



Understanding the Influence of AST-JS for Improving Malicious Webpage Detection

Rozi, Fakhrur Muhammad ; Ozawa, Seiichi ; Ban, Tao ; Kim, Sangwook ; Takahashi, Takeshi ; Inoue, Daisuke

(Citation)

Applied Sciences, 12(24):12916

(Issue Date)

2022-12-15

(Resource Type)

journal article

(Version)

Version of Record

(Rights)

© 2022 by the authors.
Creative Commons Attribution License

(URL)

<https://hdl.handle.net/20.500.14094/0100488826>



Article

Understanding the Influence of AST-JS for Improving Malicious Webpage Detection

Muhammad Fakhrrur Rozi ^{1,2,*} , Seiichi Ozawa ^{2,3} , Tao Ban ¹, Sangwook Kim ² and Takeshi Takahashi ¹ and Daisuke Inoue ¹

¹ National Institute of Information and Communications Technology, Tokyo 184-8795, Japan

² Graduate School of Engineering, Kobe University, Kobe 657-8501, Japan

³ Center for Mathematical and Data Sciences, Kobe University, Kobe 657-8501, Japan

* Correspondence: fakhrrurozi95@nict.go.jp or rozi_mahfud@stu.kobe-u.ac.jp

Abstract: JavaScript-based attacks injected into a webpage to perpetrate malicious activities are still the main problem in web security. Recent works have leveraged advances in artificial intelligence by considering many feature representations to improve the performance of malicious webpage detection. However, they did not focus on extracting the intention of JavaScript content, which is crucial for detecting the maliciousness of a webpage. In this study, we introduce an additional feature extraction process that can capture the intention of the JavaScript content of the webpage. In particular, we developed a framework for obtaining a JavaScript representation based on the abstract syntax tree for JavaScript (AST-JS), which enriches the webpage features for a better detection model. Moreover, we investigated the influence of our proposed feature on improving the model's performance by using the Shapley additive explanation method to define the significance of each feature category compared to our proposed feature. The evaluation shows that adding the AST-JS feature can improve the performance for detecting malicious webpage compared to previous work. We also found that AST significantly influences performance, especially for webpages with JavaScript content.



Citation: Rozi, M.F.; Ozawa, S.; Ban, T.; Kim, S.; Takahashi, T.; Inoue, D. Understanding the Influence of AST-JS for Improving Malicious Webpage Detection. *Appl. Sci.* **2022**, *12*, 12916. <https://doi.org/10.3390/app122412916>

Academic Editor: Sven Gotovac

Received: 15 November 2022

Accepted: 10 December 2022

Published: 15 December 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: malicious webpage; abstract syntax tree; JavaScript; Shapley additive explanation; machine learning

1. Introduction

The advent of the web has had a tremendous impact on information-system growth in almost every aspect of life. Unfortunately, such technological improvements have encouraged new sophisticated techniques to attack and scam users [1]. Such attacks include fake websites that sell counterfeit goods, financial fraud by tricking users into revealing sensitive information that leads to the theft of money or identity information, or even installing malware on the user's system. The malicious content inside websites often includes JavaScript (JS) code that hides in a hypertext markup language (HTML) either as an inline JS instruction or external code.

JS has an important role in web development. The Stack Overflow Annual Developer Survey 2022 [2] showed that JS is the most used programming or scripting language by many professional developers. The performance of JS influences its popularity to support many programmers in developing more interactive websites with fast running times. JS is prevalent because it is lightweight, flexible, and powerful. However, due to its flexibility with many supported frameworks, libraries, and functions, JS has many vulnerabilities that attackers can exploit for malicious activities.

Various malicious scripts are used to implement such attacks. One of the most atrocious attacks is the so-called crypto trojan (e.g., WannaCry [3]), which often uses JS as a payload in the first stage of infection of the victim's computer. In addition, a JS-based attack allows an attacker to inject a malicious client-side code into a website [4]. Cross-site

scripting (XSS) allows an almost limitless variety of attacks. However, they commonly include transmitting private data such as cookies or other session information to the attacker, redirecting the victim to a webpage controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site. Thus, the analysis of malicious JS code detection is crucial for attack site analysis.

In web security, code injection is still the main threat to Internet users when they access certain websites. Most users do not have the knowledge necessary to distinguish between malicious and legitimate websites. As many legitimate websites are injected with malicious code (e.g., JS), whether any given website is malicious or not is usually unknowable. Even some web security systems fail to detect such websites. Thus, we cannot fully trust a legitimate website due to this code injection threat, and the ability to analyze a malicious webpage is an advantage when an antivirus system faces a code injection threat.

To detect code injection, analyzing the JS code in a webpage is crucial. Some recent works have proposed many ways to identify malicious webpages by extracting enough features to avoid overlooking malicious code in a webpage. Previously proposed detection features include lexical features [5], image capture [6], HTML [7], host-based information [8], and JS [9]. However, the analysis of JS content still lacks semantic meaning, as JS analyzers only extract static information such as character string length, the number of JS functions, and the length of the array used in the code. To address this shortcoming, we seek an additional approach to enrich the features extracted from a webpage to improve the detection system's performance.

To solve this problem, we propose a new webpage feature extraction program—abstract syntax tree-JavaScript (AST-JS)—that recognizes the semantic meaning of the JS code embedded in HTML content. AST-JS-based extraction represents JS code in way that can achieve good performance for detecting malicious JS [10]. The AST is a tree representation of the abstract syntactic structure of source code written in a programming language. Many researchers use AST as the main feature of JS code for some source code analysis tasks. An AST assigns each statement in a source code to a syntactic unit category. The AST structure defines the program's style, which can be useful to discerning the signature feature of the source code. Due to the complexity of the obfuscation problem, we bring the structure-based analysis of AST-JS to capture the whole structure of AST into a low-dimensional representation that retains the semantic meaning of the original JS code. In this study, we used graph2vec [11] as an unsupervised learning model to create a vector representation that is then combined with other feature categories.

Moreover, we give some analysis of the influence of our proposed feature on prediction results. The aim is to understand AST-JS better and improve malicious webpage detection. In summary, the contributions of this study are as follows:

- We present a structure-based analysis of AST-JS for enriching webpage content features to improve the performance of malicious webpage detection.
- We analyze the importance of AST-JS representation for detecting malicious webpages by demonstrating a feature influence analysis of all feature categories using a Shapley additive explanation (SHAP) method.
- We bring the influence analysis to a particular group of webpages to know the benefits of certain features for detecting specific attacks.

The organization of this paper proceeds as follows. Section 2 summarizes related works on webpage detection and the use of AST as the representation of JS. Section 3 presents our proposed approach for improving the current webpage detection system using the AST feature. Section 4 gives details of our experimental setup and dataset. Section 5 explains details of our evaluation of the performance with some result discussions. Finally, we conclude our study in Section 6 by giving some limitations and the future direction of our approach.

2. Related Works

We summarize the previous works most relevant to ours and find the differences with ours. First, the study of malicious webpage detection has been ongoing for quite a long time, since Internet webpages became a source of information that users can access. Simple approaches to block malicious URLs or webpages, such as blacklisting, remain popular. Many websites and communities provide this service for free, such as PhishTank [12], jwSpamSpy [13], and DNS-BH [14]. Alternatively, some companies offer commercial products for this mechanism, such as McAfee's SiteAdvisor [15], Cisco IronPort Web Reputation [16], and Trend Micro Web Reputation [17].

Moreover, machine-learning-based approaches for malicious webpage detection have been rapidly developing. Most of them have explored many features of webpages, such as lexicon-based features that include mostly static features, host-based features that use host information, or content-based ones such as HTML and JS. We can find some research that works on the same features with specific phishing URLs or websites that also consider the webpage's information. Some recent works on phishing detection, such as [18,19], used light gradient boosting to capture the pattern of phishing URLs. The exploration of deep learning models is also an interesting way to create a more complex model that focuses on representing every detail of webpage text or URL characters [20–23]. Additionally, some research with the latest cloud technology uses machine learning to detect specific conditions [24,25] or analyze phishing URLs based on multi-domain study [26].

Furthermore, the existence of JS code on malicious webpages has prompted some researchers to work on detecting malicious code injected into a webpage that probably can be a threat, such as XSS. Most of them use a machine learning approach, such as ensemble classifiers [27]; or deep learning models, e.g., convolutional neural networks [28], deep belief networks [29,30], and graph neural networks [31]. Table 1 summarizes some related works that have been working close to malicious webpage detection.

Table 1. Summary of some related works.

