JSHint: Revealing API Usage to Improve Detection of Malicious JavaScript

Shaown Sarker$^{1[0009-0000-6700-5824]}$, Kasimir Schulz$^{1[0009-0007-6039-5883]}$, Aleksandr Nahapetyan$^{1[0009-0001-3759-993X]}$, Anupam Das$^{1[0000-0002-8961-9963]}$, and Alexandros Kapravelos$^{1[0000-0002-8839-8521]}$

North Carolina State University, Raleigh NC 27695, USA
`{ssarker,krschulz,anahape,anupam.das,akaprav}@ncsu.edu`

Abstract. In the modern web, JavaScript (JS) provides dynamic behavior and, at the same time, is often used by malicious actors for a wide range of attacks such as drive-by-downloads, malicious payload delivery, ransomware, cryptojacking, and phishing. Malicious actors leverage JS obfuscation to obscure the original intention of the source code by hiding the API usages and evading existing detection systems. In this paper, we present a subtree evaluation-based system named JSHint, that focuses on the recovery of API usage in obfuscated JS scripts. We demonstrate that introducing obfuscation using off-the-shelf tools to JS source code significantly hides the standard API usage, and after being restored by JSHint almost complete recovery of API usages is achieved, with averages as high as 95.06%. We further demonstrate that introducing obfuscation to malicious JS source code makes them evasive - by generating false negatives as high as 9.30%. Through API usage restoration by JSHint, we can defeat these evasions significantly - up to 94.97% of the false negatives are converted back into true positives.

1 Introduction

JavaScript (JS) is the most popular scripting language in the current web ecosystem and provides dynamic behaviors to complement and augment the static nature of HTML & CSS. JavaScript obfuscation is the process of making a JS source code unintelligible while keeping the functionality of the source intact. Because of JavaScript's widespread popularity and dynamic nature, it is also the choice of malicious actors for carrying out attacks on the web at scale. To increase the efficacy of their attacks, malicious actors further combine JS obfuscation to their JS-based malware to evade detection systems. A recent malicious JS injection [28] attack, which employed obfuscation, was discovered on 51k websites, many of which belong to the Tranco [1] top domain list, where the attacker redirects victims to malicious content such as adware and scam pages by inserting obfuscated malicious JS in multiple stages.

[1] `https://tranco-list.eu/`

Due to its capabilities for obscuring the original intention of the JS source code, malicious actors have long used JS obfuscation for various malicious web activities [19,14,21]. Researchers have treated JS obfuscation as a footprint of malicious activity and attempted to detect JS-based malware with JS obfuscation instead of removing it [21,6,18,35,12,2,23,20,7,13]. Previous research has shown that, even with the negative connotation of obfuscation, it is widely used across the web - almost 96% of the top Alexa [2] 100k domains contain at least one JS script with some form of obfuscation [33,34]. The hiding of the original intention of the source is achieved through transforming the logic of the original source and/or hiding the standard API usage, which makes it harder to understand the effect of the source on the underlying system.

Despite the fact that JS code obfuscation is a common practice on the modern web [33,34], and many tools [15,36,30,3,9,17] are available for applying obfuscation to JS code, there is a scarcity of effective JS systems that can remove obfuscation, specifically from the API usages. `JSDES` [1] attempted to deobfuscate automatically by first identifying a set of obfuscated functions in the JS source code manually and then tracing the functions capable of generating code dynamically within the previously collected obfuscated functions set and simulating their execution to have deobfuscated code. This approach is limited by the manual collection of obfuscated functions set and can only be applied to obfuscation present at the function level. Lu et al. proposed a semantic approach for automated simplification of JS obfuscated code that relies on dynamic analysis followed by program slicing [25]. However, due to the reliance on expensive dynamic analysis for obfuscation removal, their system is not able to perform obfuscation removal statically. Similarly, there were attempts made to create a rule-based deobfuscator to rewrite obfuscated JS source code, such as `Maude` [5], but such systems are also not feasible due to the limitations of the said obfuscation rules and not being able to handle more complex obfuscation techniques readily available today.

To bridge this gap between the availability of readily accessible obfuscation tools that obscure standard API usages and the scarcity of usable deobfuscation systems that can uncover them properly, we present JSHint, a deobfuscation system specifically focuses on API usage recovery. JSHint is built using existing tools, does not rely on complex dynamic analysis, or requires previously curated data. The main principle of JSHint is to find the most minimal subtree in the JS source code's abstract syntax tree (AST), which we can evaluate successfully to a primitive type to remove the obfuscation and subsequently propagate that evaluation throughout the rest of the tree. We iterate this evaluation process over multiple passes to achieve the maximum obfuscation removal and API usage recovery.

We achieve this by extending the current AST-based scope management system to include precise scope tracking, taint analysis, and object modification tracking. This enables us to perform partial safe evaluations of the AST subtrees, including complex expressions such as function calls, runtime array access

[2] `https://aws.amazon.com/alexa-top-sites/`

expressions, and object member expressions, and propagate the results of a successful evaluation throughout successive evaluations. We maintain a dynamic scope that we update throughout the evaluation pass over the subtrees to reflect the current state of the source code at each node and a persistent scope in between evaluation passes. This meticulous scope tracking enables us to perform safe evaluations without resulting in an inconsistent state during our deobfuscation process. Thus, we restore the original intention of the script by removing obfuscation from standard API usage.

To evaluate JSHint, we introduce obfuscation to a curated set of JS samples using a readily available JS obfuscation tool with three gradually complex obfuscation profiles and then processing the obfuscated code through JSHint. Using a standard API usage recovery metric, we demonstrate that application of obfuscation from this off-the-shelf obfuscation tool results in only 34.77%, 31.04%, and 31.34% of original standard API usages being recoverable from the obfuscated code for the three profiles, whereas JSHint can restore the standard API usages almost perfectly with averages of 95.06%, 93.92%, and 94.38% for the same three obfuscation profiles. Additionally, we compare JSHint against two existing deobfuscation systems and demonstrate that JSHint performs better when it comes to removing obfuscation from API usages.

