applied sciences

MDPI

Article

Lightweight Detection Method of Obfuscated Landing Sites Based on the AST Structure and Tokens

KyungHyun Han [1] and Seong Oun Hwang [2,*]

[1] Department of Electronics and Computer Engineering, Hongik University, Sejong 30016, Korea; co112kr@mail.hongik.ac.kr
[2] Department of Computer Engineering, Gachon University, Seongnam 13120, Korea
* Correspondence: sohwang@gachon.ac.kr; Tel.: +82-31-750-5327

Received: 17 July 2020; Accepted: 1 September 2020; Published: 3 September 2020

Abstract: Attackers use a variety of techniques to insert redirection JavaScript that leads a user to a malicious webpage, where a drive-by-download attack is executed. In particular, the redirection JavaScript in the landing site is obfuscated to avoid detection systems. In this paper, we propose a lightweight detection system based on static analysis to classify the obfuscation type and to promptly detect the obfuscated redirection JavaScript. The proposed model detects the obfuscated redirection JavaScript by converting the JavaScript into an abstract syntax tree (AST). Then, the structure and token information are extracted. Specifically, we propose a lightweight AST to identify the obfuscation type and the revised term frequency-inverse document frequency to efficiently detect the malicious redirection JavaScript. This approach enables rapid identification of the obfuscated redirection JavaScript and proactive blocking of the webpages that are used in drive-by-download attacks.

Keywords: malicious JavaScript; abstract syntax tree; static analysis; obfuscation detection; redirection detection

1. Introduction

The number of cyber-attacks that occur has continued to grow exponentially as web-related technologies and infrastructures have continually advanced. Among these attacks, the drive-by-download attack is widely employed in the web environment. Drive-by-download attacks are dangerous because malicious files are installed on the user's PC when the user visits a malicious webpage. As a result, subsequent attacks can be launched, even without the victim's awareness. Drive-by-download attacks exploit vulnerabilities in various applications, such as web browsers, plug-ins, and Active X, and they infect these applications through various means. Malicious files downloaded to a user's computer can harm the user by leaking the user's credentials, including personal information. This attack may also affect governments or any other organization by the subsequent execution of distributed denial of service attacks.

Attackers often intend to efficiently attack several users. To launch a successful assault, attackers insert their malicious redirection code in popular websites. The code will redirect the site visitors to a malicious page [1]. In other words, if a user visits a popular website, their computer can be infected by the installed malware. Popular websites that contain the embedded redirection JavaScript are known as landing sites, and the malicious page is called a distribution page. Users cannot directly distinguish a landing site from a legitimate site; consequently, malicious files are downloaded onto the user's computer without the user's consent, even if the user accesses only what they believe to be legitimate sites.

Kishore et al. [2] introduced a relation of drive-by-download and obfuscated JavaScript well and proposed their detection system. In addition to many systems have been developed to detect malicious

JavaScript. However, attackers can bypass these detection systems based on static analysis by applying obfuscation tools to the JavaScript. Obfuscation was originally used to prevent the direct exposure of source code; however, attackers began exploiting obfuscation tools to hide their malicious JavaScript. In this case, JavaScript engines recognize and interpret the obfuscated code. Such code is not easy for humans or detection systems to understand and analyze. Dynamic detection systems can also detect obfuscated JavaScript; however, their detection processes tend to be slow and time consuming.

To solve the above issue, a detection system is herein proposed that is based on static analysis using the abstract syntax tree (AST). It includes a similarity comparison method that detects obfuscated JavaScript with the use of minimal computational resources. Specifically, a lightweight AST and a modified algorithm of term frequency–inverse document frequency (TF-IDF) are proposed to detect malicious JavaScript with higher accuracy than the conventional approaches.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 presents our proposal of an obfuscated landing-page detection system. Section 4 analyzes the proposed system through an experiment. Finally, Section 5 provides our conclusions.

2. Related Work

To prevent drive-by-download attacks, researchers have proposed a variety of methods. Choi et al. [3] traced malicious websites in malware distribution networks. They classified malicious and normal websites based on the link structure. They focused on detecting malware distribution networks. This system is for analysts. However, desire landing-page detection because they do not want to be redirected to malicious pages. Therefore, the need exists for analysis of JavaScript on landing pages, which are the start point of attacks. Wang et al. [4] used two crawlers to find the landing pages. One static crawler first collects webpages and removes duplications. Based on the collected pages, the other dynamic crawler detects malicious behaviors, traces related malicious pages, and finally determines the landing pages. Then, we can collect and analyze JavaScript on landing pages.

Many studies have been conducted to detect malicious JavaScript. These studies are divided by static analysis and dynamic analysis. Dynamic analysis methods which can use JavaScript engine are higher detection accuracy, generally. Because attacker applied obfuscation technique to malicious JavaScript to avoid static analysis. Chellapilla et al. [5] and Yue et al. [6] developed methods to detect obfuscated malicious JavaScript using dynamic analysis. This detection method is inefficient in terms of time and resource consumption. Patil et al. [7] used dynamic analysis with machine learning. Their approach outperforms all of the nine well-known antivirus software programs in terms of malicious JavaScript detection accuracy. He et al. [8] developed methods to detect it using hybrid analysis. This detection method is faster than dynamic analysis methods, but it is still slow and decrease accuracy. We needed to research static analysis to provide high detection speed to user.

Generally, researchers tried to used AST to detect malicious JavaScript by using static analysis, which is efficient with respect to its time requirement. Curtsinger et al. [9] detected malicious JavaScript by extracting ASTs from JavaScript and by including features, such as the loop, function, string, if, and try. Finally, they analyzed these features using a Bayesian algorithm. However, their research was limited to detecting heap-spray attacks. Kapravelos et al. [10] compared the ASTs extracted from a new analysis target with those previously extracted from malicious and benign pages. Their research detected malicious JavaScript, which bypassed the dynamic analysis; nevertheless, it excluded redirection code detection. These methods are not optimized to detect obfuscated JavaScript, which is usually employed to redirect the distribution page.

Researchers have also proposed various methods to use machine learning with static analysis to detect obfuscated JavaScript. Wang et al. [11] used stacked denoising auto-encoders and logistic regression. They achieved an accuracy of up to 95%. Wei et al. [12] used convolutional neural network and bigram. They achieved an F-measure of 93–95%. Fang et al. [13] extracted features from the semantic level of bytecode on the document object model and the browser object model. Analysis is then conducted using long short-term memory (LSTM). They achieved 99.51% accuracy.

Furthermore, researchers tried to use static analysis based on combination of machine learning and AST. Al-Taharwa et al. [14] proposed a method to extract ASTs and analyze JavaScript using a Bayesian algorithm. This method detects malicious JavaScript, including obfuscated JavaScript, by extracting context-based features. However, it detects only readable obfuscated JavaScript because the context-based features are extracted only from readable obfuscated JavaScript. Blanc et al. [15] and Al-Taharwa et al. [16] developed methods to detect obfuscated JavaScript. To this end, they employed AST and machine learning. Although their methods can identify whether the JavaScript is obfuscated, they cannot distinguish whether the JavaScript is benign or malicious. Recently, Fang et al. [17] improved their previous system [13] by extracting ASTs and converting them into syntactic unit sequences. Then, they used the FastText algorithm to train word vectors. They used Bi-LSTM as a detection model. Although their method using machine learning is effective and generally efficient, it requires considerable resources to train the word vectors, maintain the deep learning model, and enable the detection model to be used continuously.

