Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild

Marvin Moog*†, Markus Demmel*, Michael Backes†, and Aurore Fass†
*Saarland University
†CISPA Helmholtz Center for Information Security: {backes, aurore.fass}@cispa.de

***Abstract*—JavaScript is both a popular client-side programming language and an attack vector. While malware developers transform their JavaScript code to hide its malicious intent and impede detection, well-intentioned developers also transform their code to, e.g., optimize website performance. In this paper, we conduct an in-depth study of code transformations in the wild. Specifically, we perform a static analysis of JavaScript files to build their Abstract Syntax Tree (AST), which we extend with control and data flows. Subsequently, we define two classifiers, benefitting from AST-based features, to detect transformed samples along with specific transformation techniques.**

Besides malicious samples, we find that transforming code is increasingly popular on Node.js libraries and client-side JavaScript, with, e.g., 90% of Alexa Top 10k websites containing a transformed script. This way, code transformations are no indicator of maliciousness. Finally, we showcase that benign code transformation techniques and their frequency both differ from the prevalent malicious ones.

***Index Terms*—Web Security, JavaScript, Obfuscation, Minification, Empirical Study**

I. Introduction

JavaScript is a scripting language, which has become one of the core technologies of the Web platform. In particular, it is used as a client-side programming language by almost 97% of all websites [47]. Initially, JavaScript was invented to create sophisticated and interactive web pages. However, as it is executed in users' browsers, it also provides a basis for attacks. On the one-hand, JavaScript offloads the work to the client-side, meaning that it should be lightweight and fast to reduce loading times. Specifically, saving bandwidth will improve website performance. To this end, developers transform their code to reduce its size, e.g., by shortening the length of variables, inlining functions, and various optimization shortcuts [17], [32]. We use the term *minification* to refer to the transformation techniques aiming at reducing code size. Overall, developers can also choose to protect code privacy and intellectual property. For this purpose, there are some additional code transformations, which, e.g., make reverse-engineering harder without downgrading the performance too much. On the other hand, to perform malicious activities, attackers would like to evade detection. For this purpose, they transform their code too, to hide its malicious intent or at least make it harder to analyze, e.g., string manipulations, encoding, and logic structure obfuscation. With the term *obfuscation*, we refer to techniques, which aim at hindering code analysis.

This way, both benign and malicious JavaScript use code transformations but for different purposes. Due to their inherently different objectives, these transformations leave different traces in the source code syntax. In particular, the Abstract Syntax Tree (AST) represents the nesting of programming constructs. Therefore, *regular* (meaning non-transformed) JavaScript has a different AST than *transformed* code. Specifically, previous studies leveraged differences in the AST to distinguish benign from malicious JavaScript [5], [9], [14], [15]. Still, they did not discuss if their detectors confounded transformations with maliciousness or why they did not. In particular, there are legitimate reasons for well-intentioned developers to transform their code (e.g., performance improvement), meaning that code transformations are not an indicator of maliciousness.

In this paper, we conduct an in-depth empirical study of code transformations in the wild. Contrary to Skolka et al. [44], who performed an analysis of transformed code on the most popular websites, we do not focus solely on client-side JavaScript, but we also consider library code from npm and malicious JavaScript, for a comparative analysis. Contrary to their approach, we do not target the tools leveraged to modify the code. Instead, we dive into the specific techniques used to transform it to, e.g., highlight any differences in terms of techniques or usage frequency between benign vs. malicious code transformations, based on the ten techniques that we monitor. At the same time, we perform a longitudinal analysis of code transformations in the wild to study at large-scale the evolution of these techniques, depending on time or on specific trends. Since these transformation techniques mostly work at code level, e.g., to make it harder to reverse-engineer, we seek to directly analyze the patterns that specific transformations leave in the code structure. To directly work at code structure level, we chose to perform a static analysis of JavaScript files, also motivated by its speed and high code coverage. Specifically, we enhance the traditional AST with control and data flow information to abstract the source code, without losing its semantics. Next, we extract specific features from the AST, which are either typical of regular JavaScript or of a given transformation technique. Finally, we leverage two random forest classifiers, first to predict whether a given script is transformed or not, and second–in the case of a transformed script–to detect the specific techniques developers used.