Furthermore, we display the evasive power of JS obfuscation by introducing obfuscation to a collected set of malicious JS samples and classifying them through a state-of-the-art malicious JS classifier trained on the original versions of the malicious scripts. Our results show that the obfuscated scripts can achieve evasion rates as high as 9.30%. After performing API usage recovery using JSHint, we can classify up to 94.97% of these evasions correctly, again defeating the evasion introduced through obfuscation. We list the major contributions of this paper as follows:

– We present JSHint, an efficient system for recovering API usage by updating the Abstract Syntax Tree (AST) through subtree evaluation to primitive literal types. We plan to eventually make JSHint available to the research community (either by open-sourcing it, or by making it available as a service).
– We demonstrate the power of obfuscation by applying obfuscation on a set of JS samples, which significantly obscures the standard API usage. JSHint restored almost all standard API usage in the original script. We also compare JSHint against existing off-the-shelf deobfuscation systems and display that our system outperforms the competitors in terms of restoring API usage in the scripts.
– We further display that introducing obfuscation to malicious JS can make them evasive to a state-of-the-art AST-based malicious JS classifier trained on the original versions of the JS samples, resulting in evasions through false negative rates as high as 9.30%. Finally, JSHint is capable of defeating up to 94.97% of these evasions after restoring the API usages in the obfuscated scripts.

2 Background

Besides protecting code, JS obfuscation is used to evade detection [6,7,13,18], and researchers consider JS script obfuscation as an indicator of maliciousness [2,12,20,23,35]. With the readily available off-the-shelf JS obfuscation tools, it is even easier for malicious actors to use JS obfuscation for evasive maneuvers. In this section, we present a brief overview of JS obfuscation techniques and the available off-the-shelf obfuscation tools.

2.1 Obfuscation Techniques

Based on prior research into JS obfuscation techniques and available JS obfuscation tools [27], we can categorize JS obfuscation techniques into four major categories.
Identifier & whitespace Randomization. Randomization can be applied to identifiers (renaming variables and function names randomly) as well as white-space in source code (random white-space insertions/deletions). This preserves the semantics and logical flow of the source code and makes it harder to debug and comprehend the intention.
String manipulation. Obscuring the readable strings can be greatly evasive. The techniques used for this include splitting a string into multiple sub-strings, concocting a string from char codes, and encoding a string to non-readable forms.
Structural transformation techniques. This includes two major types: string array manipulation and control-flow flattening. The string array manipulation involves invoking all function calls and member expressions in the script source indirectly through a lookup table and/or an accessor function [33]. In control flow flattening, it takes all the basic blocks in the code, which include function declarations, iterative expressions, and conditional branches, and pushes these blocks inside a global single infinite iterative block with a switch statement which determines the execution flow of the program. Unlike randomization, structural transformation often modifies the entire source structure significantly in an irreversible manner.
Code protection techniques. Code protection includes injection code that disables the debugger and the console from the runtime environment and injects dead code never executed during runtime. Although code protection techniques introduce new source code, unlike structural transformation techniques, these do not necessarily modify the code in a unidirectional manner.

2.2 Obfuscation Tools

We conducted a small survey of readily available obfuscation tools and their obfuscation techniques in Table 1. We considered five off-the-shelf JS obfuscation tools: obfuscator.io [15], javascript-obfuscator [17], gnirts [3], jfogs [36], jsobfu [30] in our survey. Among these, `obfuscator.io` [15], `gnirts` [3], `jfogs` [36] are available as npm packages, `jsobfu` [30] is ruby-based and available as a gem

Table 1. Survey of off-the-shelf obfuscation tools available online with their capabilities

Obfuscation Techniques	obfuscator.io	javascript-obfuscator	gnirts	jfogs	jsobfu
Split strings	✓	✗	✓	✗	✗
Concatenate strings	✗	✗	✓	✗	✓
Encode strings	✓	✓	✓	✗	✓
String array	✓	✗	✗	✓	✗
Identifier renaming	✓	✓	✗	✓	✓
Control flow flattening	✓	✗	✗	✗	✗
Anti-debugger code injection	✓	✗	✗	✗	✗
Dead code injection	✓	✗	✗	✗	✗
Disable console code injection	✓	✗	✗	✗	✗

package, and `javascript-obfuscator` [17] is available as a MS Windows application and a web portal. We excluded obsolete tools like daft-logic [9] from the survey [3]. Given the popularity [4] and feature richness of `obfuscator.io`, we used this tool for our purposes in this paper.

2.3 Limitations of Deobfuscation Tools

Although JS obfuscation tools are easily available, JS deobfuscation tools are however considerably rare. In `JSDES` [1] manually finds suspect obfuscation functions and then deobfuscates by simulating their execution behavior. This requires manual effort and is limited to function-level obfuscation only. Lu et al. [25] used dynamic analysis and program slicing techniques to simplify JS source code and remove obfuscation. However, such approaches require expensive dynamic analysis (the authors used Mozilla's SpiderMonkey interpreter) and suffer from very low code coverage. Rule-based JS transformers such as `Maude` [5] are limited by the set of rules they depend on and are not capable of handling the more complex obfuscation techniques mentioned above.

3 Deobfuscating API Usages

In this section, we present JSHint, a subtree evaluation-based system that recovers JS API usages without depending on any predefined set of rules or function evaluations. Our system's goal is to recover obfuscated primitive literal type information (string, number, boolean, or null) and thus also restore standard API usages.