In conclusion, many methods have been proposed to detect landing sites. However, some of them are unable to detect obfuscated redirection JavaScript, while others require numerous computations. Therefore, the need exists for a method that can detect landing sites, including obfuscated redirection JavaScript, with fewer computational resources.

3. Proposed System

3.1. System Structure

As mentioned earlier, attackers insert their malicious redirection code into popular websites and redirect users to distribution sites [1]. Popular sites as landing sites are used to lead users to the distribution sites, which are employed to infect the users' computers. These two tasks are separated so that attackers can modify their malicious pages and URLs. It is difficult for many detection systems to efficiently track malicious pages because they are frequently changed. It is also difficult to change the redirection JavaScript on popular pages because such pages are not managed by the attackers themselves. Provos et al. discovered that 80% of drive-by-download networks share at least one landing site [18]. Landing page detection is an efficient method to deal with drive-by-download attacks because a detection system can find many malicious pages that are connected to the same landing pages.

Kishore et al. discovered that the most popular method to hide redirection JavaScript is obfuscation [2]. Attackers hide essential redirection signatures, such as iframe and src, which are used for redirection by means of obfuscation. However, Cova [19] determined that a page is a landing page if the redirection JavaScript on the page is obfuscated. Therefore, our proposed method analyzes whether an input JavaScript is obfuscated and contains redirection code.

Figure 1 shows the proposed system for detecting obfuscated landing sites based on static analysis methods. A website containing obfuscated redirection code can be presumed to be an obfuscated landing site, as mentioned above. The proposed system detects obfuscated landing sites by first determining whether a JavaScript is obfuscated, and then checking whether it contains redirection code. The system takes as input a dataset, which belongs to one of three cases: (1) an obfuscated landing site (obfuscated redirection code included), (2) obfuscated benign JavaScript (obfuscated, but redirection code is not included), and (3) unobfuscated benign JavaScript. Note that we do not consider the case of an unobfuscated landing site (unobfuscated redirection code included), which can be easily detected.

The proposed system consists of two major components: feature extraction and similarity computation. For feature extraction, we extract features to determine whether an input JavaScript containing redirection code is obfuscated. The features that our system handles are the structure and token of AST. AST is a key part of feature extraction. AST is suitable in analyzing obfuscated code because AST is good at finding out a syntax structure in code and obfuscated code has its own specific syntax structure for de-obfuscation. We decided that the structure is a set of nodes that can affect the

AST structure, and the token is a set of nodes that does not affect the AST structure. First, the input JavaScript is transformed to an AST form at the AST transformation step. Next, we classify all AST nodes as structures or tokens at the structure–token extraction step. The structure and token are then transferred to the similarity computation step.

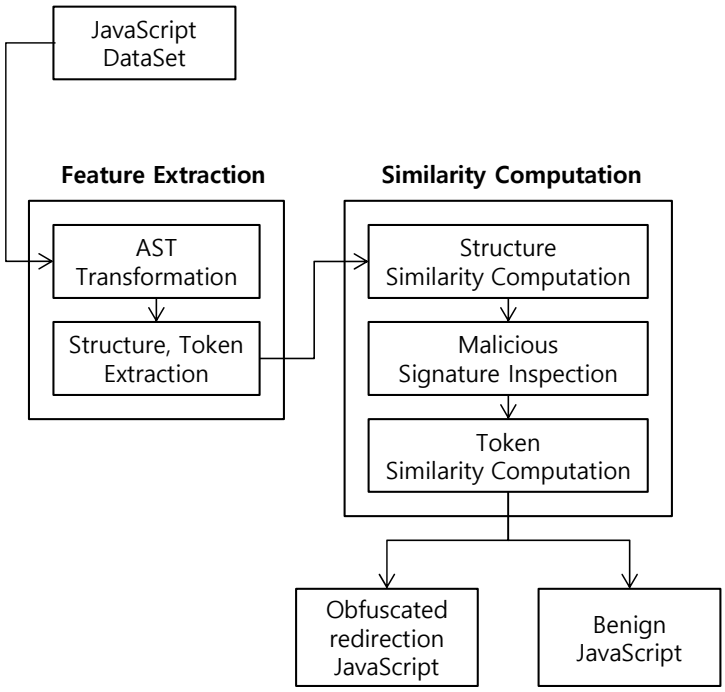

Figure 1. Proposed system.

In the case of obfuscated JavaScript, the conventional exact pattern matching approach is not suitable in detecting obfuscated JavaScript because input JavaScript is changed to various words or characters according to underlying specific obfuscator. The similarity computation approach, however, is suitable because its decision is based on the similarity with the previous, labeled JavaScript even though both do not perfectly match. During similarity computation, the extracted structure and token will be used to identify the obfuscation type and redirection code, respectively. The tokens that are needed to check the redirection code differ depending on the obfuscation type. Therefore, we first identify the type of obfuscation being used. During the structure similarity computation step, we compute the similarities between the input JavaScript to be tested and the obfuscated JavaScript sample in each obfuscation type in the syntax structure. Then, if it is obfuscated, we identify the obfuscation type. We use SimHash to calculate the similarity at this step. SimHash is a very powerful hash that is used to find duplications in crawling webpages by Google [20]. In this step, SimHash is used with a lightweight AST.

At the token similarity computation step, we check whether the JavaScript contains redirection code by using the average of TF-IDF (we will explain TF-IDF in Section 3.3) of tokens corresponding to the obfuscation type. The malicious signature inspection step is in between two steps, i.e., the structure similarity computation step and the token similarity computation step. In this step, we inspect the signatures in the input JavaScript. Note that the signatures consist of strings used in advance in the redirection code. When a signature is found, we perform the token similarity computation step. Otherwise, we do not perform the next step because the input JavaScript is presumed to be benign obfuscated JavaScript. In this way, malicious signature inspection reduces the scope of the dataset to be tested in the token similarity computation step.

3.2. Lightweight AST

In the conventional AST, some nodes serve as noise by reducing AST similarities. Some other nodes do not affect the AST similarity computation. Hence, we define below two types of nodes.

Definition 1. *An AST nonstructural node is a node whose AST structure is not affected.*

For example, identifier and literal nodes in the ECMAScript standard [21] do not affect the AST structure. They only change their attribute values, e.g., 'name' in the identifier node and 'value' in the literal node. Other examples of an AST nonstructural node are nodes whose attributes have a single type only. These nodes do not generate children nodes; rather, they include 'type' attributes only to specify their node type. However, if the type attributes of these nodes are different, the similarity is calculated to be a lower value, even though the AST structures are the same. The token mentioned in Figure 1 belongs to the AST nonstructural node.

Definition 2. *An AST structural node is a node that is not an AST nonstructural node.*

All nodes except the AST nonstructural nodes are structural nodes. These nodes can have children nodes when they obtain other attribute values in addition to the attribute type. In other words, if the attribute types of these nodes are different, the AST structure can be differentiated by connecting with the other children nodes. The structure mentioned in Figure 1 belongs to the AST structural node.

Definition 3. *A lightweight AST is an AST that excludes AST nonstructural nodes.*

According to the above definition, an AST consists of an AST structural node and an AST nonstructural node. Hence, we can set up a lightweight AST that consists of AST structural nodes only by pruning out AST nonstructural nodes. A lightweight AST is helpful for calculating similarity because it represents the structure.