In particular, we highlight the fact that code transformations are popular in the wild, both for websites (68.60% of the extracted scripts), npm packages (8.7% of the extracted scripts), and malicious JavaScript (29-73%). Specifically, websites are getting more transformed over time, meaning that web developers seem to be more concerned about bandwidth and loading times, i.e., are increasingly working on improving website performance. On the contrary, malicious JavaScript are more ephemeral, so that we rather observe trends specific to a month. Still, the transformation techniques used by benign developers are not changing, neither over time nor between client-side and library-based JavaScript, whereas malicious JavaScript uses different patterns. Specifically, the most popular benign transformation techniques relate to minification, mostly basic techniques, while the prevalent malicious ones include identifier and string obfuscation and aggressively minifying the code. Besides different transformation techniques, we also note that the remaining techniques that we monitor have differing usage frequency depending on whether the files are benign or malicious.

To sum up, our paper makes the following contributions:

- We abstract JavaScript files through their AST enhanced with control and data flows. Based on features we extract from the AST, we train two classifiers: while the first one distinguishes regular from transformed code, the second one can recognize the specific transformation techniques used.
- We evaluate our approach on a labeled dataset of regular and transformed JavaScript, also combining different transformation techniques.
- We perform a large-scale and comparative analysis of code transformations in the wild, where we target client-side, library-based, and malicious JavaScript.
- Finally, we conduct a longitudinal analysis of code transformations to study their evolution over time.

For reproducibility reasons, our source code is available [31].

II. Transforming JavaScript Code

This section first provides an overview of JavaScript obfuscation and minification techniques. Then, we select state-of-the-art systems to transform JavaScript code. Finally, we present the specific transformation techniques on which we focus in this paper.

A. *Code Transformation Techniques*

Obfuscation makes code harder to understand, both for human analysts and automatic tools. Several categories of code obfuscation can be found in the wild [19], [26], [51]:

- **Randomization obfuscation** consists in randomly inserting or changing elements of a script without altering its semantics, e.g., `whitespace` or `comments randomization`. Also, variable and function names can be randomized, e.g., to hinder manual analysis; we refer to this technique as `identifier obfuscation`.
- **Data obfuscation** regroups string and number manipulation techniques. For example, strings can be split, concatenated, or reversed so that they do not appear in plain text. Similarly, characters can be substituted, e.g., by running a regular expression replace on a string. Also, standard or custom encoding (such as ASCII, Unicode, or hexadecimal), encryption and decryption functions hinder a direct understanding of the code. We refer to these techniques as `string obfuscation`. Similarly, with `integer obfuscation`, a number does not appear in plain text but is, e.g., computed with arithmetic operators. Further techniques also enable to hide data. For example, with `obfuscated field reference`, the bracket notation is privileged over the dot notation to access an object property (as it enables to compute an expression, whereas dot notation only considers identifiers [34]). In addition, data can be fetched from a global array (`global array`) or rewritten, so that it does not contain any alphanumeric characters anymore [36] (`no alphanumeric`).
- **Logic structure obfuscation** directly targets the code logic, such as manipulating execution paths, e.g., by adding conditional branches. Another technique consists in adding irrelevant instructions (`dead-code injection`) or changing the program flow, e.g., by moving all basic blocks in a single infinite loop, whose flow is controlled by a *switch* statement [23] (`control-flow flattening`).
- **Dynamic code generation** leverages the dynamic nature of JavaScript to generate code on the fly, e.g., with *eval*.
- **Environment interactions** is specific to web JavaScript. In this case, statements can be scattered across an HTML document using multiple $<$ *script* $>$ blocks. Similarly, the payload could also be stored within the DOM and extracted subsequently so that it is not directly discernible.
- **Code protection** regroups techniques to protect code privacy and intellectual property, e.g., by impeding reverse engineering. Specifically, we focus on `self-defending`, which makes the code resilient against formatting and variable renaming [24]. In addition, we consider `debug protection`, which makes it harder to use features from the Developer Tool, for Chrome and Firefox [24].

