola, A.J. (2002). Learning with kernels: Support vector machines, regularization, optimization, and beyond. MIT press. https://doi.org/10.7551/mitpress/4175.001.0001.

[38] Schuster, M., Paliwal, K. (1997). Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing 45:2673–2681. https://doi.org/10.1109/78.650093.

[39] Seshagiri, P., Vazhayil, A., Sriram, P. (2016). AMA: Static code analysis of web page for the detection of malicious scripts. Procedia Computer Science 93:768–773. https://doi.org/10.1016/j.procs.2016.07.291.

[40] Shen, V.R.L., Wei, C.S., Juang, T.T.Y. (2018). Javascript malware detection using a high-level fuzzy petri net. In: International Conference on Machine Learning and Cybernetics (ICMLC), Chengdu, China, 2018, pp. 511-514, doi: 10.1109/ICMLC.2018.8527036.

[41] Siddhartha, M. (2021). Malicious URLs dataset. Kaggle. https://www.kaggle.com/datasets/ sid321axn/malicious-urls-dataset.

[42] Singh, A.K., Goyal, N. (2017). MalCrawler: A Crawler for Seeking and Crawling Malicious Websites. In: Krishnan, P., Radha Krishna, P., Parida, L. (eds) Distributed Computing and Internet Technology. ICDCIT 2017. Lecture Notes in Computer Science(), vol 10109. Springer, Cham. https://doi.org/10.1007/978-3-319-50472-8_17

[43] Song, X., Chen, C., Cui, B., Fu, J. (2020). Malicious JavaScript detection based on bidirectional LSTM model. Applied Sciences 10 (10). https://doi.org/10.3390/app10103440.

[44] Symantec (2019). Internet security threat report 2019. https://docs.broadcom.com/doc/istr-24-executive-summary-en.

[45] Tan, C., Sun, F., Kong, T., Zhang, W., Yang, C., Liu, C. (2018). A Survey on Deep Transfer Learning. In: Kůrková, V., Manolopoulos, Y., Hammer, B., Iliadis, L., Maglogiannis, I. (eds) Artificial Neural Networks and Machine Learning – ICANN 2018. ICANN 2018. Lecture Notes in Computer Science(), vol 11141. Springer, Cham. https://doi.org/ 10.1007/978-3-030-01424-7_27.

[46] Tellenbach, B., Paganoni, S., Rennhard, M. (2016). Detecting obfuscated JavaScripts from known and unknown obfuscators using machine learning. International Journal on Advances in Security 9(3/4):196–206. https://doi.org/10.21256/zhaw-1537.

[47] Torrey, L., Shavlik, J. (2010). Transfer learning. In: Handbook of research on machine learning applications and trends: Algorithms, methods, and techniques. IGI global, pp 242–264.

[48] Tsukerman, E. (2019). Machine learning for cybersecurity cookbook. Packt Publishing. https://github.com/PacktPublishing/Machine-

Learning-for-Cybersecurity-Cookbook/tree/master/Chapter03.

[49] Van Houdt, G., Mosquera, C., Nápoles, G. (2020). A review on the long short-term memory model. Artif Intell Rev 53(8):5929–5955. https://doi.org/10.1007/s10462-020-09838-1.

[50] Xing, X., Meng, W., Lee. B., et al. (2015). Understanding malvertising through ad-injecting browser extensions. In: WWW '15: Proceedings of the 24th International Conference on World Wide WebMay 2015, pages 1286–1295. https://doi.org/10.1145/2736277.2741630.

[51] ZZN0508 (2022). JavaScript datasets. https://github.com/ZZN0508/JavaScript_Datasets.