Figure 2 shows an algorithm that extracts a lightweight AST and tokens from JavaScript. The data structures are initialized, such as *NS_NODE* for AST nonstructural nodes, S_i for storing AST structural nodes and trees, and T_i for AST nonstructural nodes (line 3). Parse trees are generated and AST nonstructural nodes are extracted from input JavaScript (lines 4–14). First, parse trees are generated and stored at S_i per input JavaScript (lines 4–6). Next, each node is visited on the parse trees, and whether it is an AST nonstructural node is checked. If 'yes', it is added to T_i and removed from S_i. This process is repeated for all nodes in the parse trees (lines 7–14). The function ast tree () takes S_i as input and stores the resulting output, AST without AST nonstructural node, to S_i.

3.2.1. Obfuscation-Type Classification Accuracy

Theorem 1. *The proposed lightweight AST enables determination of the type of obfuscation algorithm.*

Proof of Theorem 1. Each obfuscation algorithm has its own syntax structure. This syntax structure is generated by each obfuscation algorithm and is used to restore the original code. The original code has an effect on AST nonstructural nodes; however, it has no effect on the syntax structure. The proposed lightweight AST is defined by AST, where all of the AST nonstructural nodes are removed. A lightweight AST has AST structural nodes only. In other words, a lightweight AST has its own syntax structure according to each obfuscation algorithm. Therefore, we can identify the type of obfuscation algorithm being used by employing the lightweight AST. □

```
1  Input: Javascript files: f_i (1≤i≤n)
2  Output: Set of AST structures: S_i (1≤i≤n), Set of AST tokens: T_i (1≤i≤n)
3  Initialization: Set of AST non-structural node type: NS_NODE
                   Structure set: S_i (1≤i≤n), Token set: T_i (1≤i≤n)
4  fn = get_file_number(f_i);
5  for i = 1 to fn with step = 1 do
6     S_i = parse_tree(f_i);
7     n = get_node_number(S_i);
8     for j = 1 to n with step = 1 do
9        cn = visit_node (S_i, j);
10       if node_type(cn) = NS_NODE then
11          add_token(T_i, cn);
12          remove_node(S_i, cn);
13       end if
14    end for
15  S_i = ast_tree(S_i);
16 end for
```

Figure 2. Lightweight AST and token extraction algorithm.

3.2.2. Reduction Rate of the Lightweight AST

In this section, we calculate the representative minimum node reduction rate (theoretically, per each input JavaScript) when generating lightweight ASTs. The higher reduction rate of the lightweight AST reduces the comparison time.

According to the ECMAScript 5 standard, there are expressions and statements that can be repeated in AST. The reduction rate differs depending on how frequently the expressions and statements can be repeated in lower nodes. We set E and S as the number of repetitions of expressions and statements, respectively (E, $S \geq 0$). When generating an AST, we calculate the node reduction rate of each node type using the following formula in the AST:

Node reduction rate = #nonstructural node/(#structural node + #nonstructural node)

We define the smallest minimum value among all these reduction rates as the representative minimum node reduction rate. Table 1 shows the minimum node reduction rates of each node type, as E and S vary. Among the nodes shown in the table, SwitchStatement and TryStatement have comparatively lower minimum node reduction rates than the other node types. The smallest minimum value of the two minimum node reduction rates is the representative minimum node reduction rate. The smallest minimum value can be determined depending on both E and S. Therefore, the representative minimum node reduction rate is given as

$$\begin{gathered} \text{If } E < 2S+2, \frac{1}{2S+3}, \\ \text{If } E > 2S+2, \frac{S+2}{(S+1)(E+2)+(S+2)}, \\ \text{If } E = 2S+2, \frac{1}{2S+3} = \frac{S+2}{(S+1)(E+2)+(S+2)} \end{gathered}$$

Table 1. Minimum node reduction rate of each AST node type.

Type of AST Node	Nonstructural Node Ratio	Type of AST Node	Nonstructural Node Ratio
Programs	$\frac{1}{2S+2}$	SequenceExpression	$\frac{1}{E+2}$
Identifier	1	ExpressionStatement	$\frac{1}{E+2}$
Literal	1	BlockStatement	$\frac{1}{S+2}$
FunctionDeclaration	$\frac{1}{2}$	EmptyStatement	1
VariableDeclaration	$\frac{1}{3}$ or $\frac{2}{E+4}$	DebuggerStatement	1
ThisExpression	1	WithStatement	$\frac{S+2}{(S+1)(E+1)+(S+2)}$

Table 1. *Cont.*

Type of AST Node	Nonstructural Node Ratio	Type of AST Node	Nonstructural Node Ratio
ArrayExpression	$\frac{1}{E+2}$	ReturnStatement	$\frac{1}{E+2}$ or 1
ObjectExpression	$\frac{E+2}{3E+4}$	LabeledStatement	$\frac{S+2}{2S+3}$
FunctionExpression	1	BreakStatement	$\frac{1}{2}$ or 1
UnaryExpression	$\frac{E+2}{2E+3}$	ContinueStatement	$\frac{1}{2}$ or 1
UpdateExpression	$\frac{E+2}{2E+3}$	IfStatement	$\frac{S+2}{(S+1)(E+1)+(S+2)}$
BinaryExpression	$\frac{2^{E+2}-1}{2^{E+1}\cdot 3-2}$	SwitchStatement	$\frac{S+2}{(S+1)(E+2)+(S+2)}$
AssignmentExpression	$\frac{2^{E+2}-1}{2^{E+1}\cdot 3-2}$ or $\frac{2E+3}{3E+4}$	ThrowStatement	$\frac{1}{E+2}$
LogicalExpression	$\frac{2^{E+2}-1}{2^{E+1}\cdot 3-2}$	TryStatement	$\frac{1}{2S+3}$
MemberExpression	$\frac{2^{E+1}}{2^{E+2}-1}$	WhileStatement	$\frac{S+2}{(S+1)(E+1)+(S+2)}$
ConditionalExpression	$\frac{3^{E+1}}{1+\sum_{k=1}^{E+1}3^k}$	DoWhileStatement	$\frac{S+2}{(S+1)(E+1)+(S+2)}$
CallExpression	$\frac{2^{E+1}}{2^{E+2}-1}$	ForStatement	$\frac{1}{S+2}$
NewExpression	1	ForInStatement	$\frac{2S+3}{(S+1)(E+3)+(2S+3)}$

As shown in Table 2, we analytically compute the representative minimum node reduction rate for 23 different obfuscation types. This result is obtained by first computing the maximum number of expression and statement repetitions and then applying them to the minimum node reduction formula.

Table 2. Analytical representative minimum node reduction rate according to obfuscation type.