On the contrary, minification aims at reducing code size, e.g., saving bandwidth improves website performance. Basic techniques consist of deleting whitespaces and comments (while obfuscation randomly added them to hinder the analysis), shortening variable names, and removing dead-code (`minification simple`). More advanced techniques directly modify the logic structure of the code, e.g., by eliminating unreachable or redundant code, inlining functions [17], or replacing an *if statement* with the conditional operator shortcut [32] (`minification advanced`).

B. *Code Transformation Tools*

To transform JavaScript code, we focus on several tools:

- **obfuscator.io** [24] is a highly configurable JavaScript obfuscator. For example, it includes several string obfuscation methods, self-defending, and control-flow flattening [23].
- **JSFuck** [27] is an obfuscator, which rewrites JavaScript code without any alphanumerical characters by only keeping the six following characters: "[", "]", "(", ")", "!", and "+".
- **gnirts** [2], which obfuscates strings in JavaScript code (cf. Section II-A), without using encoding escape.

- **custom-encoding**, our approach to obfuscate strings, with encoding.
- **JavaScript Minifier** [8], which minifies samples, e.g., by shortening variable names and removing whitespaces.
- **Google closure compiler** [16], which minifies JavaScript. It can perform more advanced optimizations, such as dead-code elimination, functions inlining, and constant folding.

We specifically selected (or implemented) these tools because they are configurable, meaning that for a given file, we could choose the specific code transformation technique(s) to use. Contrary to Skolka et al. [44], the tool used to perform the transformation is not relevant to us. Therefore, we did not consider two different tools performing the same transformation the same way.

C. Transformation Technique Selection

Based on the previous state-of-the-art tools to transform JavaScript code, along with their configuration settings, we can focus on the detection of the 10 following techniques: `identifier obfuscation`, `string obfuscation`, `global array`, `no alphanumeric`, `dead-code injection`, `control-flow flattening`, `self-defending`, `debug protection`, `minification simple`, and `minification advanced`.

Since we can leverage the previous tools to transform JavaScript code using one or several techniques, we are able to build a ground-truth dataset. This way, we can evaluate our classifier on the detection of specific techniques.[1] Also, we can still recognize techniques, which we do not monitor, as *transformed*, even though we do not name the specific technique, e.g., `obfuscated field reference`.

III. Transformed Code Detection

In this section, we present our approach, implemented in Python, to detect and analyze transformed code. First, we abstract input code with its AST, enhanced with control and data flows, and tokens. Second, we leverage the fact that regular and transformed code have a different structure (AST). This way, we extract specific features, which are typical of regular code or of specific transformations. Third, we define two multi-task detectors to a) detect transformed code and b), in the case of transformed code, predict the specific transformation techniques used. Fourth, we present our experimental approach to train our two detectors. Finally, we evaluate the performance of these two detectors on different ground-truth datasets.

A. Source Code Abstraction

To detect and analyze code transformations, we chose to perform a static analysis of JavaScript files to directly work at code level. In particular, the AST is a tree abstraction of the syntactic structure of the code. As it represents the way programmatic and structural constructs are arranged in a given file, different code transformations have a dissimilar AST. For example, the `string obfuscation` technique, which consists in splitting a string into several variables and concatenating them later on, will be characterized, e.g., by an unusually high number of variable declarations and binary expressions (to represent the "+" operator).

We use the AST generation of the open-source JavaScript static analyzer JSTAP [14], which augments the traditional AST from Esprima [18] with control and data flows. In particular, control flows enable to reason about conditions that should be met for a specific execution path to be taken. As for data flows, they represent the influence of a variable on another. We propose several adjustments to the original implementation. For example, we restrict flows of control to nodes having an impact on program execution paths, meaning *statement* nodes [11], *CatchClause*, and *ConditionalExpression*. Similarly, we only consider data flows on *Identifier* nodes, i.e., there is a data flow between two *Identifier* nodes if and only if a variable is defined at the source node and used at the destination node. We also improve the way to handle objects and scoping. For performance reasons, we set a two-minute timeout to generate data flow edges. After that, we consider the AST only enhanced with control flows. Finally, we also leverage Esprima to collect lexical units (i.e., tokens).