[3] The daft-logic obfuscator was last updated in 2009
[4] The obfuscator.io npm package weekly downloads exceed 100k

3.1 Design Principle

The goal of JSHint is to find the most minimal subtree in the Abstract Syntax Tree (AST) of the JS source code that we can evaluate successfully to a primitive type. We begin by extracting the AST from the JS source code and then extracting the scope information from the AST. We begin our multiple passes over the AST with a subtree evaluation pass starting at the root. We traverse the AST in a top-down manner. At each node, we attempt to evaluate the subtree rooted at the node using a concocted evaluation environment. We replace the original subtree with the evaluation result upon a successful evaluation. After all evaluation passes are complete or there are no more updates to be done, we generate the deobfuscated version of the original from the updated AST.

3.2 Code Evaluation

We used `MetaES` [26] for our lightweight JS code evaluation purposes. `MetaES` is a meta-circular interpreter since it is in the same language it interprets, which supports ECMAScript5.1+ and both JS code snippets and AST. See Listing 1.1 and 1.2, for examples of evaluation of a simple JS snippet in both source and AST form.

```
let res;
let environment = { values: { a: 2 } };
metaesEval("a*2",
    val => {res = val},
    console.error,
    environment);
```

Listing 1.1. MetaES evaluation of snippet, `res` contains 4

```
let res;
let environment = { values: { a: 2 } };
metaesEval(
    {
        type: "BinaryExpression",
        operator: "*",
        left: {
            type: "Identifier",
            name: a,
            raw: "2"
        },
        right: {
            type: "Literal",
            value: 2,
            raw: "2"
        }
    },
    val => {res = val},
    console.error,
    environment);
```

Listing 1.2. MetaES evaluation of AST, `res` contains 4

3.3 Extending `escope`

Code evaluation requires a scope containing the required variable and function values. We selected the widely used scope analyzer tool `escope` [11] part of the estools toolchain[5] for this purpose. However, `escope` has three major limitations. First, `escope` lacks of any tracing for passed-in parameter values or arguments in a function call expression. Second, `escope` has very little taint tracking capabilities, specifically in cases for variable reassignment and declaration using another variable. Finally, `escope` does not track modifications to an object in the current scope through function calls or member expression assignments. We extended `escope` with the following additional features for these three limitations and named it `kscope`, which is fully compatible with existing `escope` code.

Precise scope tracking. `escope` lacks any tracing for passed-in parameter values or arguments in a function call expression. In `kscope`we solve this by extracting and categorizing all possible scopes into four types in order of their hierarchy - global, function, iteration, and block. The global scope nests all other scopes and all nested scopes inherit from the global scope. A function scope begins with a function declaration (named functions) or a function expression (anonymous functions). Iteration scopes are initialized with an iterative node (`for`, `for...in`, `for...of`, `while`, and `do...while` statements), and includes the variables initialized and incremented (if any) in the iteration statements. Finally, block scope is initiated when we encounter other statement blocks that maintain their own lexical scope through variable declarators - `let` and `const`. All scopes are aware of the parent scope and inherit from it. However, we had to take extra caution for a variable declared through `var`. Since such variables are part of the global scope despite being declared inside a non-global scope, we propagate such variables through the parent scopes till we reach the global scope in `kscope`.

Taint analysis. `escope` has very little variable taint tracking capabilities. Since we require the most updated current values within our scope, `kscope` includes taint analysis for both variables and function names. `kscope` simply achieves this by marking variables and function names as tainted upon being declared again or being assigned to a new value within the scope of the original. `kscope` further propagates the taint flag when a variable or function name is assigned a value and then reassigned to another variable or function name by marking the latter variable or function name as tainted.

Object modification tracking. `escope`does not include any ability to track modifications to a scoped object modified through function calls or member expression assignments. `kscope` tracks whether an object has potentially been

[5] `https://github.com/estools`

changed through a function call (a reference of the object passed to a function and then modified within the function body) and/or a member expression invocation or assignment (e.g., calling `push` on an array object).

3.4 Implementating JSHint

With `MetaES` and `kscope`, we start the subtree evaluation process as detailed in 3.1. Given an (obfuscated) script, we extract the AST of the source code using `esprima`, and gather scope information from the AST using `kscope`. We then make a predetermined number of evaluation passes beginning with the AST root. We traverse the AST using `estraverse` during each pass. We perform several scope management tasks during this traversal and attempt subtree evaluations whenever applicable. After each successful evaluation, we extract the evaluation result's AST and replace the evaluated subtree's original node with this AST. After all the passes are completed or no more updates are possible, we generate the resultant code from the modified AST using `escodegen`.

3.5 Deobfuscation Case Study

Listing 1.3 displays a piece of malicious code snippet that initiates an XHR request for a malware payload, which upon delivery executes on the page using the notorious `eval` function.

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '//onongo.info/?XXwIxqOa=AVZBQQd'+
// Removed for brevity
'Ak8AG1wKGjNnTUk=');
xhr.withCredentials = true;
xhr.onload = function() {
    var ref = document.referrer;
    eval(xhr.responseText);
};
xhr.send();
```

Listing 1.3. Malicious JS script executing payload

Using the obfuscator.io tool, we introduced identifier randomizing, unicode escape sequences, and structural manipulation. The obfuscated source is shown in listing 1.4. This obfuscates all standard API usages on the `XMLHttpRequest` object including the malicious payload URL.