Obfuscation Type	*E*	*S*	Analytical Representative Minimum Node Reduction Rate
Type 1	9	8	5.2%
Type 2	5	6	6.6%
Type 3	5	3	11.1%
Type 4	5	3	11.1%
Type 5	8	3	11.1%
Type 6	6	2	14.2%
Type 7	3	3	11.1%
Type 8	4	5	7.6%
Type 9	4	9	4.7%
Type 10	5	3	11.1%
Type 11	5	3	11.1%
Type 12	7	5	7.6%
Type 13	5	3	11.1%
Type 14	3	3	11.1%
Type 15	7	3	11.1%
Type 16	7	7	5.8%
Type 17	4	1	20.0%
Type 18	5	1	12.5%
Type 19	6	5	7.6%
Type 20	6	6	6.6%
Type 21	7	5	7.6%
Type 22	11	1	10.3%
Type 23	6	10	4.3%
Average			9.59%

Theorem 2. *The proposed lightweight AST reduces the structure comparison time with a higher rate than the node reduction rate compared to the conventional AST.*

Proof of Theorem 2. We use SimHash to compare the ASTs. The complexity of SimHash is O(*nlogn*), where *n* is larger than 1 because it denotes the number of AST nodes. In other words, the structure comparison time and *n* are proportional. Let x be the node reduction rate. Hence, the structure comparison time reduction rate is calculated as follows:

Structure comparison time reduction rate.

$$\begin{aligned} &= 1 - \frac{\text{the time complexity of lightweight AST}}{\text{the time complexity of conventional AST}} \\ &= 1 - \frac{\text{the time complexity of lightweight AST}}{\text{the time complexity of conventional AST}} \\ &= 1 - \frac{(1-x)n\ \log(1-x)n}{n\ \log n} \\ &= 1 - \frac{(1-x)\log(1-x)n}{\log n} \\ &= 1 - \frac{(1-x)(\log(1-x)+\log n)}{\log n}. \end{aligned}$$

Here, n is larger than $(1-x)n$. The maximum reduction rate is 1 because $\frac{\log(1-x)n}{\log n}$ is 0 when $(1-x)n$ is 1, and the minimum reduction rate is x because $\lim_{n\to\infty} \frac{(1-x)(\log(1-x)+\log n)}{\log n} = 1-x$. Therefore, the structure comparison time reduction rate is higher than the node reduction rate, which ranges from x to 1.

In addition, the extraction time of the lightweight AST is longer than that of the conventional one. Nevertheless, it can be ignored because the process that distinguishes the AST structural node and AST nonstructural node occurs within the same time period as the AST transformation. Moreover, it has an O(*n*) complexity that is lower than O(*nlogn*) of SimHash. Therefore, the lightweight AST reduces the structure comparison time. □

3.3. Revised TF-IDF

Obfuscation is performed to protect the internal information or structure of an algorithm from web crawlers. Obfuscation usually divides words and then recomposes them, which makes it difficult to detect a redirection JavaScript. We classified the obfuscation type in the previous step, whereby we could detect the redirection JavaScript by detecting the divided redirection words according to the obfuscation type. We refer to the divided redirection JavaScript as the redirection token. Note that we cannot determine if the JavaScript, including the redirection token, is malicious because the redirection token can be generated by other words that are not redirection code.

Therefore, we use the TF-IDF to compare how many redirection tokens are included in each JavaScript. Let us analyze the total number of JavaScripts, represented as $javascript_{total}$, and calculate the TF-IDF for redirection token *t*. When we analyze the *i*-th JavaScript, represented as $javascript_i$, the conventional TF-IDF is calculated as

$$\text{tf} - \text{idf} = \text{the number of } t \text{ in } javascript_i \times \frac{javascript_{total}}{\text{the number of javascript including } t}.$$

The conventional TF-IDF represents how many more *t's* are included in the *i*-th JavaScript than in the other JavaScript. However, the obfuscated code of a benign page is generally long, unlike the obfuscated redirection JavaScript, which is short because attackers aim to hide. Obfuscation algorithms generate a number of redirection tokens, even if a benign page does not have redirection code. Therefore, it mistakenly outputs higher values because the number of redirection tokens is extremely large. To avoid this problem, we revise the conventional TF-IDF by dividing the number of redirection tokens to be analyzed by the total number of all tokens using the following definition:

Definition 4. *Revised TF-IDF.*

$$\text{revised tf} - \text{idf} = \frac{\text{Number of } t \text{ in } javascript_i}{\text{Number of all } tokens \text{ in } javascript_i} \times \frac{javascript_{total}}{\text{Number of javascript including } t}.$$

Property 1. *The proposed TF-IDF effectively distinguishes the redirection JavaScript compared to the original TF-IDF.*

The number of all tokens in $javascript_i$ is used to filter out obfuscated benign JavaScript. We obtain TF by normalizing the ratio of the total number of JavaScripts to the number of JavaScripts containing redirection tokens. Accordingly, we can obtain lower revised TF-IDF values by the given formula while analyzing the obfuscated benign JavaScript. This is because, even if the obfuscation algorithms generate several redirection tokens, they also generate several other tokens, excluding redirection tokens. In obfuscated redirection JavaScript, we obtain large revised TF-IDF values because they have many redirection tokens compared to their sizes. Hence, we can obtain TF-IDF values that can distinguish between benign JavaScript and obfuscated redirection JavaScript with higher accuracy.

4. Experiment

4.1. Experimental Environment

The experiment of this study was performed on an Ubuntu 16.04 64-bit operating system with an AMD Phenom(tm) 2 X4 980 Processor, 3.70 GHz CPU, and 8 GB RAM. 1000 of benign JavaScript samples were collected from Alex top 100 sites using web crawlers. The collected JavaScripts were first verified to be benign by using the Google Safe Browsing API [22] and then by manual assessment. 125 of malicious JavaScript samples were collected from the CYDRON project, which provided malicious URLs at malwares.com. In addition, we selected 25 of benign JavaScript randomly and 125 of malicious JavaScript to generate obfuscated JavaScript using obfuscation tools, including Dean Edwards Packer, JavaScript Encryptor, JavaScript Obfuscator, and Text Encoding Obfuscation. In this way we constructed our dataset including 1000 unobfuscated benign JavaScripts, 100 obfuscated benign JavaScripts, and 500 obfuscated malicious JavaScripts as shown in Table 3. We implemented our system as independent modules. The input of our system is JavaScript files and the output is a detection result. Therefore, any detection system like antivirus software or browser plugin, which takes JavaScript file from URL as input, can work combined with our system. The source code of our system is publicly available online at [23].

Table 3. JavaScript type and dataset size.

JavaScript Type	Benign	Malicious
Alex Top 100	1000	-
Dean Edwards Packer	25	125
JavaScript Encryptor	25	125
JavaScript Obfuscator	25	125
Text Encoding Obfuscation	25	125
Total	1100	500

4.2. Lightweight AST

In the following, we compare the conventional AST and lightweight AST. It is shown that the lightweight AST classifies obfuscated JavaScript better than the conventional one in terms of time efficiency and detection accuracy. Table 4 presents the comparison results of the practical minimum node reduction rate according to the obfuscation type required to generate ASTs.

Table 4. Practical minimum node reduction rate according to obfuscation type.

Obfuscation Type	Number of AST Structural Nodes	Number of AST Nonstructural Nodes	Reduction Rate
Type 1	52	49	48.51%
Type 2	30	28	48.28%
Type 3	36	32	47.06%
Type 4	22	23	51.11%
Type 5	238	237	49.89%
Type 6	147	130	46.93%
Type 7	23	22	48.89%
Type 8	91	84	48.00%
Type 9	45	39	46.43%
Type 10	30	24	44.44%
Type 11	33	27	45.00%
Type 12	50	43	46.24%
Type 13	49	47	48.96%
Type 14	33	30	47.62%
Type 15	30	25	45.45%
Type 16	206	148	41.81%
Type 17	36	38	51.35%
Type 18	36	38	51.35%
Type 19	72	60	45.45%
Type 20	83	78	48.45%
Type 21	72	57	44.19%
Type 22	18	16	47.06%
Type 23	125	100	44.44%
Average			47.26%

4.2.1. Analysis of the Average Reduction Rate of Lightweight ASTs

Table 4 also shows the percentage of nodes produced while generating lightweight ASTs for samples of 23 different obfuscation types. On average, we can reduce the number of nodes by 47.26% during the generation process. To generally validate the average reduction rate, we performed a *t*-test using a statistical significance of 0.05 under hypothesis H_0, which states that the average reduction rate during JavaScript obfuscation using a lightweight AST is 47% as follows:

H_0 (null hypothesis): $\mu = 47$, H_1 (alternative hypothesis): $\mu \neq 47$, where μ is the population mean.