B. Feature Set

After building the AST, enhanced with control and data flows, and token information, we traverse the graph to extract specific features. We consider two feature categories, namely automatically selected features and hand-picked features.

First, we extract 4-gram features from the AST. It has indeed been shown that n-grams are an effective means for modeling reports [14], [15], [28]–[30], [39]. In fact, moving a window of length four over the list of syntactic units extracted enables to retain information about the code original syntactic structure.

Second, to determine features typical of specific transformation techniques, we performed an in-depth study of the transformation techniques we presented in Section II-A. In the following, we introduce a subset of the features we considered (for reproducibility reasons, we made our source code available [31]). To distinguish regular from transformed code, we leverage generic features, such as an AST depth and breadth divided by a script number of lines. We also consider more in-depth features, like the ratio of *MemberExpression* compared to the number of unique *Identifier* nodes, the proportion of *CallExpression*, *Literal*, and *Identifier* nodes, the presence or absence of specific built-in functions, or the number of string operations. Similarly, for the scripts reported as transformed, we focus on the specific transformation techniques used. To target minification, our feature set includes the average length of *Identifiers*, the average number of characters per line, or the proportion of *ternary operators* [32]. Regarding obfuscation techniques, we consider, e.g., the ratio of dot to bracket notations to access an object's properties [34], the average size of arrays/dictionaries, or the proportion of variables fetched from these structures (by leveraging data flows).

[1]We discuss limitations of our approach, which can recognize only these known techniques, in Section V-A

Finally, we construct a vector space such that each feature is associated with one consistent dimension, and its corresponding value is stored at this position in the vector. As a first pre-filtering layer, we consider one vector space to analyze files with the aim of distinguishing regular from transformed JavaScript code. In the following, we use the term *level 1* to refer to this first step. For the samples reported as transformed, we then construct a second vector space to detect the specific transformation techniques used on these inputs (similarly, we refer to this second step as *level 2*).

C. Detector Definition

The learning-based *level 1* and *level 2* detectors complete the design of our approach. In both cases, we define a multi-task system [6]. In particular, multi-task learning includes both a multi-class and a multi-label system. With multi-class classification, we consider more than two classes, e.g., for *level 2*: 10 transformation techniques. Still, each sample can only be assigned to one class. On the contrary, with multi-label classification, several labels can be assigned to a given sample. Formally, a multi-task system with C different classes is comparable to running C binary classifiers. Either the classifiers can be evaluated independently, or they can be arranged into a chain to account for possible correlations between the classes. In the second case, all classifiers use the same features, but the binary classifier at position P also leverages the predictions of all classifiers with a position in $[\![0, P-1]\!]$ as an additional feature [38]. For both multi-task classifiers, we use the Scikit-learn implementation [42].

First, to distinguish regular from transformed code, we define a multi-task detector with the following classes: regular, minified, and obfuscated. Since a file can be both obfuscated and minified, or even have a first part regular and a second part transformed (e.g., when a minified jQuery version is added to a regular sample), *level 1* can output several labels for a given input. We consider that a file is transformed if *level 1* flagged it as obfuscated and/or minified. Second, for transformed samples, we detect the specific transformation techniques that developers used. As previously, we define a *level 2* multi-task system with the 10 classes from Section II-C.

D. Detector Training

Next, we train our two detectors with regular files and samples we transformed using the tools presented in Section II-B.

1) Regular File Collection: To train our detectors, we first collected a set of regular JavaScript files from popular GitHub projects [46] and popular JavaScript libraries or other resources [7]. We removed files smaller than 512 bytes to keep only those with enough features to be representative of a class. For performance reasons, we chose to collect only samples smaller than 2 megabytes (even though our static approach can still analyze them). Also, to remove, e.g., JSON files, or samples containing exclusively comments, we only consider files with at least a conditional control flow node,[2] a function node,[3] or a *CallExpression* node[4] in their AST. Finally, we manually verified a subset of our collected samples to ensure that they are regular: of the 100 analyzed files, 99 are regular. Overall, we collected 21,000 scripts.