References	Approach	Feature
[12–14]	Blocklisting	URL
[18,19]	Light Gradient Boosting	URL
[20–23]	Deep learning	Webpage text and URL
[24,25]	Cloud-based machine learning	URL and lexical
[26]	Multi-domain study	Image and web content
[27]	Ensemble classifier	Lexical and web content
[30]	Deep Beliefs Networks (DBN)	Web content
[31]	Graph Neural Networks (GNN)	Web content

Meanwhile, regarding AST representation, the use of ASTs for JS feature representation has been adopted as the leading and best way to capture the semantic meaning of a program. We can find some studies that used AST for defect prediction of JS code [32–34], vulnerability detection [35], or code summarization [36]. AST representation is useful for determining how the programmers wrote the code to help us find defects or vulnerabilities. In addition, we can use the AST representation to summarize the semantics of the code and evaluate its maliciousness. We can use machine learning or deep learning for a detection model in which an AST is considered as a list of tokens [37] or graph representation [10,38]. We hope to extend the scope of the research for AST representation for malicious websites [39] or webpages.

3. Overview of Malicious Webpage Detection

In this section, we briefly discuss the background of the cybersecurity risks that we specifically address here, that is, malicious webpages. Then we explain our proposed

method for improving previous works with a JS feature-based approach. We also emphasize a discussion about feature analysis using SHAP values as a method to clarify the transparency and interpretability of a machine learning model.

3.1. Malicious Webpages

Due to the advancement of the Internet, many vulnerabilities and attacks threaten users. Attackers or hackers create malicious websites to perform actions such as stealing credentials or data, or installing malware on the target's computer. Installing malware is still the most effective way to gain unauthorized access to a server, a hosting control panel, or even the admin panel of the content management system.

A malicious webpage is hosted on a website to attack users. The malicious activity can take many different forms, including extracting data from a user, taking control of a browsing device, or using the device as an entry point into a network. Generally, there are two common types of malicious webpages, phishing webpages and malware webpages. For phishing webpages, the attackers try to make a website that looks legitimate, so the target will not recognize it as a risk. Meanwhile, malware webpages refer to a webpage containing malicious payloads that can direct the user's computer to execute the payload code, which may install malware or any intermediary apps. The webpage can be either made by the attacker or by a legitimate webpage with hidden malicious code.

We show two examples of malicious webpages. Figure 1a is an example of a phishing webpage, and Figure 1b is a fake website made with malicious code. A phishing webpage is created to look very similar to a known legitimate site. It tries to imitate every aspect of a legitimate website (e.g., logo, position, and content) so the target users will not recognize that the website is fake. Attackers will make it more promising by sending a fake email recommending that a user access the webpage to fill out a credential information form. Figure 1c is a legitimate website that Figure 1a tries to imitate.

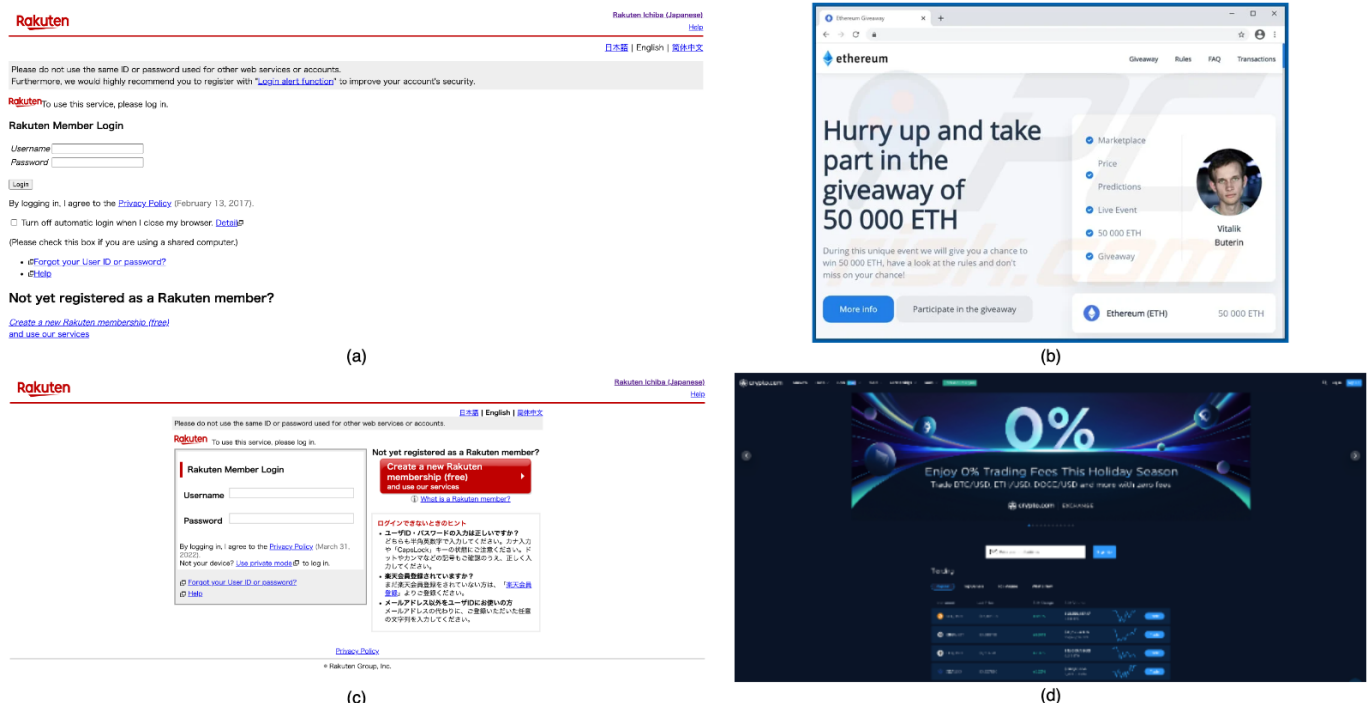


Figure 1. Some examples of malicious and legitimate webpages: (a) a phishing Rakuten website; (b) a fake cryptocurrency website; (c) a legitimate Rakuten website; (d) a legitimate cryptocurrency website.

Meanwhile, the attacker-created fake webpage will likely build its brand entirely from fictitious information, including the logo, address, and content. It makes little attempt to resemble a legitimate webpage, but instead poses as a new legitimate company. Fur-

thermore, an attacker can inject a malicious payload using a fake webpage on a legitimate website. This is the most difficult malicious page to detect, and it can send information to the original (legitimate) server while also sending it secretly to the attacker's server. Figure 1d is one of the counter examples for the fake website in Figure 1b.

3.2. AST-JS

The AST is a tree representation of code that interprets how the programmers wrote the source code. Like human language, we can parse a sentence into a syntax tree representation that explains language grammar, and the AST representation is the output of how we represent a program written in a programming language with a specific syntax rule.

An AST captures the input's essential structure in a tree form while omitting unnecessary syntactic details [40]. We can distinguish it from concrete syntax trees by skipping tree nodes to represent punctuation marks, such as semicolons, to terminate statements or commas to separate function arguments. The AST also omits tree nodes that represent unary operands in the grammar. Such information is directly represented in an AST by the structure of the tree. We can create ASTs with a hand-written parser or code produced by parser generators, generally with a bottom-up approach.

An AST is usually the result of a compiler's syntax analysis phase, a fundamental part of compiler functions. Many programs apply AST in the transforming process. It is not strictly applied in JS environments, such as node.js or a browser. We can find ASTs in Java with Netscape as well. The AST is used intensively during semantic analysis, during which the compiler checks for correct usage of the program's elements and the language. The compiler also generates symbol tables based on the AST during semantic analysis. A complete traversal of the tree verifies the program's correctness. After verifying correctness, the AST serves as the base for code generation. The AST is often used to generate an intermediate representation, sometimes called an intermediate language, for code generation.

Some characteristics of an AST can support subsequent steps of the compilation process. We can edit and enhance an AST with information such as properties and annotations for every element it contains. Such editing and annotations are impossible with the source code of a program because they would change the instructions in the code. An AST does not include unimportant punctuation and delimiters (braces, semicolons, and parentheses) if we compare it to the source code. An AST usually contains extra information about the program due to the compiler's consecutive stages of analysis.

For example, we present simple JS code in Listing 1. We can derive from it the AST in the JSON format file that is shown in Listing 2. If we illustrate the AST as a graph structure, we can represent it as shown in Figure 2, where each node of the graph denotes a construct occurring in the source code, and the edge is the hierarchical relation between nodes. There are many types of syntax tree formats for the AST representation depending on the language's environment. For the JS environment, we have ESTree [41], an organization that creates the standard specification for ASTs. The format is originally from the Mozilla Parser application interface (API) that previously exposed the SpiderMonkey engine's JS parser as a JS API. They formalized and expanded the previous AST's form as the AST's primary reference in any AST-JS parser.