We compute the sample mean $(\overline{x}) = 47.26$ and the sample standard deviation $s = 2.47$.

The *t*-statistic is computed as $t = \frac{\overline{x}-\mu_0}{s/\sqrt{n}} = \frac{47.26-47}{2.47/\sqrt{23}} = 0.504$.

According to the statistical significance of 0.05, t-statistic to reject null hypothesis must be higher than 1.960, but it is 0.504. Thus, we cannot reject the null hypothesis H_0: $\mu = 47$. Therefore, we can conclude that the average node reduction rate is 47%.

The experimental average reduction rate of 47.26% for the samples of 23 different obfuscation types is greater than the theoretical average reduction rate of 9.59%, as shown in Table 2. This is because the representative minimum node reduction rate formula is established based on the expression and statement, which minimize the number of AST nonstructural nodes. In the real JavaScript environment, the expression and statement are used such that the AST nonstructural node is not minimal. Therefore, when generating lightweight ASTs, the rate of reduced nodes increases.

4.2.2. Analysis of the Structure Similarity Computation Time

Table 5 shows the comparison results of the structure similarity computation time. On average, the lightweight AST requires 3.021 s, which is 24.15% less than the 3.983 s of the conventional AST. In particular, the obfuscated JavaScript comparison time is reduced more than 47%. This result is in accordance with those of Theorem 2 and *t*-test. The effect is small in the unobfuscated JavaScript because it does not have a special structure compared to its size. However, in the obfuscated JavaScript case, the effect is large when using the lightweight AST because it has many tokens by which words are divided by obfuscation.

Table 5. Structure similarity computation time.

JavaScript Set	Structure Similarity Computation Time with Conventional AST	Structure Similarity Computation Time with Lightweight AST	Reduction Rate
1000 unobfuscated benign samples	8.404 s	7.264 s	13.56%
1000 obfuscated benign samples	1.252 s	0.661 s	47.20%
1000 obfuscated malicious samples	2.293 s	1.139 s	50.32%
Average	3.983 s	3.021 s	24.15%

4.2.3. Classification Accuracy of Obfuscation Type Using Lightweight AST

We here conduct an experiment in which we compare the accuracies when obfuscation classification is respectively based on the conventional AST and the lightweight AST. First, we prepare an experimental JavaScript set that consists of both a non-obfuscated JavaScript set (1000 JavaScripts) and obfuscated JavaScript sets of four types (125 JavaScript sets per type by using four different obfuscation tools). Then, we randomly select a representative JavaScript from the type 1 obfuscated JavaScript set and convert it into a conventional AST.

Next, we convert all JavaScripts from the experimental JavaScript set into conventional ASTs and measure their similarities using a representative JavaScript. We set a threshold to classify the JavaScript that has a structural similarity exceeding specific criteria as obfuscated JavaScript of type 1. We repeat the same process for types 2, 3, and 4. In this way, we can measure the accuracies when the obfuscation classification is based on the conventional AST. We can likewise measure the accuracies when the obfuscation classification is based on the lightweight AST.

Figure 3 shows the structural similarities measured between the representative type 1 obfuscated JavaScript and all JavaScripts in the conventional AST. The structural similarity ranges from 0 to 1. The larger the number is, the higher the similarity is. Even though the representative JavaScript is selected from the type 1 obfuscated JavaScript set, the latter does not show comparatively higher similarities than the others.

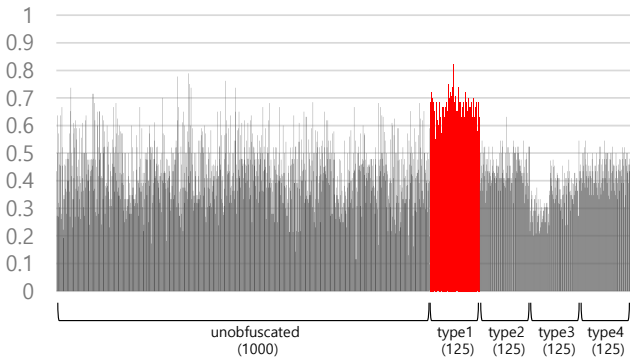

Figure 3. Structure similarity of data compared with obfuscation type 1 with a conventional AST.

At this point, we must set a threshold to classify the type 1 obfuscated JavaScript, which is a difficult task. If we set the threshold too high, we cannot detect all type 1 obfuscated JavaScripts. On the other hand, if we set it too low, we might misclassify non-type 1 obfuscated JavaScripts as type 1.

Figure 4 shows the classification results according to the threshold of obfuscation type 1 in the conventional AST. In general, it is difficult to determine an optimal threshold value because the accuracy and false negative rate (FNR) usually change together. For example, if we use a higher threshold for a higher accuracy, the false positive rate (FPR) decreases, but the FNR increases. If we use a lower threshold for a lower FNR, the accuracy decreases.

Figure 5 shows the structural similarities measured between the representative type 1 obfuscated JavaScript and all JavaScripts when the lightweight AST is used. For type 1, the structural similarity is measured as 1, because all obfuscation type 1 JavaScripts have the same lightweight AST by Theorem 1. This applies the same to type 2, 3, and 4, respectively. Note that their similarities are not 1 because they are not type 1.

Figure 6 shows the classification result according to the threshold of obfuscation type 1 in a lightweight AST. In this case, we can easily select an optimal threshold because the FNR does not increase, even if the accuracy is increased by the change in threshold. For example, if the threshold is 0.76, then the accuracy is 1, and the FPR and FNR are 0. Even if the threshold increases, the accuracy is already the maximum and the FNR does not increase.

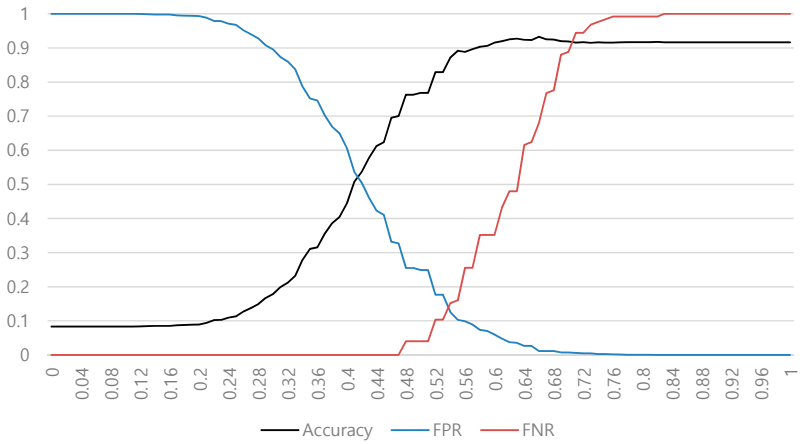

Figure 4. Classification result according to the obfuscation type 1 threshold in the conventional AST (FPR: False Positive Rate, FNR: False Negative Rate).