```
function _0x3ee3() {
    var _0x3eb75e = [
    'open', 'nfo/?XXwIx', 'ddFQBSFkOG', 'ElcEGwVQRQ' /* Removed for
    ↪ brevity */];
    _0x3ee3 = function() {
        return _0x3eb75e;
    };
    return _0x3ee3();
}
var _0x4a01dc = _0x5a7b,
    xhr = new XMLHttpRequest();

function _0x5a7b(_0x5a0000, _0xae3f87) {
    var _0x8213bb = _0x3ee3();
    return _0x5a7b = function(_0x29901f, _0x559179) {
        // Removed for brevity
    }, _0x5a7b(_0x5a0000, _0xae3f87);
}
xhr[_0x4a01dc(0x0)]('GET',
'//onongo.i' + _0x4a01dc(0x1) + 'qOa=AVZBQQ' /* Removed for brevity*/] =
    ↪ !![],
xhr[_0x4a01dc(0xa)] = function() {
    var _0x574bfa = _0x4a01dc,
        _0xae3f87 = document[_0x574bfa(0xb)];
    eval(xhr[_0x574bfa(0xc) + 'xt']);
}, xhr['send']();
```

Listing 1.4. Previous sample obfuscated with identifier randomization, unicode escape sequence, and string array techniques by the obfuscator.io tool

We applied our system to the obfuscated script with five evaluation passes. The resultant script in listing 1.5 has all standard API usages restored on the `xhr` object and the malicious payload URL.