Listing 1. Example JavaScript code.

```
1 var a = 19;
2 var b = 12;
3 var c = a + b;
```

Listing 2. Output of an AST parser (in JSON format) applied to the code in Listing 1.

```
1 {"type": "Program",
2  "body": [{
3    "type": "VariableDeclaration",
```

```

4      "declarations": [{
5        "type": "VariableDeclarator",
6        "id": {
7          "type": "Identifier",
8          "name": "a"},
9        "init": {
10         "type": "Literal",
11         "value": 19,
12         "raw": "19"}}],
13     "kind": "var"},
14   {"type": "VariableDeclaration",
15     "declarations": [{
16       "type": "VariableDeclarator",
17       "id": {
18         "type": "Identifier",
19         "name": "b"},
20       "init": {
21         "type": "Literal",
22         "value": 12,
23         "raw": "12"}}],
24     "kind": "var"},
25   {"type": "VariableDeclaration",
26     "declarations": [{
27       "type": "VariableDeclarator",
28       "id": {
29         "type": "Identifier",
30         "name": "c"},
31       "init": {
32         "type": "BinaryExpression",
33         "operator": "+",
34         "left": {
35           "type": "Identifier",
36           "name": "a"},
37         "right": {
38           "type": "Identifier",
39           "name": "b"}}}],
40     "kind": "var"}],
41   "sourceType": "script"}

```

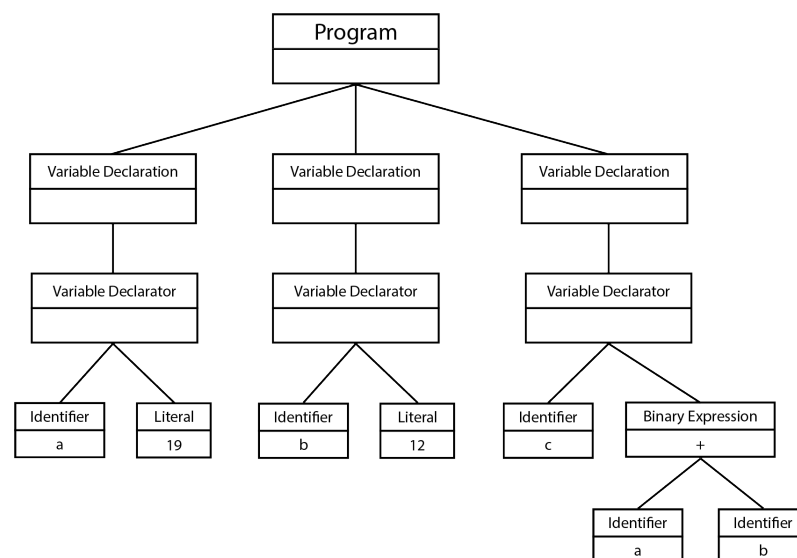


Figure 2. Visualization of an AST structure as a graph.

The AST parser engine will parse a JS code statement into syntactic units. Each node in the resulting syntax tree is a regular JS object that implements a particular node interface object. This interface has a “type” property in strings that contain various types of nodes as syntactic units. Based on the ESTree specification, the standard AST has 69 types of

syntactic units. They represent the type of JS code for each statement or block of code, such as a variable expression, looping expression, or if–else statement.

3.3. AST Graph Features

A graph is a collection of objects including a set of interactions between pairs of objects [42]. It is a ubiquitous data structure and a universal language to describe a complex system. There are many systems that use graph representation. For instance, to encode a social network as a graph, we might use nodes to represent individuals and use edges to express the friendships.

In mathematical notation, a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined by a set of nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$ between these nodes. We denote an edge going from node $u \in \mathcal{V}$ to node $v \in \mathcal{V}$ as $(u, v) \in \mathcal{E}$ [42]. An adjacency matrix $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ is a way to represent graphs. We can order the nodes in the graph to index a particular row and column in the adjacency matrix. We can then represent the presence of edges as entries in this matrix: $\mathbf{A}[u, v] = 1$ if $(u, v) \in \mathcal{E}$, and $\mathbf{A}[u, v] = 0$ otherwise. If the graph contains only undirected edges, $\mathbf{A}$ will be a symmetric matrix, but if the graph is directed (i.e., the edge direction contains information), $\mathbf{A}$ will not necessarily be symmetric. Some graphs can also have weighted edges, where the entries in the adjacency matrix are arbitrary real values rather than $\{0, 1\}$.

Furthermore, in many cases, attributes or feature information are associated with a graph. These are node-level attributes that we represent using a real-valued matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times m}$, where m is the feature size of nodes and we assume that the ordering of the nodes is consistent with the ordering in the adjacency matrix. In heterogeneous graphs, we commonly assume that each different type of node has a distinct set of attributes. In some cases, we have to consider graphs with real-valued edge features in addition to discrete edge types, and we even associate real-valued features with entire graphs.

We often find problems that need a graph-based approach to understand them better and solve them. One example is generating a graph structure of AST features to simplify the representation of JS code in different ways. By representing the structure and content of an AST as tree-graph data, we can use the graph-based approach to find the signature of malicious AST through some features of the graph. We may consider the AST graph structure as the overall JS program code representation that shows the attackers' obfuscation technique, which is generally implemented with nonstandard tools that the average programmer would tend not to use. Even though benign code may contain some obfuscation, we assume that malicious code always has this characteristic. Therefore, we want to analyze the AST graph's structure feature to detect malicious JS code signatures using a graph-based approach.

3.4. Proposed Method

Figure 3 shows the overall process for our analysis method. We propose improving malicious webpage detection by enriching the feature extraction with AST-JS representation of JS content. Using the structure analysis of AST-JS, we identified the semantic information of a malicious JS program that was embedded in or injected into certain parts of the HTML content. We also evaluated our proposed AST-JS feature analysis to determine whether it significantly enhances the current malicious webpage detection system. We obtained information about the contribution of a feature to the detection model using SHAP [43].

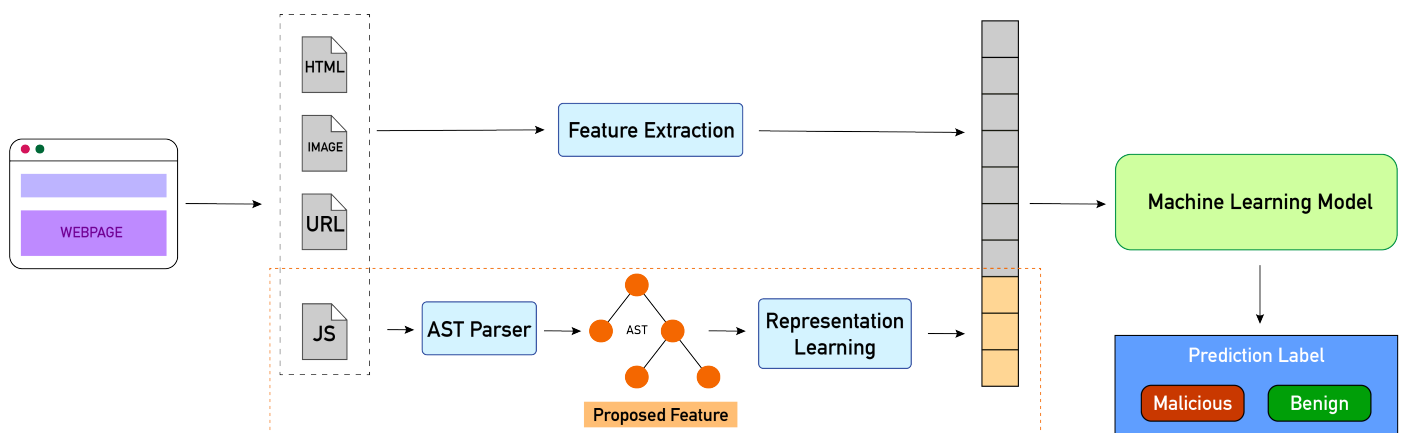


Figure 3. Overview of the proposed method.

3.4.1. Framework Overview

Starting with an input of a webpage, we can extract all of its features, such as HTML, images, the URL, and the JS code. Then, we preprocess all features to represent each one numerically. To enrich the features, we add one more process that makes a semantic representation of the JS code based on the AST-JS structure. Using the JS content that we already have, we first parse the JS code dataset to obtain the AST representation for each JS file. This first output is in the form of JSON format files containing all the JS AST representation information. Next, we construct the graph structure of the AST information. We assume each syntactic unit as a single graph node and each edge as a hierarchical relationship between two syntactic units. We use graph2vec [11], a representation learning model, to create a low-dimensional representation of the AST-JS graph. After that, we combine our representation with other feature vectors to create a complete webpage representation. Then, we can use any machine learning model to predict whether it is malicious or benign.