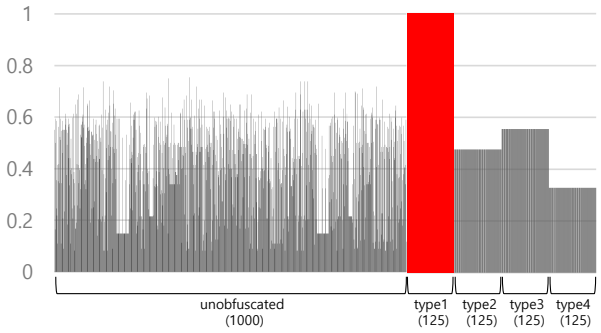

Figure 5. Structure similarity of data compared with obfuscation type 1 in a lightweight AST.

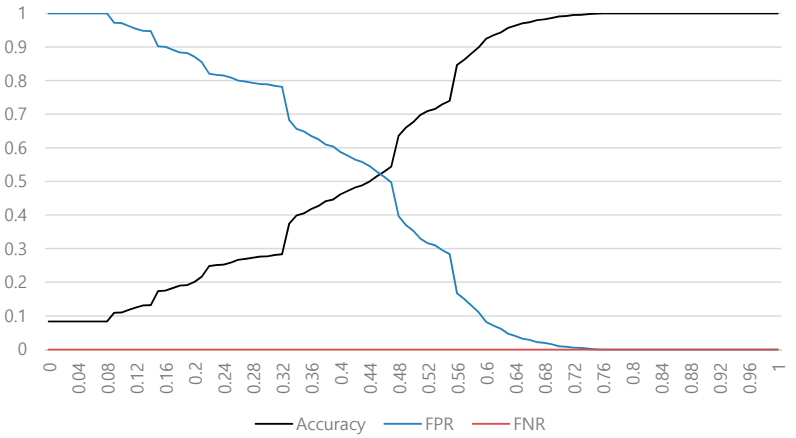

Figure 6. Classification result according to the obfuscation type 1 threshold in a lightweight AST.

Table 6 shows the results of the measurement classification accuracy, FPR, and FNR according to the threshold values of the conventional and lightweight ASTs, respectively. From these results, it is evident that the findings for obfuscation types 2 to 4 are similar to those for obfuscation type 1. When using the conventional AST, the results differ according to the priority of the accuracy, FPR, and FNR. Nevertheless, when using the lightweight AST, there are threshold values that meet all three criteria in all obfuscation types. The accuracy of classifying each obfuscation type of the lightweight AST is 100%. In conclusion, the lightweight AST is more efficient for classification of obfuscation types compared to the conventional AST.

Table 6. Classification accuracy, FPR, and FNR according to the threshold in conventional and lightweight ASTs.

Obfuscation Type	Conventional AST			Lightweight AST	
	Priority	Threshold	Result	Threshold	Result
Type 1	Maximum accuracy	0.66	Accuracy: 0.93 FPR: 0.01 FNR: 0.68	0.76–1	Accuracy: 1 FPR: 0 FNR: 0
	Minimum FPR	0.82–1	Accuracy: 0.91 FPR: 0 FNR: 0.99		
	Minimum FNR	0–0.47	Accuracy: 0.73 FPR: 0.29 FNR: 0		
Type 2	Maximum accuracy	0.66	Accuracy: 0.96 FPR: 0.01 FNR: 0.27	0.87–1	Accuracy: 1 FPR: 0 FNR: 0
	Minimum FPR	0.77–1	Accuracy: 0.93 FPR: 0 FNR: 0.78		
	Minimum FNR	0–0.55	Accuracy: 0.88 FPR: 0.12 FNR: 0		
Type 3	Maximum accuracy	0.5	Accuracy: 0.97 FPR: 0.002 FNR: 0.26	0.82–1	Accuracy: 1 FPR: 0 FNR: 0
	Minimum FPR	0.56–1	Accuracy: 0.96 FPR: 0 FNR: 0.37		
	Minimum FNR	0–0.22	Accuracy: 0.32 FPR: 0.73 FNR: 0		
Type 4	Maximum accuracy	0.75	Accuracy: 0.94 FPR: 0.01 FNR: 0.55	0.88–1	Accuracy: 1 FPR: 0 FNR: 0
	Minimum FPR	0.89–1	Accuracy: 0.91 FPR: 0 FNR: 0.96		
	Minimum FNR	0–0.55	Accuracy: 0.78 FPR: 0.23 FNR: 0		

Furthermore, newly inserted obfuscation cases will share a certain similarity value, which will not match 100% of any existing obfuscation case. When a system shows multiple JavaScripts with a certain similarity value, we can speculate that new obfuscation type could be used for attacks. This speculation can be verified after conducting in-depth analysis of the suspicious obfuscated JavaScript(s).

4.3. Malicious Signature Inspection

To perform malicious signature inspection, an experimental setup was built with a dataset having a 1:1 ratio of benign to malicious elements. We used the signatures (iframe, src) and (script, src), which are typically used in redirection code. Note that these signatures are generated for each obfuscation type. As shown in Table 7, obfuscated malicious JavaScripts are 100% correctly classified because they necessarily contain redirection code, such as (iframe, src) or (script, src). In other words, there is no misclassification of malicious code as being benign.

Table 7. Classification result from malicious signature inspection.

Dataset	No. of Data	No. Detected as Benign	No. Detected as Malicious
Obfuscated malicious JavaScript	100	0	100
Obfuscated benign JavaScript	100	47	53
Total	200	47	153

On the other hand, among the 100 obfuscated benign JavaScripts, 47 are classified as benign; however, 53 are actually malicious. The latter case occurs because they contain strings that are used in the redirection code. Malware signature inspection excludes a large number of obfuscated benign JavaScripts from the test.

4.4. Revised TF-IDF

Figure 7 shows the average TF-IDF values over redirection tokens in 100 obfuscated benign JavaScripts using the conventional TF-IDF. The TF-IDF average ranges from a minimum of 1.4 to a maximum of 16.99, with an average of 4.47. Figure 8 shows the average TF-IDF values over redirection tokens in 500 obfuscated malicious JavaScripts using the conventional TF-IDF. The TF-IDF average ranges from a minimum of 1.42 to a maximum of 3.45, with an average of 2.30. To classify JavaScripts containing redirection code, we use 3.43 as the threshold value. We classify JavaScripts with a TF-IDF average that is less than the threshold value of those containing redirection code. In this case, the accuracy of the classification is 92.5%.

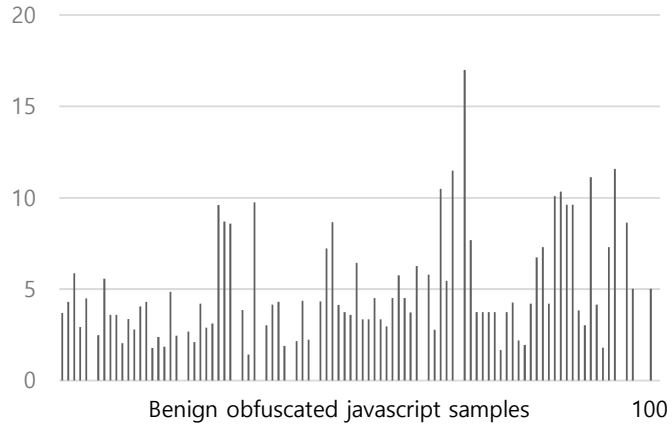

Figure 7. TF-IDF average values over redirection tokens in 100 obfuscated benign JavaScripts using the conventional TF-IDF.