```
 // Redacting redundant dead structures for brevity
xhr['open']('GET', '//onongo.info/?'+ /* Removed for brevity
    ↪ */'FpJAk8AG1wKGjNnTUk='),
xhr['withCredentials'] = !![],
xhr['onload'] = function () {
    var _0x574bfa = _0x4a01dc,
    _0xae3f87 = document['referrer'];
    eval(xhr['responseText']);
}, xhr['send']();
```

Listing 1.5. After removing obfuscation through the subtree deobfuscator

4 Evaluation

We evaluate JSHint from two perspectives: correctness of the resultant script and restoration of API usages. Since we focus on evasive JS malware, we constructed a set of malicious JS samples and introduced obfuscation using an off-the-shelf tool. Then we applied JSHint on the obfuscated scripts and measured both the syntax correctness and the API restoration capabilities.

We compiled a set of total 3,755 malicious JS samples consisting of 3,563 samples from VirusTotal [6] and 192 samples from the GitHub repository from HynekPe-

[6] `https://www.virustotal.com/`

Table 2. Obfuscation profiles and their corresponding obfuscation techniques

Obfuscation technique	Profile-1	Profile-2	Profile-3
Renaming variables (hexadecimal)	✓	✓	✓
Unicode escape sequence	✓	✓	✓
Split strings to arrays	✗	✓	✓
Rotate string arrays	✗	✓	✓
Control flow flattening	✗	✗	✓

trak [16]. We filtered out obsolete (before 2019) scripts, duplicates, and scripts that did not follow ECMAScript 5 standards[10].

We decided to use the popular `obfuscator.io` [15] tool from 2 to obfuscate our set of JS samples. We used three obfuscation profiles, each increasing the obfuscation than the previous, as shown in Table 2. We applied these obfuscation profiles on our JS scripts set, followed by applying our system on the resultant obfuscated scripts with five passes and a timeout of eight minutes. We retrieved 3,121, 2,745, and 2,669 scripts with a success rate of 83.12%, 73.11%, and 71.09% for the three profiles respectively. 47, 58, and 47 scripts caused our system to throw error for the profiles in order, the rest of the failures were due to timeouts.

4.1 Correctness of Output Scripts

A major concern of any code transformation is to produce code with correct syntax. Since JSHint applies on a single script basis, we used `esvalidate` - an ECMAScript standard validation tool [29] to check for syntax errors in our set of output scripts for each of the three profiles.

After applying `esvalidate`, we found that 2 (0.0007%), 5 (0.0019%), and 9 (0.0035%) output scripts had syntax breakages for profiles in order. The 2 cases is due to the evaluation of functions on the prototype of JS native objects in MetaES rather than by our system. As the obfuscation profiles are incremental, these syntax errors also cascade to the outputs of the other profiles. The other syntax breakage cases on profiles two and three stem from an identifier node replacement in `for-in` expressions which resulted from `kscope` due to stack limit reached and can be fixed in future iterations of `kscope`. Given our complex and incremental obfuscation profiles that perform a unidirectional transformation, JSHint produces output with extremely rare cases of syntax breakages (<0.001%).

4.2 API Usage Recovery

Sarker et al. demonstrated that JS obfuscation could be identified in any JS script through the presence of statically hidden JavaScript standard API usages [33]. We extrapolate on this and propose a standard API usage recovery metric based on the overlap of standard JavaScript API calls and property accesses between the original script, the obfuscated versions, and the output of JSHint versions.

We extracted 5,465 standard API calls and property names in the topmost object and member identifier format, e.g., `Document.createElement`, similar

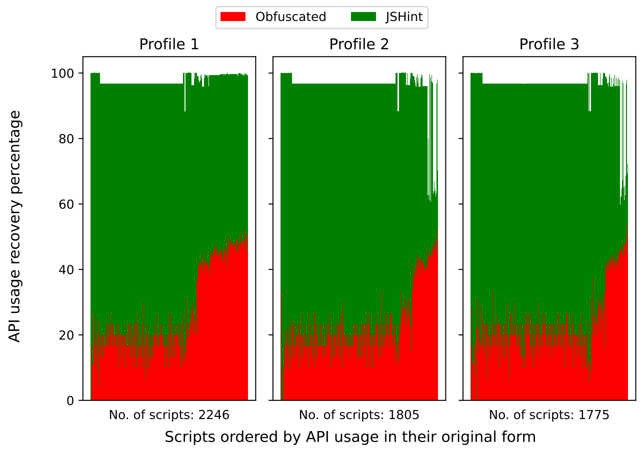

Fig. 1. Comparison of standard API recovery rates between original and obfuscated code vs between original and obfuscated code processed by JSHint versions for the obfuscation profiles (higher is better).

to [33], from Chromium major version 112. To determine all statically visible API usage from the script, we extracted the AST of the script and traversed it to find matches against all function calls (such as `Document.createElement` and member expressions (such as `Document["createElement"]`.

We extracted the set of statically visible API usages from the original script, as well as the corresponding obfuscated versions and our system's output versions, for all three obfuscation profiles. To evaluate the API usage recovery, we computed the API usage recovery percentage metric between each pair of JavaScript scripts as follows:

$$\text{API recovery} = \frac{|O \cap T|}{|O|} \times 100\%$$

where O and T are the set of API usages in the original and transformed version respectively.

In figure 1, we display this API recovery percentage metric for each of the profiles for between the original and the obfuscated versions, and between the original and the obfuscated code processed by JSHint versions of the scripts, excluding the cases with no API usage in the original versions. Each point on the x-axis represents a single script ordered by their standard API usage count in their original version. From the figure, we can see that our obfuscation profiles significantly obscure the API usages (50% to 80%) in the obfuscated script, and JSHint does restore almost completely all observed standard API usage in the original. This is further evident from figure 2, where we similarly display the distribution of the API recovery metric for each of the three profiles. While the mean API recovery rate between the original and the obfuscated versions are 34.77%, 31.04%, and 31.34%, obfuscated code processed by JSHint versions

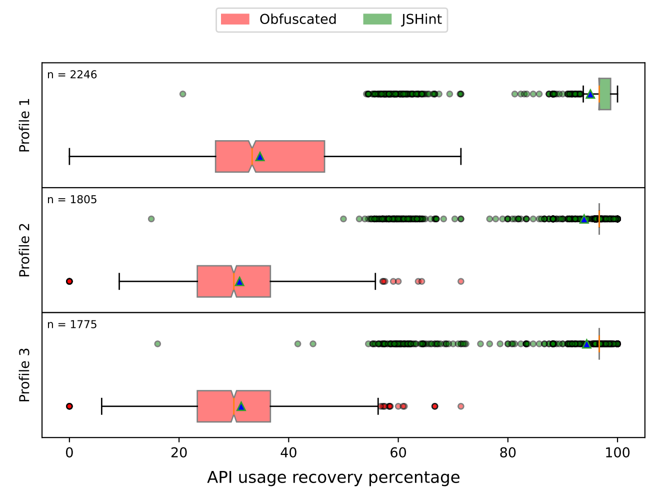

Fig. 2. Distribution of standard API recovery percentage between original and obfuscated code vs between original and obfuscated code processed by JSHint versions for the obfuscation profiles (blue triangle indicates mean).

result in a mean API recovery rate of 95.06%, 93.92%, and 94.38% for the three profiles in order.

4.3 Comparison against Deobfuscation Systems

In this section, we compare the efficacy of restoring API usage of our system against two other readily available deobfuscation systems in terms of API usage recovery.