2) Training Set: Next, we built our dataset of transformed samples. For this purpose, we considered the previous 21,000 scripts and transformed them with one of the 10 techniques presented in Section II-C, for all techniques. This way, we transformed these 21,000 scripts 10 times. We stored these 10 transformed variants separately so that we do not mix different techniques at this stage. To train our *level 1* detector, we first removed 5,000 regular samples, which we will use as a validation set. Then, we randomly selected half of our remaining regular set (i.e., 8,000 scripts), as many minified, and as many obfuscated samples from the previously generated pools. For the minified files, we represent the 2 minification techniques equally (i.e., 4,000 files per category), and the same applies to the 8 obfuscation techniques (1,000 files per category). The process is similar for *level 2*, where we selected 2,000 samples for each of the 10 transformation techniques.

3) Validation Set: To optimize the predictions of our learning-based detectors, we performed an empirical study to evaluate several off-the-shelf systems. In particular, we compared two multi-task classifier implementations: a) classifiers chain [41] and b) classifiers independence assumption [43] (cf. Section III-C). Both for *level 1* and *level 2*, we randomly built a new dataset (disjoint from the training set) and, in both cases, the random forest classifier with the classifiers chain approach performed best. In the following, we always refer to this classifier for the learning based-detection.

E. Detector Accuracy

To evaluate the accuracy of our two detectors, we leverage three different datasets. First, we consider the remaining samples from Section III-D2 (disjoint from both training and validation sets). Second, we analyze 35,000 files, which we transformed with multiple techniques. Finally, we show that our approach generalizes to samples transformed with a new tool, namely the *Daft Logic* obfuscator [10], [12].

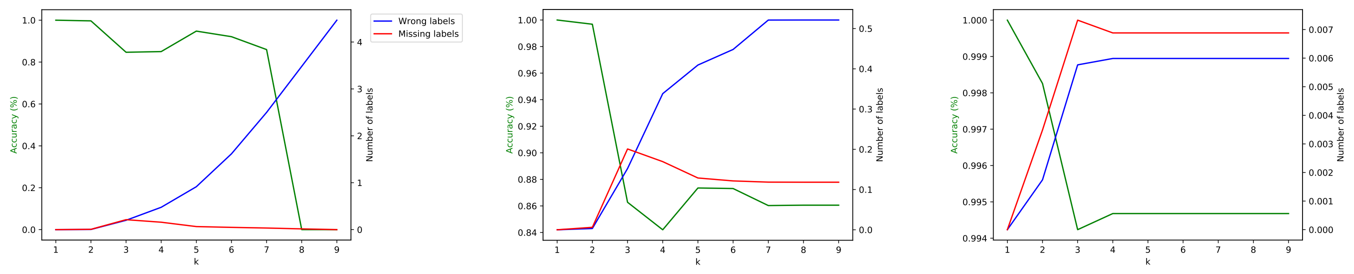
1) Test Set 1 - Remaining Samples: First, to evaluate the *level 1* detector, we consider the remaining 8,000 regular samples from Section III-D2, as many minified, and as many obfuscated. Overall, we accurately detect 7,892 files as regular (98.65%), 7,985 as obfuscated (99.81%), and 7,977 as minified (99.71%) so that we have a high detection accuracy of 99.41%. In the remaining sections, we consider files obfuscated or minified as *transformed* (accuracy: 99.69%), since we focus on the specific transformation techniques at *level 2*.