3.4.2. Current Malicious Webpage Detection System

This research aims to improve on previous work on malicious webpage detection, which lacked the exploration of the semantic meaning of JS content. Specifically, we tried to improve on the previous work of Chaiban et al. [44], which used almost all components or features of a webpage, from the URL to HTML and JS content. We can see the detailed feature information used in this research in the original paper. However, overall they used seven feature categories:

- Lexical features (L): This feature category is about all URL-related information, and we extract many features based on that. There are various features for this category. It can be just the length of the URL character string or the number of specific symbols or characters in the URL, such as semicolons, underscores, ampersands, and equal signs.
- Host-based features (H): This category has five features: IP address, geographic location, WHOIS information, the type of protocol, and the inclusion of the Alexa top domain.
- Content-based features (C): This category is based on HTML content that also has JS content embedded in the script tag. This feature category includes any lexical exploration of HTML and JS content, such as the number of suspicious JS functions, the length of JS and HTML code, the number of script tags in HTML, or the average and maximum arrays we can find in the JS code.
- Image embedding (IM): After we capture the webpage, we can represent it using MobileNetV2 [45].
- Content embedding (CE): This category uses the HTML content to make vector embedding with CodeBERT [46].

- URL embedding with Longformer (UL): We use the Longformer transformer [47] to transform URL text into a low-dimensional representation.
- URL embedding with BERT (UB): We use the Distilbert transformer [48] to transform URL text into a low-dimensional representation.

The reason for improving on the work of Chaiban et al. [44] is that their work covers all kinds of features of a webpage, from the URL to the JS code. They also proposed a good dataset that is quite challenging for building a model to separate malicious and benign webpages. However, their work did not sufficiently explore a webpage's JS content, which is the critical point of maliciousness code. Most web security threats, such as drive-by download, XSS, or injection, use JS as the payload for their attack. Therefore, we need an additional extraction for JS content to improve detection performance, especially for specific threats related to JS.

3.4.3. AST Parsing

We start with the JS code file in Listing 1. We can then parse each statement from top to bottom to break it into smaller source code components. We have to check the syntactic category to which the code belongs. For example, in Listing 1, the first code line is a "VariableDeclaration" object consisting of one "VariableDeclarator" object. This object has two main properties: "BinaryExpression" and "Identifier". For binary expression, we will have two literal values, 6 and 7, as the property of that object. Figure 4 depicts how we can parse one line of JS code into an AST.

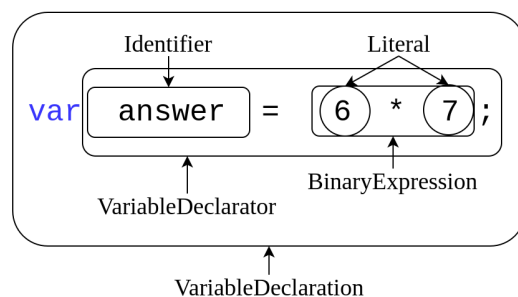


Figure 4. How we break down a single JavaScript statement into an AST.

There are many tools for representing JS code files as ASTs. We used the popular open-source AST-JS parser Esprima [49]. This tool performs lexical and syntactical analysis of JS programs running in some JS environments, including web browsers (including Edge, Firefox, Chrome, and Safari), node.js, and a JS engine in Java (e.g., Rhino). The Esprima parser takes a string representing an accurate JS program and generates a syntax tree, an ordered tree that defines its syntactic structure. Esprima acquires the syntax tree format originating from the Mozilla Parser API, formalized and expanded as the ESTree specification.

3.4.4. AST Graph Construction and Node Reduction

The Esprima parser transforms JS code into an AST representation. However, in the JSON format, its elements are a list of syntactic unit objects with some properties depending on the type of syntactic unit, such as an "AssignmentPattern" with three properties: type, left, and right. Nevertheless, every object always has a type property that indicates the name of the syntactic type. Thus, we have to generate graph objects that describe the structure and attributes of the AST representation.

Let $\mathcal{S} = \{S_0, S_1, S_2, \dots, S_N\}$ be a set of AST files that we want to use to generate the graph object. Each of them has a list of the syntactic unit types that construct the AST object. As the number of units is finite, we can create the syntactic units' vocabulary T as the node type. We use a recursion algorithm to generate all nodes of the AST. The detailed recursive

algorithm is presented as Algorithm 1. We assign the parent input as the “root” node for the starting node.

Algorithm 1: Generating a graph from an AST.

input : AST object S_i , parentNode, a graph $\mathcal{G}$
output: $\mathcal{G}$
Function *Recursive-Generator*(S_i , parentNode, $\mathcal{G}$) **is**
 $nodeName = S_i[type]$;
 $\mathcal{G}.\mathcal{V}.add(nodeName)$;
 $\mathcal{G}.\mathcal{E}.add(parentNode, nodeName)$;
 if S_i has an object **then**
 for s in S_i **do**
 | *Recursive-Generator*(s , nodeName, $\mathcal{G}$)
 end
 end
end

Reducing the number of nodes in an AST graph is necessary because a large number of nodes within a graph increases the burden of the graph representation learning model when obtaining its optimal value. It will take much time to visit all nodes within the graph, making detection inefficient. If we consider all possible nodes, the memory consumption will exceed the capacity of typical computing systems. Therefore, to solve this problem, we omit some nodes within the graph without removing the semantic meaning of either the structure or node attributes.

Thus, omitting some nodes can give some advantages to our detection system. The first advantage is that it can reduce the time spent processing the input data because it does not need to visit (examine) all nodes within the graph, which can have more than 100,000 nodes. In addition, node omission can reduce the memory used to train all parameters in our model. We cannot ignore that memory capacity limits training, especially if we want to train with a large dataset. The final advantage is that it prevents our model from overfitting due to too many parameters, which would happen if we consider an AST graph with a relatively large number of nodes. Han et al. [50] proved that the lightweight AST, which reduces the number of nodes in the AST graph, can have higher accuracy and time efficiency than the conventional AST.

Furthermore, we cannot reduce the number of AST graph nodes randomly. We need to choose nodes that are “safe” to remove without compromising our purpose. We use a breadth-first search (BFS) tree algorithm to select nodes for removal. BFS refers to the method in which we traverse a graph by examining all children of a node before moving on to that child’s children Figure 5. The BFS algorithm visits all graph nodes that traverse the graph layerwise, thereby exploring the neighboring nodes. As we can see, BFS provides a layerwise algorithm that fits our problem. The AST graph has a top-down structure in which the parent nodes sufficiently represent the child nodes under the parents. We can consider the bottom node as the top node’s detailed information, described by the node type.

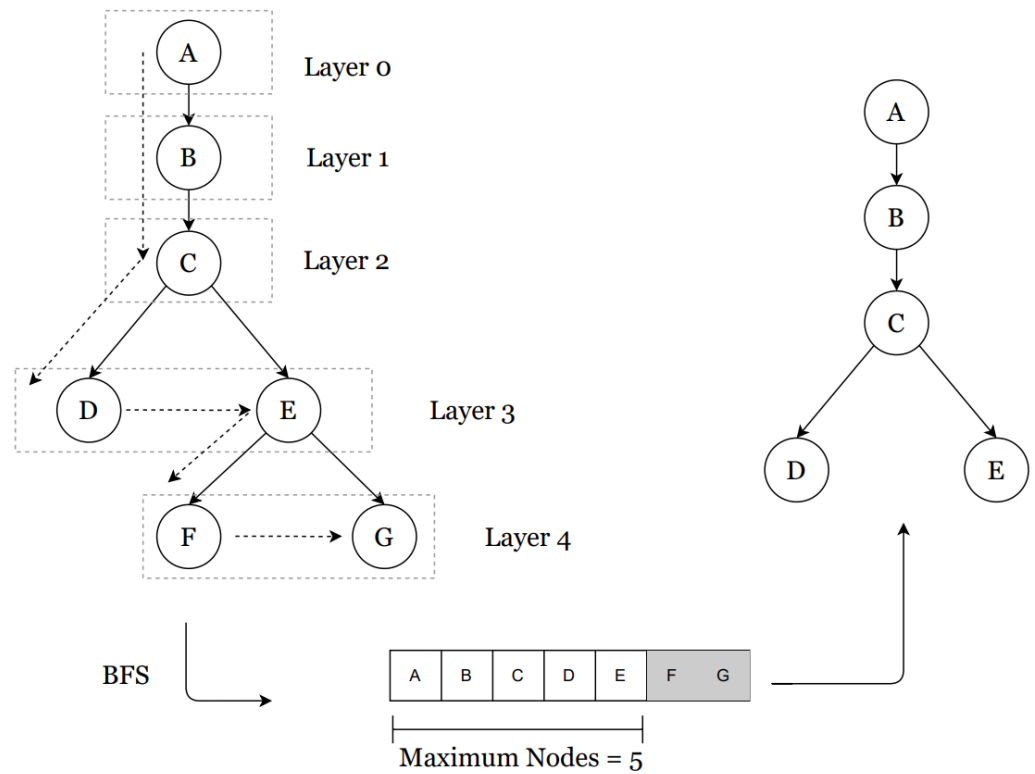


Figure 5. Using the BFS algorithm to trim the AST graph to obtain the minimum number of nodes needed for machine learning.