JSNice. JSNice [31] is a JS deobfuscation tool based on the Nice2Predict framework [4]. JSNice uses predictive learning from large open-source code bases to recover identifier names and annotation types in obfuscated code. We used the updated version of JSNice that supports ECMAScript 6, packer detection and JavaScript code prettifier.

Synchrony. We selected Synchrony [32] particularly because it is a bespoke rule-based JS deobfuscation system targeted at the specific tool we used to introduce obfuscation [15]. We used the npm package version of Synchrony for our comparison [7].

From our run with Synchrony, we received 1,129 (30.07%), 1,144 (30.46%), and 1,128 (30.04%) output scripts for the three profiles in order. Synchrony resulted in success for only less than one-third of the scripts, which is significantly less than the success rate of JSHint. On the other hand, JSNice does not throw any errors due to its reliance on a predictive learning framework and static transformation process.

We applied our obfuscation profiles on 3,480 benign samples and measured the delta time it took the different systems to process successfully. Jshint time

[7] `https://www.npmjs.com/package/deobfuscator`

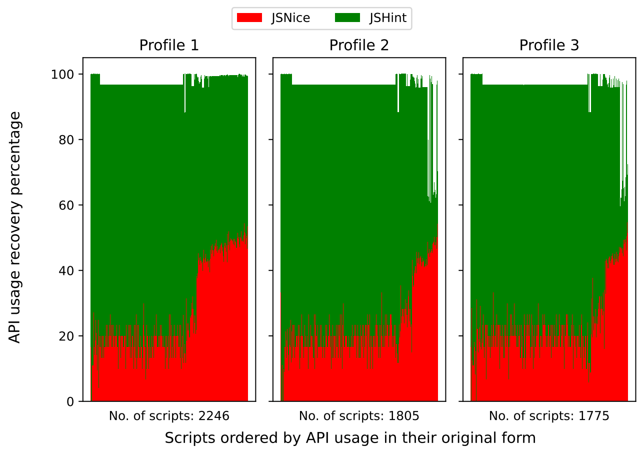

Fig. 3. Standard API recovery percentage between original and deobfuscated scripts by JSNice vs between original and obfuscated code processed by JSHint versions for the obfuscation profiles (higher is better).

averages were 3.1 ± 0.6, 3.6 ± 0.4, 4.2 ± 0.4 seconds for the three profiles, respectively. Synchrony averages were much higher with 35.6 ± 54.2, 62.5 ± 53.4, 44.8 ± 53.9 seconds.

We display the API recovery percentage metric for both between the original and the JSNice deobfuscated scripts, and between the original and the output of JSHint on obfuscated code for the three obfuscation profiles in Figure 3, and corresponding distribution in Figure 4. It is evident from the figure that JSNice performed very little obfuscation removal, and barely restored any API usage from the obfuscated versions of the script.

Figure 5 shows the API recovery percentage metric for both between the original and the Synchrony deobfuscated scripts and between the original and the output of JSHint on obfuscated code for the three obfuscation profiles, with the distributions displayed in Figure 6. From the figure, it can be seen again our system restores almost completely all observed standard API usage in the original, whereas although Synchrony manages to restore the standard API usage significantly better than JSNice, the restoration is visibly lower compared to JSHint. The Synchrony deobfuscated scripts results in a mean standard API recovery rate of 78.24%, 76.28%, and 78.63% for three profiles, where the recovery rate for JSHint for the three profiles are much higher at 96.51%, 93.31%, and 96.15%.

webcrack. We also evaluated JSHint against webcrack [8], the successor of Synchrony. Webcrack took a mean time of $3.0 \pm 32.9/4.3 \pm 32.6/3.4 \pm 33.3$ seconds respectively to process our input sample set for the three profiles respectively. Both JSHint and webcrack outperform Synchrony, while JSHint had better success rates. For the API recovery metric, webcrack had a mean recovery rate of 92.18%, 89.88%, and 96.22% respectively. Although, webcrack outperforms Syn-

[8] https://github.com/j4k0xb/webcrack

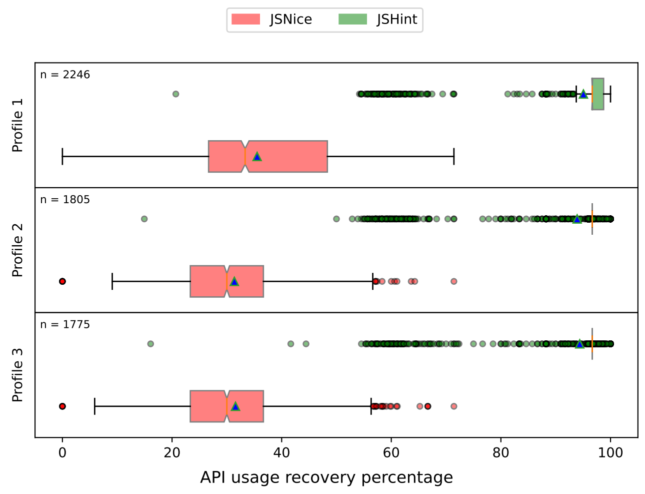

Fig. 4. Distribution of standard API recovery percentage between original and deobfuscated scripts by JSNice vs between original and obfuscated code processed by JSHint versions for the obfuscation profiles (blue triangle indicates mean).

chrony, and performs very close to JSHint, it is a bespoke system that specifically targets the obfuscation tool used in this paper, while JSHint performance is agnostic of the underlying obfuscation tool.

5 Obfuscation as Evasion

The primary motivation for using obfuscation in malicious JS scripts is to evade detection systems [23,20,2,18,12]. A successful evasion for a malicious JS script results in a false negative (FN) from a malicious JS detection system. A transformation defeats an evasion when the detection system can detect the script as a true positive (TP) from a previous FN. In this section, we study the effect of obfuscation on improving evasion and the effect of restoring API usages in the obfuscated malicious JS scripts on improving detection.

For this purpose, we used the collected malicious JS scripts from 4 and introduced obfuscation using the obfuscation profiles from Table 2, and passed the obfuscated scripts through a detection system. We measure the false negatives as evasions. Subsequently, we apply JSHint to the obfuscated scripts to restore the standard API usages and classify the scripts again using our detection system. We measure if any previously generated false negatives become true positives and categorize them as defeating evasion.

5.1 Detection System

For our study, we used JaSt, which is a state-of-the-art fully syntactic malicious JS script detector [12], as our detection system. JaSt uses a random forest clas-

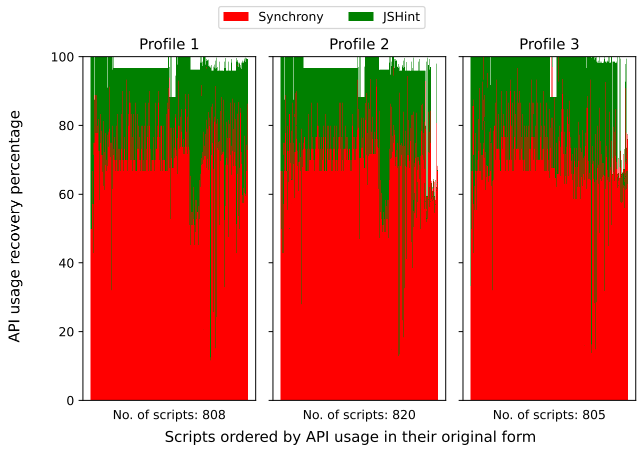

Fig. 5. Standard API recovery percentage between original and deobfuscated scripts by Synchrony compared between original and obfuscated code processed by JSHint versions for the obfuscation profiles (higher is better).

sifier on the pattern features extracted from the AST with high accuracy and a very low false negative rate [9]

To train JaSt, we required both a malicious and a benign JS script set. However, to prevent JaSt from learning obfuscation as maliciousness, we needed to filter out already obfuscated JS scripts from our malicious scripts. To detect obfuscation in our malicious set, we used previous research that defines JS obfuscation in terms of API usage where obfuscation can be determined in JS scripts through finding statically obscure JS API usages [33].