For *level 2*, we randomly selected 2,000 samples from Section III-D2, for each transformation technique (disjoint

[2] *DoWhileStatement*, *WhileStatement*, *ForStatement*, *ForOfStatement*, *ForInStatement*, *IfStatement*, *ConditionalExpression*, *TryStatement*, and *SwitchStatement*

[3] *ArrowFunctionExpression*, *FunctionExpression*, and *FunctionDeclaration*

[4] including *TaggedTemplateExpression*



(a) Threshold = 0%   (b) Threshold = 10%   (c) Threshold = 50%

Figure 1: Top-k score accuracy, depending on the threshold

from the previous sets). While we built this dataset with the tools from Section II-B, we used only one configuration per file. This way, we limited the number of multiple techniques used on a single file. Still, some tools always perform a specific technique in combination with others so that one given sample can have up to three different labels. We first measure the accuracy on this set by considering only the correct predictions, i.e., both the labels predicted, and their number must match the ground truth. We achieve an accuracy of 86.95%, which we deem to be high, given that our detector can make 1,023 different predictions.[5] Due to the high number of possible labels, another way of measuring the accuracy consists of using the Top-k score [22]. In our case, we consider that a Top-k prediction is correct when the k predictions with the highest probability are part of the expected ground-truth labels. For example, if we consider a multi-task detector D, which can predict the following labels {A, B, C, D}, and a file with the following ground truth [A, B, C]. If D makes the following predictions: Top-1 = [B], Top-2 = [B, C], Top-3 = [B, C, D], and Top-4 = [B, C, D, E], then only Top-1 and Top-2 are correct. Specifically, in our code transformation setting, Top-1 accuracy is 99.63%, Top-2 90.85%, and Top-3 98.95% (the remaining Top-k are 0%, as the ground truths have at most 3 labels here). Thus, considering the k most probable predictions of our classifier is an accurate way of detecting the specific transformation techniques used in given samples.

2) Test Set 2 - Mixed Samples: Next, we show that our detectors can also analyze files transformed with multiple techniques (as we envision that samples from the wild mix these techniques). To this end, we generated 35,000 new files, which we transformed with the tools from Section II-B by combining different configuration settings.

For *level 1*, we accurately detect 99.99% of the transformed files, compared to 99.69% in Section III-E1. We assume that mixing several techniques increases the proportion of features typical of transformed scripts, hence a higher accuracy.

For *level 2*, we use the Top-k metric to evaluate the accuracy. Contrary to Section III-E1, the ground truth may contain between 1 and 7 labels. Figure 1a presents the evolution of the detection accuracy when k increases, as well as the average number of wrong and missing labels. From $k = 7$, we have an artificial fast decline in terms of accuracy, as the ground truth contains at most 7 labels. Still, for $k = 8$, our detector would output 8 labels, possibly including labels with a very low probability of being correct. Therefore, we will only consider the first k labels if they have a probability of being correct over a threshold that we will determine. Our threshold should respond to the following challenges: 1) minimize the number of wrong labels, 2) maximize the number of detected techniques, and 3) maximize the accuracy. Specifically, choosing a very high threshold (i.e., considering predictions only if they have a very high confidence) would minimize the number of wrong labels and maximize the accuracy. In turn, we would only be able to detect a few transformation techniques. For example, even with a threshold of 50%, we could only recognize 3 or 4 transformation techniques (see Figure 1c), while we would like to recognize most of them. We empirically selected a threshold of 10%. As represented in Figure 1b, with such a threshold, we have less than 0.32 wrong labels on average and can detect up to 7 transformation techniques with an accuracy still over 89% (and over 99.84% for 1 and 2 techniques).

3) Test Set 3 - Daft Logic Samples: Finally, we show that our detectors generalize to samples transformed with another tool. For this purpose, we generated 10,000 new samples, which we transformed with a popular obfuscator [44], namely the Daft Logic obfuscator [10]–based on Dean Edwards packer [12].[6] Specifically, *level 1* accurately recognizes 99.52% of the samples as transformed. As for *level 2*, our Top-4 metric based on a threshold of 10% reports the following techniques: minification advanced and simple, identifier obfuscation, and string obfuscation, which is in line with the transformations performed by the packer. In fact, our approach generalizes, as it is not specific to a tool but recognizes generic transformation techniques through syntactic patterns.