3.5. Representation Learning

In this study, we used graph2vec [11] for representation learning to obtain a low-dimensional vector. After we had a set of graphs $\mathbb{G} = \{G_1, G_2, \dots, G_N\}$, which was the result of AST-JS transformation, we used graph2vec to encode each graph to closely represent the AST graph's characteristics. This unsupervised learning process produced a vocabulary embedding vector to store all AST-JS representations for building a malicious webpage detection model.

For a more detailed explanation, let $G = (N, E, f)$, where N is a set of nodes, E is a set of edges, and f is a function for mapping each node with a label l . Then, a rooted subgraph can be written as $sg = (N_{sg}, E_{sg}, f_{sg})$. In a graph G_i , a rooted subgraph of degree D around node $n \in N$ contains all the nodes reachable in d hops from n . We identify all the neighbors of $n \in N$ using the BFS algorithm where $d > 0$. Using all rooted subgraphs, we can train an unsupervised model such as doc2vec [51], where a set of AST graphs $\mathbb{G} = \{G_1, G_2, \dots, G_N\}$ and a sequence of rooted subgraphs $SG = \{sg_1, sg_2, \dots, sg_N\}$ are sampled from AST graph $G_i \in \mathbb{G}$. The graph2vec skip-gram learns F -dimensional embeddings of the graph $G_i \in \mathbb{G}$, and each rooted subgraph sg_i sampled from SG . This model considers a rooted subgraph $sg_i \in SG$ to be occurring in the context of graph G_i and maximizes the following log-likelihood:

$$\sum_{j=1} \log Pr(sg_j | G_i) \quad (1)$$

where the probability of $Pr(sg_j | G)$ is defined as

$$\frac{\exp(G \cdot sg_j)}{\sum_{sg \in \mathbf{V}} \exp(G \cdot sg)}, \quad (2)$$

where $\mathbf{V}$ is the vocabulary of all the rooted subgraph across all graphs in G .

3.6. SHAP Values

SHAP is a unified framework for interpreting predictions by assigning an importance value for each feature that influences a particular prediction [43]. This method addresses one of the concerns in machine learning, namely, interpretability, where researchers try to explain how their model makes predictions. In addition, SHAP can also give more perspective on the influences of specific features, whether they are essential or not.

To implement this method, we need to calculate the SHAP values that define how each datum influences the entire prediction. SHAP values are the framework that unifies Shapley regression, Shapley sampling, and the quantitative input that influences feature attributions while allowing for connections with LIME [52], DeepLIFT [53], and layer-wise relevance propagation. We can then retrieve information on how high or low feature values can support negative or positive prediction results based on a summary of these values.

4. Experimental Setup

This section introduces the experimental setup for our evaluation using a benchmark dataset and explains the best setup for the machine learning model.

4.1. Dataset

We used GAWAIN, a benchmark dataset by Chaiban et al. [44] intended for malicious webpage detection, to evaluate our proposed model and analysis. This dataset is much more complicated than those used by other studies on malicious webpage detection. We think that this dataset has two main criteria that can be the best options for us to evaluate our new proposed feature, which can be described as follows:

- The inclusion of webpage features that have been proposed in some previous works. Features from related works were also added to better represent a single web page, such as image embedding, content embedding, or two different URL embeddings based on pre-trained word representation (Longformer and BERT). The availability of previous feature categories can help us compare our proposed feature's significant influence among all feature category combinations.
- Representative distribution of data. This dataset was constructed so that the distribution between malicious and benign data is more difficult to separate. They believed this condition is closer to the actual scenario where attackers try to imitate the legitimize content as much as possible to evade detection.

Table 2 explains the distribution of malicious and benign data in GAWAIN. It contains data for 105,485 webpages, 61,080 of them benign and 44,405 malicious. It provides all existing features that combine previous works with the same topics in malicious webpage and website detection using machine learning.

In the current GAWAIN dataset, not all data contain JS for feature extraction. For analysis purposes regarding JS's influence on malicious webpage detection, we separated the data that contained JS from the data that did not. By separating the data, we can know how important this feature is to the model's performance.

Table 2. Dataset information.

Dataset	Benign	Malicious	Total
All	61,080	44,405	105,485
With JavaScript	36,443	23,923	60,366
Without JavaScript	24,637	20,482	45,199

4.2. Parameter Setup

We evaluated our proposed approach with experiments that could confirm the ability of AST-JS to improve the performance of malicious webpage detection. The objective of this study was to offer a new feature. We focused more on the addition or non-addition of AST-JS in the system. We still focused on using XGBoost [44] as the main machine

learning model, which was also used in previous work on the same dataset, GAWAIN. In addition, XGBoost gives an advantage when calculating SHAP values [43]. However, we also compared the model to other machine learning models. For more details, Table 3 explains some parameters we used in the experiment. We also used values for some parameters that were different from the original settings in [44]. This is because we used the SHAP method for analysis in the end, which is not applicable for the original setup. We found that our new settings give better results than the original one.

For deeper analysis, we performed a clustering experiment on the dataset to label a particular group of webpage types. We decided to implement the BIRCH algorithm [54] because it is simple yet produces a good cluster on the dataset. We set five cluster groups to quickly identify each group's characteristics that perhaps correlate with the performance in predicting malicious webpages.

Table 3. Parameter setup for the XGBoost model.

Parameter Name	Original Setup	New Setup
Learning rate	0.3	0.01
Subsample	0.8	0.5
Evaluation metric	Logloss	Logloss
Gamma	0.01	0.01
Booster	Dart	Gbtrees
Min. child weight	1	1
Max. depth	10	6

5. Evaluation

In this section, we present the performance of the proposed method for detecting malicious webpages with a new approach based on AST-JS information. We describe its overall performance, analyze the manual features' influences, and provide an analysis of webpage features and AST-JS structures that correlate with webpage types based on clustering results.

5.1. Overall Performance

We tested our proposed approach with several conditions based on the parameter settings and the inclusion of the AST-JS feature. Table 4 shows the accuracy performance for each condition with the best combination of feature categories. We experimented with new settings based on Table 3 and compared them with the original settings from previous work [44]. We also experimented with the inclusion of AST-JS to compare the performance before and after adding the proposed feature. We had to find the best combination of feature categories for every condition to get the optimum result.

Table 4. Overall performance of proposed method with and without AST-JS.

Setup	Without AST-JS		With AST-JS	
	Best Feat. Cat. Comb.	Accuracy (%)	Best Feat. Cat. Comb.	Accuracy (%)
Original	L+H+C+UL+IM+CE+UB	83.67	L+C+UL+IM+UB+AST	83.74
New	L+H+C+UL	84.75	L+H+C+UL+IM+CE+UB+AST	84.84

L = lexical; H = host-based; C = content; UL = URL embedding with Longformer; IM = image embedding; CE = content embedding; UB = URL embedding with BERT.

The experiments showed that our new settings give better results: around 84.75% accuracy, higher by 1.08% than the original setup. We can achieve those results by using the combination of feature categories L, H, C, and UL, which is fewer features than in the combination used in the original setup. Moreover, when adding AST-JS as our proposed feature, we can get a better result of around 84.84% accuracy with the best combination of feature categories of L, H, C, UL, IM, CE, UB, and AST-JS.

AST-JS enriches webpage features that better represent the characteristics of JS code that probably contains the payload that attackers try to inject into the webpage. To test our hypothesis, we divided the dataset into two groups: webpages without JS and webpages with JS. This treatment can strengthen the performance because the model will focus more on learning these features, so it is not distracted with webpages without JS. Table 5 shows the experimental results. We achieved higher performance, around 86.10% accuracy, when all learning data contained JS information. Compared to webpages without JS coding, we obtained lower performance, around 81.63% accuracy, with our new model settings. Both the original and new settings can achieve better performance when we have JS information as a webpage feature. This is because the model should generalize and consider the webpage without JS, influencing the precision for obtaining more true positives. Table 6 shows the performance with different machine learning models.