To filter our malicious JS script set, we applied JSHint on the 3,755 malicious JS scripts. This resulted in 3,407 scripts in the output set that was successfully processed through our system. We then extracted and compared the set of standard API usages in the similar manner from section 4.2, from both the original malicious scripts and their corresponding output from our system. If we detected any standard API usages discovered in the output version of a malicious script that was not present in the original version, we marked the script as obfuscated and filtered these out from our JaSt training set. After this process, we ended up with 2,889 malicious JS scripts in our training set. For the benign set, we retrieved 3,500 scripts from `cdnjs` [10], a popular open-source content delivery network system hosting widely used JS libraries. Using this training data, we trained the open-source version of JaSt [11] with default options for malicious JS classification, and from the 5-fold cross-validation, JaSt achieved 99.7% accuracy on the training data.

[9] We initially reached out to VirusTotal for this purpose. VirusTotal, however, denied our request, stating our experiment conflicted with their end-user licence agreement, and revoked our API key.

[10] `https://cdnjs.com/`

[11] `https://github.com/Aurore54F/JaSt`

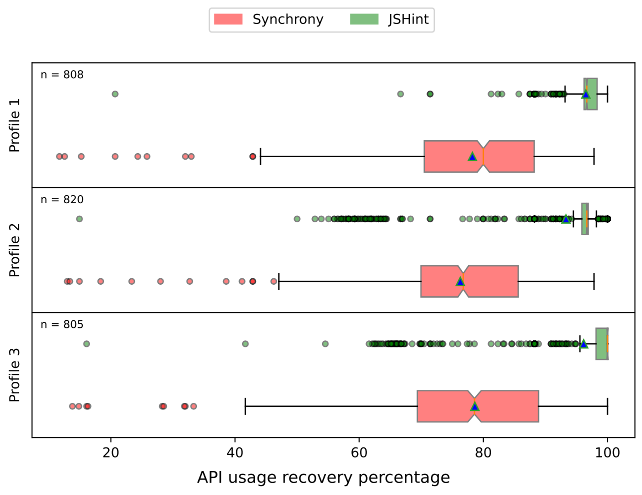

Fig. 6. Distribution of standard API recovery percentage between original and deobfuscated scripts by Synchrony compared to the between original and obfuscated code processed by JSHint versions for the obfuscation profiles (blue triangle indicates mean).

Table 3. JaSt detection evasions after introducing obfuscation and after performing API recovery.

	Evasions after Obfuscation		Evasions after API Usage Recovery	
Profile	Scripts	Percentage	Improvements	No Changes
#1	239	9.30%	227	12
#2	193	9.03%	171	22
#3	171	8.09%	118	53

5.2 Measuring Evasions

We applied the three obfuscation profiles described previously (see Table 2) on the set of filtered malicious JS scripts used for training JaSt, followed by applying JSHint on the obfuscated scripts. We received 2,570, 2,138, and 2,113 scripts for profiles 1, 2, and 3, respectively, without any errors and timeouts (a success rate of 88.96%, 74.00%, and 73.14% respectively).

We applied our trained JaStmodel on each obfuscated script and the corresponding output version from JSHint for all three profiles. As mentioned before, we mark an evasion when a malicious script with obfuscation results in a false negative from JaSt, meaning JaSt fails to classify the script as malicious despite being trained on the original version of the script. After applying JSHint, we consider an evasion to be improved if JaSt successfully classifies it as malicious, while the obfuscated version results in an FN. If JaSt considers the obfuscated and JSHint-generated version benign, we label it a "no-change" case.

Table 3 displays the evasions from JaSt detection in our obfuscated malicious scripts and their status after applying JSHint on the obfuscated versions. Introducing obfuscation results in 9.30%, 9.03%, and 8.09% evasions for the three profiles in order. However, after passing through JSHint, we were able to defeat 94.97%, 88.06%, and 69.00% of these evasions for the three profiles, respectively. 4.18%, 11.39%, and 30.40% of the evasions persisted through the transformation by JSHint. We had only a few cases (1, 1, and 5, respectively, for the three profiles) where JaSt detection degraded from a true positive to a false negative after applying JSHint.

We demonstrate that introducing obfuscation can enable JS-based malware to evade a highly trained detection system, even one that was already trained on the regular version of the script. We also observed that recovering the API usages through JSHint improved the detection rate. While more complex obfuscation does not necessarily result in better evasions, our gradually decreasing improvement rates and increasing persistent evasion rates suggest a potential threat.

6 Discussion & Future Work

Due to the dynamic nature of JavaScript, it is possible to have comples and bespoke obfuscation techniques. This technique can target JaSt like tools to avoid detection, make the source code more challenging to read for humans and LLMs, or introduce convoluted control flows to increase the complexity for static tools. We demonstrate a common side effect of these transformations is hiding the standard API usage, which obscures the effect and actions of the JS source code on the system. In our work, we demonstrate the sub-tree evaluation can uncover the hidden API usage introduced by obfuscation, which allows state-of-the-art detectors to perform better when evaluating potentially malicious javascript.

There are, however, limitations to JSHint, most specifically along the lines of optimization and latency. Most of the unsuccessful transformations were due to timeouts, which can be further lowered by optimizing `kscope`, our scope management tool to have better resource utilization. We should also be able to address the syntax breakages in output from JSHint by handling the edge cases that we observed from our sample set. We hope to include these updates to the next iteration of `kscope` and JSHint.

A natural extension of our work in the future is to extract IOCs from the JS source code processed by JSHint in the form of regex, YARA rules, and so on. We can leverage from using tools such as ioc-extractor [12]), followed by verifying the extracted IOCs. We consider this as an future enhancement of our work in this paper.

7 Related Work

There is extensive research for detecting obfuscated JS malware. JaSt [12] classifies malicious JS code through a random forest classifier on the AST patterns.

[12] https://github.com/ninoseki/ioc-extractor

Revolver [18] detected obfuscated evasive malware on the web. There are numerous examples of detection systems that use the presence of JS obfuscation as a feature in their detection system [2,20,23,35,8]. However, unlike these systems, we study the evasive effect of obfuscation on malicious JS, and prove that restoring API usages can result in better detection rate.

Notable attempts at removing obfuscation include static rule-based semantic JS rewriter such as `Maude` [5], automated simplification of JS obfuscation system presented by Lu et al. [25], and `JSDES` [1]. In contrast, our subtree evaluation-based system does not require expensive dynamic analysis and instead focuses on revealing API usages.

Deobfuscation systems for other scripting languages include `PSDEM` [24] for PowerShell and the lightweight deobfuscation system for PowerShell from Li et al. [22]. Unlike these approaches, we do not require obfuscation detection in our JS source code, and our API usage recovery can work beyond a predetermined set of obfuscation techniques.