Figure 9 shows the average TF-IDF values over redirection tokens in 100 obfuscated benign JavaScripts using the revised TF-IDF. The TF-IDF average ranges from a minimum of 0.0006 to a maximum of 0.024, with an average of 0.004. Figure 10 shows the average TF-IDF values over redirection tokens in 500 obfuscated malicious JavaScripts using the revised TF-IDF. The TF-IDF average ranges from a minimum of 0.008 to a maximum of 0.131, with an average of 0.049. Unlike the

conventional TF-IDF, the TF-IDF averages of the obfuscated redirection JavaScripts are higher than those of the obfuscated benign JavaScripts in the revised TF-IDF. To classify the JavaScripts containing redirection code, we use 0.01 as the threshold value. We classify the JavaScripts with an average TF-IDF greater than the threshold value as those containing redirection code. In this case, the accuracy of the classification is 96.8%.

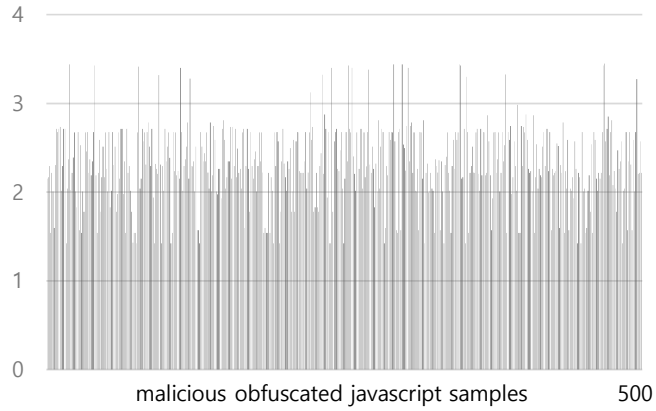

Figure 8. TF-IDF average values over redirection tokens in 500 obfuscated malicious JavaScripts using the conventional TF-IDF.

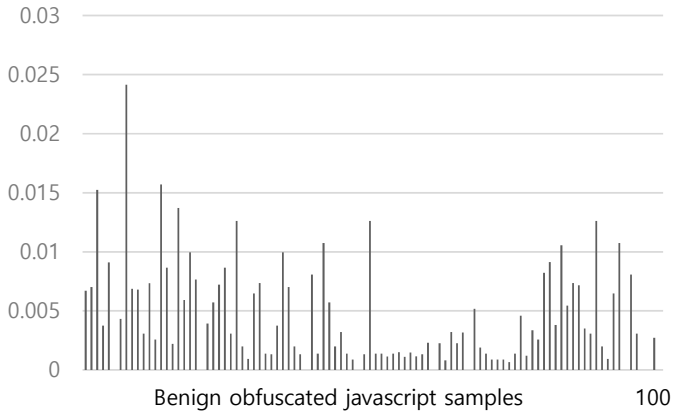

Figure 9. TF-IDF average values over redirection tokens in 100 obfuscated benign JavaScripts using the revised TF-IDF.

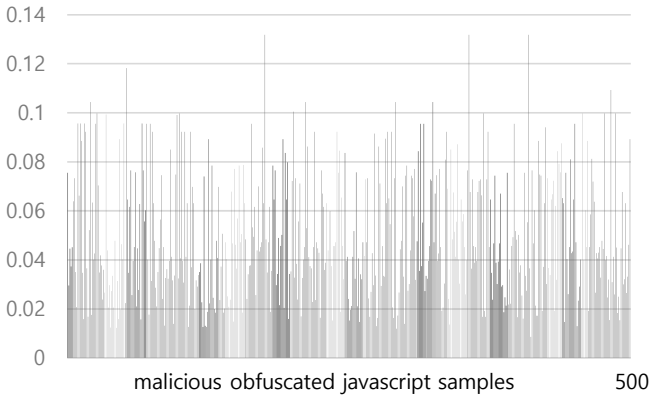

Figure 10. TF-IDF average values over redirection tokens in 500 obfuscated malicious JavaScripts using the revised TF-IDF.

Table 8 summarizes the above experimental results. Overall, the revised TF-IDF shows better performance than the conventional TF-IDF in terms of accuracy, FPR, and FNR. In particular, when the

FPR is 0, the accuracy and FNR are extremely poor in the conventional TF-IDF. In conclusion, the proposed revised TF-IDF classifies obfuscated JavaScript better than the conventional TF-IDF.

Table 8. Comparison performance of conventional TF-IDF and revised TF-IDF.

TF-IDF	Accuracy	FPR	FNR
Conventional TF-IDF	92.5%	0.074	0.076
	16.6%	0	0.833
	91.3%	0.074	0
Revised TF-IDF	96.8%	0.019	0.014
	85.5%	0	0.465
	95.8%	0.036	0

4.5. Comparison

Here we compare the proposed system with existing dynamic analysis systems, which are usually employed to analyze obfuscated JavaScript in the real world. Accordingly, we compare the proposed system with the JSUNPACK dynamic analysis system [24] by using 600 obfuscated JavaScripts. In Table 9, it is observed that the proposed system requires 0.02 s to process a JavaScript, whereas JSUNPACK requires 27.21 s. The proposed system is therefore four times faster than JSUNPACK. Moreover, the accuracy of the proposed system is 96.8%, which is comparable to JSUNPACK's accuracy.

Furthermore, we compare our system with a recent static analysis method [17]. Our system requires 0.020 s to process a JavaScript, whereas that in [17] requires 0.024 s. Hence, the proposed system is 20% more efficient than that in [17] in terms of analysis time. However, the accuracy of our system is lower than that of [17], which is enabled by long-time model training and intensive computation for deep learning. The overhead is heavy for maintaining such a system, as training must be occasionally performed to achieve its detection accuracy. Furthermore, when a small number of new obfuscation cases are injected into the system for malicious purposes, the model being trained in [17] is not reflected accordingly because it is based on deep learning, which relies on a sufficiently large dataset to achieve good detection accuracy.

In contrast, in the proposed system, no training is required; consequently, the maintenance overhead is very low. Even if there is a small number of new obfuscation cases injected into the system, our proposed solution can detect them quickly, unlike the system in [17]. The proposed system enables detection of malicious landing sites in a very short time (within 'golden time'), thereby consuming fewer computational resources with almost the same accuracy as the dynamic analysis system.

Table 9. Performance comparison.

	Analysis Time	Accuracy	Training	Required System Resources
Dynamic analysis (JSUNPACK [24])	27.21 s	100%	none	high
Static analysis ([17])	0.024 s	99.3%	required	high
Proposed	0.020 s	96.8%	none	low

4.6. Discussion

We claimed that our lightweight AST can identify obfuscation type with 100% in Theorem 1. This was proved by the current obfuscation types. Theorem 1, however, can also be applied to new types of obfuscation algorithm which may emerge in the future. Figures 4 and 5 show similarities of JavaScripts compared with type 1 obfuscated JavaScript. In contrast with Figure 4 showing experiment results using normal AST, Figure 5 shows some stable plateaus, giving hints that the JavaScripts obfuscated by other types may exist. In this way, our lightweight AST can help detect new obfuscation as well.

We provide the obfuscation distinguishment function by improving AST. Additionally, we provide malicious JavaScript detection function by revising TF-IDF in a lightweight and fast manner. Table 9 compares our approach with the other approaches including dynamic analysis and deep learning based analysis. When compared to dynamic analysis, our approach has a little bit lower, but relatively comparable accuracy. Particularly, our approach can detect certain JavaScript that dynamic analysis cannot detect due to inapposite triggers. Furthermore, our approach uses relatively few resources and is over 100 times faster than dynamic analysis. When compared to deep learning based analysis requiring powerful computing resource, our approach uses very few resources and is 20% faster with 3% lower accuracy. Because of these advantages, our approach is specifically suitable to anti-virus or web browser on smartphones, which cannot support powerful computing resource required by dynamic analysis and deep learning. Furthermore, our approach can support smartphone users' real-time, but safe, web surfing by preventing drive-by-download attacks.