Table 5. Overall performance of the proposed method on webpages that contain JS and those that do not.

Setup	Webpage without JS		Webpage with JS	
	Best Feat. Cat. Comb.	Accuracy (%)	Best Feat. Cat. Comb.	Accuracy (%)
Original	L+H+C+IM+UB	80.34	L+H+C+UL+IM+UB+AST	85.57
New	L+C+UL+IM+CE+UB	81.63	L+H+C+UL+IM+UB+AST	86.10

L = lexical; H = host-based; C = content; UL = URL embedding with Longformer; IM = image embedding; CE = content embedding; UB = URL embedding with BERT.

Table 6. Performance results for different machine learning models.

Model	Precision	Recall	F-Score
Gaussian naive Bayes	0.7134	0.5893	0.5504
Logistic regression	0.7169	0.6806	0.6822
Decision tree	0.7304	0.7299	0.7301
Random forest	0.8189	0.8001	0.8055
XGBoost	0.8539	0.8317	0.8382

Not all webpages have JS information. Attackers could use WebAssembly to install malware if the browser “sandbox” holds up, or they could perform other attacks [55]. Even though JS is widely used for malware infection, attackers can run any vulnerable extension. Further, if the browser uses outdated libraries, then a carefully crafted element can lead to code execution even without JS.

5.2. Influences of Feature Categories

We determined the absolute mean SHAP value for each feature category, representing the influence on prediction output. The aim was to determine how our proposed feature can help the performance improvement after adding it to the other features. However, due to the existence of JS in a webpage, AST-JS features only help process a webpage with JS information, so we tried to calculate the SHAP values for two datasets: the full dataset and webpages with JS.

Figure 6 reports the absolute mean SHAP values for two datasets, showing that AST-JS contributes much to the prediction output. It can be observed that the proposed feature, AST-JS, has a clear influence on producing the predicted labels. AST-JS is one of the top three feature categories in the whole dataset that mixes webpages with and without JS information. This result is understandable because in the whole dataset, we found that around 60% (60,939) of webpages do not have AST-JS features. This condition may influence how AST-JS contributes more to malicious code detection. This also reveals that URL information is a critical feature for achieving higher performance in malicious webpage prediction when we cannot find JS content.

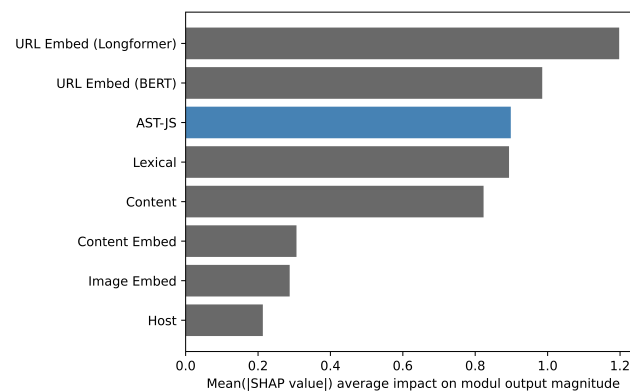


Figure 6. Average SHAP value for each feature category.

Despite that, we still need to consider JS information, where most attackers apply their efforts. Furthermore, AST-JS successfully captures JS information via a low-dimensional representation that can enrich the features to identify a malicious webpage. In Figure 7, we also can see that when the model focuses on webpages with JS, the AST-JS feature has the most influence among all feature categories and can even have more influence than URL information.

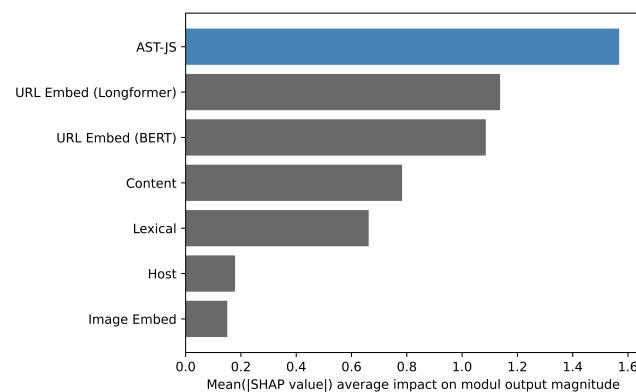


Figure 7. Average SHAP value for each feature category with a dataset that has JavaScript content.

5.3. Comparison with Other JavaScript-Related Features

The original GAWAIN dataset [44] contains JS information that can be extracted as syntactical features. However, these features do not truly capture the whole semantic meaning of JS code, so the model fails to benefit from the existence of JS code. Even though it catches the code's most crucial information (e.g., the number of suspicious JS functions), those features are still insufficient due to some obfuscation techniques that may be applied to evade a detection system and security analyst. To ensure that our proposed approach to extracting the JS information is better than using other JS-related features, we calculated the SHAP values for all those features and compared them with AST-JS's embedding representation. The other features were excluded from this analysis to focus more on features extracted from JS code in a webpage and to make a fair comparison.

Figure 8 shows the results. There are two summaries of SHAP values for two different datasets, a full dataset and a dataset where the webpages have JS information. These two datasets show how the existence of JS can affect the distribution of the influence of each feature for prediction. In Figure 9, we can see that in the top 20 feature influences, AST-JS representation dominates the list compared to other JS-related features. Nevertheless, JS length ("js-length") and the average length of JS code ("js_array_length_avg") still rank number one and two, respectively, for SHAP value distribution.

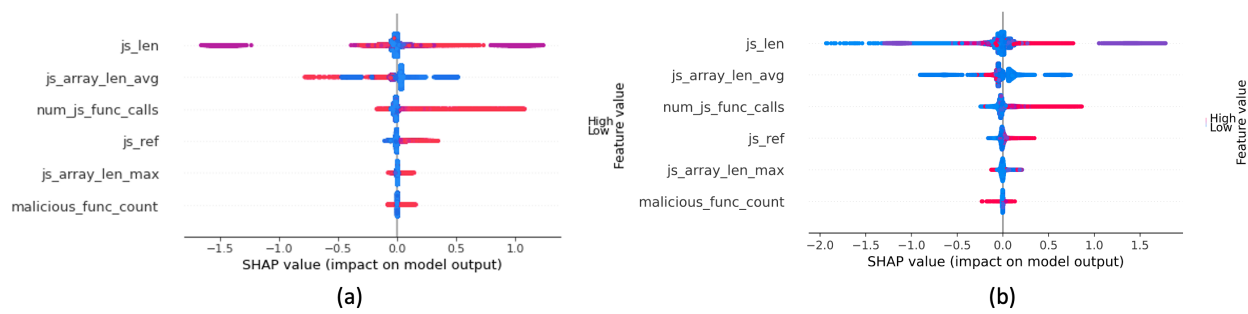


Figure 8. Two summaries of SHAP values: (a) full dataset; (b) dataset containing only JavaScript code.

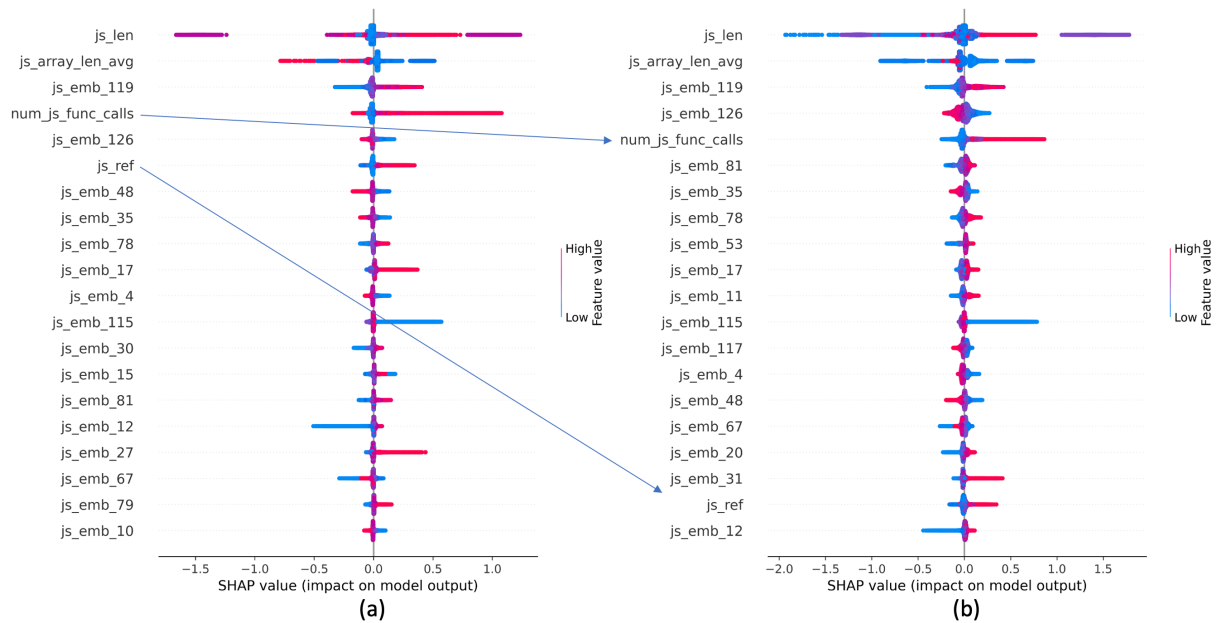


Figure 9. Top 20 features based on SHAP values: (a) full dataset; (b) dataset containing JavaScript code.