IV. Large-Scale Analysis of Transformed Code

In this section, we present the results of our large-scale analysis of code transformations in the wild. We focus, in particular, on benign JavaScript extracted from the most popular websites and the most popular npm packages. Next, we

[5]The number of predictions from our classifier follows the *combination without repetition* distribution, knowing that, for a given file, our classifier can predict $k \in [\![1, 10]\!]$ labels. Therefore, the number of possible predictions is $\sum_{k=1}^{10} \binom{10}{k} = 1,023$

[6]We chose not to consider this tool in Section II-B, as it combines several techniques without providing a way for targeting specific ones at a time

Table I: JavaScript dataset description

Source	Creation	#JS	Class	Section
Alexa Top 10k	2020	46,238	Benign	Section IV-B1
npm Top 10k	2020	51,053	Benign	Section IV-B2
DNC	2015-2017	4,514	Malicious	Section IV-C
Hynek	2015-2017	29,484	Malicious	Section IV-C
BSI	2017	36,475	Malicious	Section IV-C
Alexa Top 2k * 65	2015-2020	327,164	Benign	Section IV-D1
npm Top 2k * 65	2015-2020	482,834	Benign	Section IV-D2

consider malicious JavaScript samples, which enables us to highlight transformation technique differences depending on the intent of the files. To avoid any bias due to our malware collection being older than our benign samples, we also perform a longitudinal analysis of benign code transformations between 2015 and 2020. Finally, we summarize our findings regarding the proportion of transformed code and specific techniques used in benign vs. malicious samples.

A. System Setup

Our large-scale analysis of transformed code in the wild rests on several datasets. For client-side JavaScript, we focus on the most popular Alexa websites [1], which we statically scraped, also including external scripts. We first focus on Alexa Top 10k in September 2020 (Section IV-B1), before considering Alexa Top 2k websites, one crawl per month between May 2015 and September 2020 (i.e., on 65 months, Section IV-D1). Given the fact that we statically extracted JavaScript from the start pages of high-profile sites, we assume this JavaScript collection to be benign. Similarly, for library-based JavaScript, we consider the most popular npm packages, based on their number of downloads [35] (first, Top 10k in Section IV-B2, then, Top 2k over 65 months in Section IV-D2). As previously, we assume them to be benign. Finally, we also analyze malicious JavaScript from three different sources (Section IV-C). In particular, we consider exploit kits provided by Kafeine DNC (DNC) [25], the malware collection of Hynek Petrak (Hynek) [20], as well as JScript-loaders from the German Federal Office for Information Security (BSI) [4]. Based on the classification of DNC, Hynek, and BSI, including, e.g., anti-virus systems, and a runtime-based analysis, we consider that these files are malicious.

As our datasets contain both benign (client-side and library-based) and malicious JavaScript, in the following sections, we will highlight any transformation technique differences depending on the intent of the scripts. Also, we will showcase any evolution in the transformed code landscape between 2015 and 2020. Table I summarizes the content of our datasets. Similarly to Section III-D1, we consider scripts between 512 bytes and 2 megabytes, containing at least a conditional control flow node, a function node, or a *CallExpression* node. We perform the following experiments with the *level 1* and *level 2* detectors, which we defined and trained in Section III-D.

B. Code Transformations in the Wild

In this section, we provide an in-depth study of the prevalence of code transformations in the wild. We first focus on client-side JavaScript with Alexa Top 10k websites before considering library-based code with npm Top 10k packages.

1) Alexa Top 10k Websites: In our first experiment, we leverage our *level 1* detector to classify JavaScript files extracted from the 10,000 most popular websites in September 2020. In particular, we found that code transformations (mostly minification) are highly used, with over 89.4% of the websites, which contain at least one transformed script. In fact, minification is especially used to reduce loading times, thus to improve website performance. A more in-depth study of the specific scripts included by Alexa Top 10k websites suggests that 68.60% of them are transformed, with over 68.20% of them reported as minified and 0.40% as obfuscated. To verify these results, we randomly selected 400 files, which we manually reviewed). In particular, out of the 100 files classified as regular, we confirm that 83 are regular. Regarding transformed samples, 96 / 100 are minified, 99 / 100 are obfuscated, and 100 / 100 are transformed. We noticed that manually assigning a label to samples is not straightforward because they may have been made harder to understand but without impeding the analysis too much. In this case, we mostly reported them as regular because they were, in general, solely using slight identifier obfuscation. Also, we observed samples that could be obfuscated, minified, and regular at the same time. In this case, if one class was significantly prevalent, we assigned the label of that class; otherwise, we considered multiple labels. Overall, our manual analysis underlines the fact that our system can detect transformed samples very accurately, while it slightly over-approximates the number of regular instances. As an additional verification step, we classified the dataset of 150,000 regular samples collected by Raychev et al. [37]. We retain an accuracy of 98.65%, which highlights the accuracy of our approach to detect regular samples too.