Additionally, we briefly describe a means of achieving a complementary hybrid system by using our system first and then applying the other systems. In the real world, static and dynamic analyses are mostly used together when system resources are sufficient. When our system is combined with a dynamic analysis system, such as JSUNPACK, it first filters out benign JavaScript with 100% accuracy, which significantly reduces the workload in the dynamic analysis system. When combined with a machine learning-based static analysis system, such as that in [17], our system classifies the obfuscation types, which are then fed back into the static analysis system as a training dataset, leading to high accuracy.

5. Conclusions

In this paper, we proposed a static analysis system based on an AST to detect obfuscated JavaScript. In particular, a lightweight AST was proposed to classify the obfuscated JavaScript with higher accuracy and time efficiency compared to the conventional AST. In addition to the lightweight AST, a revised TF-IDF was proposed to detect the redirection code with higher accuracy than the conventional TF-IDF. Our experimental analysis showed that the proposed static analysis system has high accuracy that is comparable to that of a dynamic analysis system while requiring significantly less analysis time. It thus offers the best attribute of static analysis. As a result, the proposed system can be used to quickly detect obfuscated landing sites, thereby preventing users from drive-by-download attacks and their consequences in the real world. In the future, we will collect additional malicious JavaScripts, including various obfuscation features, and apply our analysis method to them in a machine-learning environment.

Author Contributions: Conceptualization, K.H.; methodology, K.H. and S.O.H.; validation, K.H. and S.O.H.; formal analysis, K.H. and S.O.H.; investigation, K.H. and S.O.H.; resources, S.O.H.; writing—original draft preparation, K.H.; writing—review and editing, K.H. and S.O.H.; supervision, S.O.H.; funding acquisition, S.O.H. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIT) under grant 2020R1A2B5B01002145. This work was also supported by the Gachon University Research Fund of 2019 (GCU-2019-0796).

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

References

1. Howard, F. Malware with your Mocha? Obfuscation and antiemulation tricks in malicious JavaScript. *Sophos Tech. Pap.* **2010**, *14*.
2. Kishore, K.R.; Mallesh, M.; Jyostna, G.; Eswari, P.L.; Sarma, S.S. Browser JS Guard: Detects and Defends against Malicious JavaScript Injection based Drive by Download Attack. In Proceedings of the Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT), Bengaluru, India, 17–19 February 2014; pp. 92–100.

3. Choi, S.Y.; Lim, C.G.; Kim, Y.M. Automated Link Tracing for Classification of Malicious Websites in Malware Distribution Networks. *J. Inf. Process. Syst.* **2019**, *15*, 100–115.
4. Wang, G.; Stokes, J.W.; Herley, C.; Felstead, D. Detecting malicious landing pages in Malware Distribution Networks. In Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, 24–27 June 2013; pp. 1–11.
5. Chellapilla, K.; Maykov, A. A Taxonomy of JavaScript Redirection Spam. In Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web, Banff, AB, Canada, 8 May 2007; pp. 81–88.
6. Yue, C.; Wang, H. A Measurement Study of Insecure JavaScript Practices on the Web. *ACM Trans. Web* **2013**, *7*, 1–39. [CrossRef]
7. Patil, D.R.; Patil, J.B. Detection of malicious javascript code in web pages. *Indian J. Sci. Technol.* **2017**, *10*, 1–12. [CrossRef]
8. He, X.; Xu, L.; Cha, C. Malicious JavaScript code detection based on hybrid analysis. In Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, 4–7 December 2018; pp. 365–374.
9. Curtsinger, C.; Livshits, B.; Zorn, B.G.; Seifert, C. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In Proceedings of the USENIX Security Symposium, San Francisco, CA, USA, 8–12 August 2011; pp. 33–48.
10. Kapravelos, A.; Shoshitaishvili, Y.; Cova, M.; Kruegel, C.; Vigna, G. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In Proceedings of the 22nd {USENIX} Security Symposium ({USENIX} Security 13), Washington, DC, USA, 14–16 August 2013; pp. 637–652.
11. Wang, Y.; Cai, W.D.; Wei, P.C. A deep learning approach for detecting malicious JavaScript code. *Secur. Commun. Netw.* **2016**, *9*, 1520–1534. [CrossRef]
12. Wei, J.; Huiqiang, W.; Keke, W. Method for Detecting JavaScript Code Obfuscation based on Convolutional Neural Network. *Int. J. Perform. Eng.* **2019**, *14*, 3167–3173.
13. Fang, Y.; Huang, C.; Liu, L.; Xue, M. Research on Malicious JavaScript Detection Technology Based on LSTM. *IEEE Access* **2018**, *6*, 59118–59125. [CrossRef]
14. AL-Taharwa, I.A.; Lee, H.M.; Jeng, A.B.; Wu, K.P.; Ho, C.S.; Chen, S.M. JSOD: JavaScript obfuscation detector. *Secur. Commun. Netw.* **2014**, *8*, 1092–1107. [CrossRef]
15. Blanc, G.; Miyamoto, D.; Akiyama, M.; Kadobayashi, Y. Characterizing Obfuscated JavaScript using Abstract Syntax Trees: Experimenting with Malicious Scripts. In Proceedings of the 2012 26th International Conference on Advanced Information Networking and Applications Workshops, Fukuoka, Japan, 26–29 March 2012; pp. 344–351.
16. Al-Taharwa, I.A.; Lee, H.M.; Jeng, A.B.; Ho, C.S.; Wu, K.P.; Chen, S.M. Drive-by Disclosure: A Large-Scale Detector of Drive-by Downloads Based on Latent Behavior Prediction. *IEEE Trust. BigDataSE ISPA* **2015**, *1*, 334–343.
17. Fang, Y.; Huang, C.; Su, Y.; Qiu, Y. Detecting malicious JavaScript code based on semantic analysis. *Comput. Secur.* **2020**, *93*, 101764. [CrossRef]
18. Provos, N.; Mavrommatis, P.; Rajab, M.; Monrose, F. All Your iFRAMEs Point to Us. In Proceedings of the USENIX Security Symposium, San Jose, CA, USA, 28 July–1 August 2008; pp. 1–16.
19. Cova, M.; Kruegel, C.; Vigna, G. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In Proceedings of the 19th International Conference on World Wide Web, Raleigh, NC, USA, 26–30 April 2010; pp. 281–290.
20. Manku, G.S.; Jain, A.; Das Sarma, A. Detecting near-duplicates for web crawling. In Proceedings of the 16th International Conference on World Wide Web, Banff, AB, Canada, 8–12 May 2007; pp. 141–150.
21. ECMAScript Language Specification—ECMA-262 Edition 5.1. Available online: https://www.ecma-international.org/ecma-262/5.1/ (accessed on 29 January 2020).
22. Overview|Safe Browsing APIs (v4)|Google Developers. Available online: https://developers.google.com/safe-browsing/v4/ (accessed on 11 February 2018).

23. Lightweight Detection Method of Obfuscated Landing Sites. Available online: https://github.com/hkh112/Lightweight-detection-method-of-obfuscated-landing-sites (accessed on 10 August 2020).
24. JSUNPACK. Available online: https://github.com/urule99/jsunpack-n (accessed on 10 December 2018).