5.4. AST Feature Influence

In this experiment, we evaluated the influence of each feature category on a particular group of webpages that probably have a different contribution distributions for detecting malicious webpages. This analysis consisted of two main steps: clustering and feature importance analysis. The clustering step aimed to obtain some group clusters with similar features in the vector space to analyze the influence of feature categories based on SHAP values. Another consideration for clustering instead of webpage type labeling is that we need the ground truth, which is challenging. In addition, many webpages in the dataset are offline, which makes it hard to access them and identify their type.

Table 7 shows the clustering results using the BIRCH algorithm [54]. Clusters 0 and 2 have many similar webpages compared to clusters 1, 3, and 4. We also identified each cluster based on the percentage of unique hostnames per total number of URLs in each cluster. In Table 8, we can see how attackers try to attack the target with a single webpage or inject the payload into some webpages with the same hostname. The high percentage in cluster 2 indicates that most webpages are single webpages, where attackers probably create the webpages randomly without making a complex website.

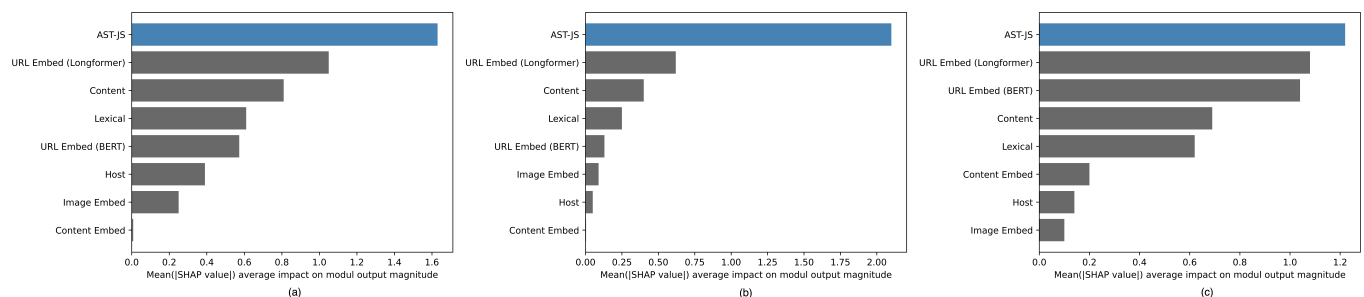
Table 7. Clustering results.

Cluster	Benign	Malicious
0	1568	712
1	123	33
2	21,879	10,857
3	12	3
4	2	0

Table 8. Percentage of unique hostnames per URL in each cluster.

Cluster	Label	Percentage (%)
0	Benign	57
	Malicious	58
1	Benign	42
	Malicious	64
2	Benign	78
	Malicious	78
3	Benign	17
	Malicious	67
4	Benign	50
	Malicious	0

For the following analysis, we evaluated the influences of all feature categories to determine whether they are beneficial for building a detection model. In Figure 10, we show a summary of the total average SHAP value for each feature category. As we can see in the graph for cluster 2, even though the AST has the most significant influence, URL information still has a strong influence because we can find many single webpages that tend to have a randomized or unique URL name. We did not analyze clusters 3 and 4 due to a lack of data.

**Figure 10.** Total average SHAP values for (a) cluster 0, (b) cluster 1, and (c) cluster 2.

6. Conclusions

This study involved the development of a new approach for improving malicious webpage detection based on analyzing the AST structure of JS coding. This addition supports a deeper analysis of the semantic meaning of JS code, whereas the previous work [44] did not include that feature for building the model. Our evaluation shows that adding an AST-JS feature representation can improve the performance of the detection model, especially for webpages with JS content. Furthermore, our evaluation shows the significant influence of AST-JS on the detection model to detect either benign or malicious webpages, and it also is dominant compared to other JS-related features. We found that the AST-JS and URL features share a strong influence on some webpages that tend to be single websites that may have randomized URLs.

However, our proposed approach only works well on webpages with JS content, which is the limitation of this study. Many malicious webpages without JS content probably

focus on phishing attacks, which do not need complicated payloads. Moreover, this study did not include real-time application analysis as a plug-in in web browsers to check the actual performance in real conditions. In addition, due to the use of a machine learning model, our proposed approach is susceptible to an adversarial attack that can affect network parameters, making our model produce many false negatives.

Author Contributions: Conceptualization, M.F.R., S.O. and T.B.; methodology, M.F.R., S.K., S.O., T.B. and T.T.; software, M.F.R.; validation, S.K., S.O., T.B., T.T. and D.I.; formal analysis, M.F.R.; investigation, M.F.R.; resources, S.O. and D.I.; data curation, M.F.R.; writing—original draft preparation, M.F.R.; writing—review and editing, M.F.R., S.K., S.O., T.B., T.T. and D.I.; visualization, M.F.R.; supervision, S.K., S.O., T.B., T.T. and D.I.; funding acquisition, D.I. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially supported by JSPS/MEXT KAKENHI Grant Numbers JP21H03444 and JP21KK0178.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Carroll, F.; Adejobi, J.A.; Montasari, R. How Good Are We at Detecting a Phishing Attack? Investigating the Evolving Phishing Attack Email and Why it Continues to Successfully Deceive Society. *SN Comput. Sci.* **2022**, *3*, 170. [CrossRef] [PubMed]
2. Stack Overflow Annual Developer Survey 2022. Available online: <https://insights.stackoverflow.com/survey> (accessed on 14 November 2022).
3. Symantec Security Response: What You Need to Know about the WannaCry Ransomware. Available online: <https://www.symantec.com/blogs/threat-intelligence/wannacry-ransomware-attack> (accessed on 19 January 2021).
4. Cross-Site Scripting. Available online: https://developer.mozilla.org/en-US/docs/Glossary/Cross-site_scripting (accessed on 18 January 2021).
5. Joshi, A.; Lloyd, L.; Westin, P.; Seethapathy, S. Using Lexical Features for Malicious URL Detection—A Machine Learning Approach. *arXiv* **2019**, arXiv:1910.06277.
6. Lin, Y.; Liu, R.; Divakaran, D.M.; Ng, J.Y.; Chan, Q.Z.; Lu, Y.; Si, Y.; Zhang, F.; Dong, J.S. Phishpedia: A Hybrid Deep Learning Based Approach to Visually Identify Phishing Webpages. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*; USENIX Association: Berkeley, CA, USA, 2021; pp. 3793–3810.
7. Hess, S.; Satam, P.; Ditzler, G.; Hariri, S. Malicious HTML File Prediction: A Detection and Classification Perspective with Noisy Data. In *Proceedings of the 2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*, Aqaba, Jordan, 28 October–1 November 2018; pp. 1–7. [CrossRef]
8. Rashid, J.; Mahmood, T.; Nisar, M.W.; Nazir, T. Phishing Detection Using Machine Learning Technique. In *Proceedings of the 2020 First International Conference of Smart Systems and Emerging Technologies (SMARTTECH)*, Riyadh, Saudi Arabia, 3–5 November 2020; pp. 43–46. [CrossRef]
9. Canfora, G.; Mercaldo, F.; Visaggio, C.A. Malicious JavaScript Detection by Features Extraction. *E-Inform. Softw. Eng. J.* **2014**, *8*, 65–78. [CrossRef]
10. Rozi, M.F.; Kim, S.; Ozawa, S. Deep Neural Networks for Malicious JavaScript Detection Using Bytecode Sequences. In *Proceedings of the 2020 International Joint Conference on Neural Networks (IJCNN)*, Glasgow, UK, 19–24 July 2020; pp. 1–8.
11. Narayanan, A.; Chandramohan, M.; Venkatesan, R.; Chen, L.; Liu, Y.; Jaiswal, S. graph2vec: Learning Distributed Representations of Graphs. *arXiv* **2017**, arXiv:1707.05005.
12. PhishTank Dataset. Available online: <https://phishtank.org/> (accessed on 19 October 2022).
13. jwSpamSpy. Available online: <https://www.jwspamspy.com/> (accessed on 19 October 2022).
14. DNS-BH. Available online: <https://github.com/epix-dev/dns-bh> (accessed on 19 October 2022).
15. McAfee's SiteAdvisor. Available online: <https://www.mcafee.com/> (accessed on 19 October 2022).
16. Cisco IronPort Web Reputation. Available online: <https://www.cisco.com/> (accessed on 19 October 2022).
17. Trend Micro Web Reputation. Available online: <https://global.sitesafety.trendmicro.com/> (accessed on 19 October 2022).
18. Ahammad, S.H.; Kale, S.D.; Upadhye, G.D.; Pande, S.D.; Babu, E.V.; Dhumane, A.V.; Bahadur, M.D.K.J. Phishing URL detection using machine learning methods. *Adv. Eng. Softw.* **2022**, *173*, 103288. [CrossRef]
19. Oram, E.; Dash, P.B.; Naik, B.; Nayak, J.; Vimal, S.; Nataraj, S.K. Light gradient boosting machine-based phishing webpage detection model using phisher website features of mimic URLs. *Pattern Recognit. Lett.* **2021**, *152*, 100–106. [CrossRef]