8 Conclusion

Obfuscation that obscures API usages is widespread on the web and can easily be applied through off-the-shelf tools to achieve potent evasive malware. In this paper, we introduce a subtree evaluation-based JS standard API recovery system named JSHint. We demonstrate that our system can reveal almost all obfuscated standard API usages. Additionally, we demonstrate that simply applying JS obfuscation can prevent malicious JS scripts from being detected. Finally, we prove that restoring the API usages through JSHint can help the detection system defeat evasive, obfuscated JS code.

Acknowledgement

We thank the anonymous reviewers for their helpful feedback. This work was supported by the National Science Foundation (NSF) under grants CNS-2138138 and CNS-2047260. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

1. Moataz AbdelKhalek and Ahmed Shosha. JSDES: An Automated De-Obfuscation System for Malicious JavaScript. In *Proceedings of the 12th International Conference on Availability, Reliability and Security - ARES*, 2017.
2. I.A. Al-Taharwa, C.H. Mao, H.K. Pao, K.P. Wu, C. Faloutsos, H.M. Lee, S.M. Chen, and A.B. Jeng. Obfuscated malicious javascript detection by causal relations finding. In *Advanced Communication Technology (ICACT)*, 2011.
3. anseki. gnirts: Obfuscate string literals in JavaScript code. `https://github.com/anseki/gnirts`. Accessed: 03-14-2023.

4. Pavol Bielik, Veselin Raychev, and Martin Vechev. Programming with" big code": Lessons, techniques and applications. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
5. G Blanc, R Ando, and Y Kadobayashi. Term-Rewriting Deobfuscation for Static Client-Side Scripting Malware Detection. In *Mobility and Security (NTMS)*, 2011.
6. Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler : A Fast Filter for the Large-Scale Detection of Malicious Web Pages Categories and Subject Descriptors. In *Proceedings of the International World Wide Web Conference (WWW)*, 2011.
7. Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the International World Wide Web Conference (WWW)*, 2010.
8. Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ZOZZLE: Fast and precise In-Browser JavaScript malware detection. In *20th USENIX Security Symposium (USENIX Security 11)*, San Francisco, CA, August 2011. USENIX Association.
9. daftlogic.com. daft-logic: JavaScript Obfuscator. `https://www.daftlogic.com/projects-online-javascript-obfuscator.htm`. Accessed: 03-14-2023.
10. ECMA International. Ecmascript language specification. Standard ECMA-262, June 2011.
11. estools. Escope (escope) is ECMAScript scope analyzer extracted from esmangle project. `https://github.com/estools/escope`. Accessed: 03-14-2023.
12. Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28–29, 2018, Proceedings 15*, 2018.
13. Ben Feinstein and Daniel Peck. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. In *Black Hat USA*, 2007.
14. F. Howard. Malware with your Mocha? Obfuscation and antiemulation tricks in malicious JavaScript. In *Sophos Technical Papers (2010)*, 2010.
15. https://github.com/javascript-obfuscator/javascript-obfuscator. JavaScript Obfuscator. `https://obfuscator.io/`. Accessed: 03-14-2023.
16. HynekPetrak. HynekPetrak - JavaScript malware collection. `https://github.com/HynekPetrak/javascript-malware-collection`. Accessed: 03-14-2023.
17. javascriptobfuscator.com. javascript-obfuscator: The Most Secure Way to Protect JavaScript Code. `https://javascriptobfuscator.com/`. Accessed: 03-14-2023.
18. Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Proceedings of the USENIX Security Symposium*, 2013.
19. Kaspersky. Chrome 0-day exploit cve-2019-13720 used in operation wizardopium. `https://securelist.com/chrome-0-day-exploit-cve-2019-13720-used-in-operation-wizardopium/94866/`. Accessed: 11-11-2022.
20. Byung-Ik Kim, Chae-Tae Im, and Hyun-Chul Jung. Suspicious malicious web site detection with strength analysis of a javascript obfuscation. In *International Journal of Advanced Science and Technology*, 2011.
21. Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

22. Zhenyuan Li, Qi Alfred Chen, Chunlin Xiong, Yan Chen, Tiantian Zhu, and Hai Yang. Effective and light-weight deobfuscation and semantic-aware attack detection for powershell scripts. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.
23. Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, 2009.
24. Chao Liu, Bin Xia, Min Yu, and Yunzheng Liu. Psdem: A feasible de-obfuscation method for malicious powershell detection. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018.
25. Gen Lu and Saumya Debray. Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012.
26. metaes.org. MetaES: JavaScript metacircular interpreter. `https://github.com/metaes/metaes`. Accessed: 03-14-2023.
27. Marvin Moog, Markus Demmel, Michael Backes, and Aurore 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.
28. Palo Alto Networks. Malicious javascript injection campaign infects 51k websites. `https://unit42.paloaltonetworks.com/malicious-javascript-injection/`. Accessed: 11-11-2022.
29. npmjs.com. esvalidate - ecmascript validator. `https://www.npmjs.com/package/esvalidate`. Accessed: 03-14-2023.
30. rapid7. jsobfu: ruby-based JavaScript obfuscator. `https://github.com/rapid7/jsobfu`. Accessed: 03-14-2023.
31. Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from" big code". *ACM SIGPLAN Notices*, 2015.
32. relative. Syncrhony - JavaScript cleaner & deobfuscator. `https://github.com/relative/synchrony`. Accessed: 03-14-2023.
33. Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. Hiding in plain site: Detecting javascript obfuscation through concealed browser api usage. In *acm-imc*, 2020.
34. Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. Anything to hide? studying minified and obfuscated code in the web. In *Proceedings of the International World Wide Web Conference (WWW)*, 2019.
35. Wei Xu, Fangfang Zhang, and Sencun Zhu. Jstill: Mostly static detection of obfuscated malicious javascript code. In *Proceedings of the third ACM conference on Data and application security and privacy - CODASPY*, 2013.
36. zswang. jfogs: Javascript code obfuscator. `https://github.com/zswang/jfogs`. Accessed: 03-14-2023.