20. Wang, C.; Chen, Y. TCURL: Exploring hybrid transformer and convolutional neural network on phishing URL detection. *Knowledge-Based Syst.* **2022**, *258*, 109955. [CrossRef]
21. Alshehri, M.; Abugabah, A.; Algarni, A.; Almotairi, S. Character-level word encoding deep learning model for combating cyber threats in phishing URL detection. *Comput. Electr. Eng.* **2022**, *100*, 107868. [CrossRef]
22. Xiao, X.; Xiao, W.; Zhang, D.; Zhang, B.; Hu, G.; Li, Q.; Xia, S. Phishing websites detection via CNN and multi-head self-attention on imbalanced datasets. *Comput. Secur.* **2021**, *108*, 102372. [CrossRef]
23. Sun, G.; Zhang, Z.; Cheng, Y.; Chai, T. Adaptive segmented webpage text based malicious website detection. *Comput. Netw.* **2022**, *216*, 109236. [CrossRef]
24. Alani, M.M.; Tawfik, H. PhishNot: A Cloud-Based Machine-Learning Approach to Phishing URL Detection. *Comput. Netw.* **2022**, *218*, 109407. [CrossRef]
25. Gupta, B.B.; Yadav, K.; Razzak, I.; Psannis, K.; Castiglione, A.; Chang, X. A novel approach for phishing URLs detection using lexical based machine learning in a real-time environment. *Comput. Commun.* **2021**, *175*, 47–57. [CrossRef]
26. Chen, Y.; Zahedi, F.M.; Abbasi, A.; Dobolyi, D. Trust calibration of automated security IT artifacts: A multi-domain study of phishing-website detection tools. *Inf. Manag.* **2021**, *58*, 103394. [CrossRef]
27. Subasi, A.; Balfagih, M.; Balfagih, Z.; Alfawwaz, K. A Comparative Evaluation of Ensemble Classifiers for Malicious Webpage Detection. *Procedia Comput. Sci.* **2021**, *194*, 272–279. [CrossRef]
28. Mokbal, F.M.M.; Dan, W.; Xiaoxi, W.; Wenbin, Z.; Lihua, F. XGBXSS: An Extreme Gradient Boosting Detection Framework for Cross-Site Scripting Attacks Based on Hybrid Feature Selection Approach and Parameters Optimization. *J. Inf. Secur. Appl.* **2021**, *58*, 102813. [CrossRef]
29. Alex, S.; Rajkumar, T.D. Spider bird swarm algorithm with deep belief network for malicious JavaScript detection. *Comput. Secur.* **2021**, *107*, 102301. [CrossRef]
30. Wang, Q.; Yang, H.; Wu, G.; Choo, K.K.R.; Zhang, Z.; Miao, G.; Ren, Y. Black-box adversarial attacks on XSS attack detection model. *Comput. Secur.* **2022**, *113*, 102554. [CrossRef]
31. Liu, Z.; Fang, Y.; Huang, C.; Han, J. GraphXSS: An efficient XSS payload detection approach based on graph convolutional network. *Comput. Secur.* **2022**, *114*, 102597. [CrossRef]
32. Shi, K.; Lu, Y.; Chang, J.; Wei, Z. PathPair2Vec: An AST path pair-based code representation method for defect prediction. *J. Comput. Lang.* **2020**, *59*, 100979. [CrossRef]
33. Shippey, T.; Bowes, D.; Hall, T. Automatically identifying code features for software defect prediction: Using AST N-grams. *Inf. Softw. Technol.* **2019**, *106*, 142–160. [CrossRef]
34. Wu, Q.; Liu, Q.; Zhang, Y.; Wen, G. TrackerDetector: A system to detect third-party trackers through machine learning. *Comput. Netw.* **2015**, *91*, 164–173. [CrossRef]
35. Marashdih, A.W.; Zaaba, Z.F.; Suwais, K. Predicting input validation vulnerabilities based on minimal SSA features and machine learning. *J. King Saud Univ. Comput. Inf. Sci.* **2022**. [CrossRef]
36. Gao, X.; Jiang, X.; Wu, Q.; Wang, X.; Lyu, C.; Lyu, L. GT-SimNet: Improving code automatic summarization via multi-modal similarity networks. *J. Syst. Softw.* **2022**, *194*, 111495. [CrossRef]
37. Ndichu, S.; Kim, S.; Ozawa, S.; Misu, T.; Makishima, K. A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors. *Appl. Soft Comput.* **2019**, *84*, 105721. [CrossRef]
38. Fang, Y.; Huang, C.; Zeng, M.; Zhao, Z.; Huang, C. JStrong: Malicious JavaScript detection based on code semantic representation and graph neural network. *Comput. Secur.* **2022**, *118*, 102715. [CrossRef]
39. Rozi, M.F.; Ban, T.; Ozawa, S.; Kim, S.; Takahashi, T.; Inoue, D. JStrack: Enriching Malicious JavaScript Detection Based on AST Graph Analysis and Attention Mechanism. In *Proceedings of the Neural Information Processing*; Mantoro, T., Lee, M., Ayu, M.A., Wong, K.W., Hidayanto, A.N., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 669–680.
40. Jones, J. Abstract Syntax Tree Implementation Idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, Monticello, IL, USA, 8–12 September 2003.
41. The ESTree Spec. Available online: <https://github.com/estree/estree> (accessed on 20 January 2021).
42. Hamilton, W.L. Graph Representation Learning. *Synth. Lect. Artif. Intell. Mach. Learn.* **2020**, *14*, 1–159.
43. Lundberg, S.; Lee, S.I. A Unified Approach to Interpreting Model Predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*, Long Beach, CA, USA, 4–9 December 2017; pp. 4768–4777. [CrossRef]
44. Chaiban, A.; Sovilj, D.; Soliman, H.; Salmon, G.; Lin, X. Investigating the Influence of Feature Sources for Malicious Website Detection. *Appl. Sci.* **2022**, *12*, 2806. [CrossRef]
45. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*; IEEE Computer Society: Los Alamitos, CA, USA, 2018; pp. 4510–4520. [CrossRef]
46. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2020; pp. 1536–1547. [CrossRef]
47. Beltagy, I.; Peters, M.E.; Cohan, A. Longformer: The Long-Document Transformer. *arXiv* **2020**, arXiv:2004.05150.
48. Sanh, V.; Debut, L.; Chaumond, J.; Wolf, T. DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *arXiv* **2019**, arXiv:1910.01108.

49. Esprima. Available online: <https://esprima.org/> (accessed on 26 January 2021).
50. Han, K.; Hwang, S.O. Lightweight Detection Method of Obfuscated Landing Sites Based on the AST Structure and Tokens. *Appl. Sci.* **2020**, *10*, 6116. [\[CrossRef\]](#)
51. Le, Q.V.; Mikolov, T. Distributed Representations of Sentences and Documents. *arXiv* **2014**, arXiv:1405.4053.
52. Ribeiro, M.T.; Singh, S.; Guestrin, C. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 1135–1144. [\[CrossRef\]](#)
53. Shrikumar, A.; Greenside, P.; Kundaje, A. Learning Important Features through Propagating Activation Differences. In Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, 6–11 August 2017; Volume 70, pp. 3145–3153.
54. Zhang, T.; Ramakrishnan, R.; Livny, M. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Association for Computing Machinery (SIGMOD '96), Montreal, QC, Canada, 4–6 June 1996; pp. 103–114. [\[CrossRef\]](#)
55. Fass, A.; Backes, M.; Stock, B. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019.