Contrary to our system, which detects 68.60% of the scripts extracted from Alexa Top 10k as transformed, Skolka et al. [44] found that 38% of the scripts extracted from the 100,000 most popular websites are transformed. Based on their results, we verified whether a website rank could influence the probability of it containing a transformed script. For Alexa Top 10k, we arranged the sites by groups of 1,000, ordered by popularity. We found that, while 72.35% of the scripts belonging to Alexa Top 10k, but not to Alexa Top 9k, are transformed, almost 80% of the Top 1k are transformed.[7] As a further comparison step, we extracted scripts belonging to Alexa Top 100k, but not to Alexa Top 90k, and reported 64.72% as transformed. These observations suggest that there is a link between website popularity and code transformations. Also, and contrary to our approach, Skolka et al. limited their analysis to files smaller than 40 kB (vs. 2 MB for us),

[7]These results differ slightly from Alexa Top 10k because, for Alexa Top 10k, we consider unique scripts, while in this setting, the scripts are unique per 1,000 website sampling but may appear in the ten samplings, which increases the weight of, e.g., widespread minified libraries

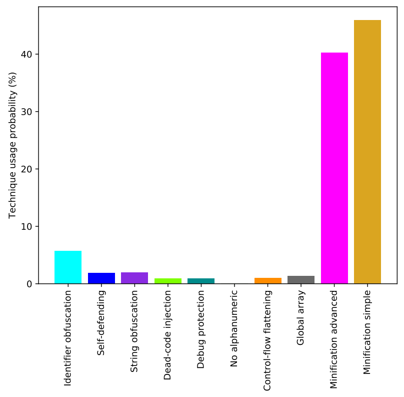


Figure 2: Transformation technique probability of being used in a transformed script from Alexa Top 10k

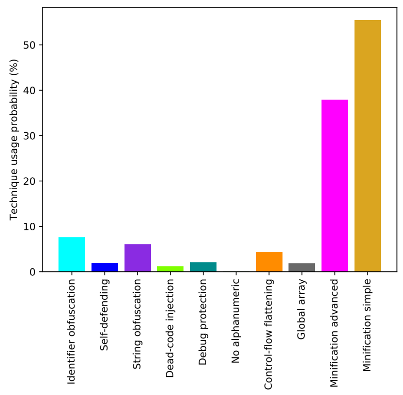


Figure 3: Transformation technique probability of being used in a transformed script from npm Top 10k

for performance reasons. Based on W3Techs report [48], we downloaded 120 minified versions of the most popular libraries and found that 87 (73%) of them have a size over 40 kB. For this reason, we assume that we reported more files as transformed than previous work for two reasons. First, we considered websites that are more popular, which we showed to be more transformed. Second, our design choices enabled us to analyze bigger files, which can be, e.g., minified libraries; thus, increasing the proportion of transformed samples detected.

Next, we focus on the samples reported as transformed. With our *level 2* detector, we determine which techniques are the most prevalent. For this purpose, we computed the average probability of a given technique being used, based on our detector confidence score. Since the predictions are independent, the sum of their confidence is not necessarily 100%. As indicated in Figure 2, and as expected, minification is highly used on the most popular websites, to reduce loading times. We observe, in particular, basic minification techniques (45.96%), e.g., variable shortening and whitespace deletion, but also more advanced ones (40.24%), e.g., offered by the Google closure compiler. Regarding the remaining transformation techniques we monitored, they are seldom used with a usage probability below 5.72% (identifier obfuscation) and even below 1.94% for the other techniques. Therefore, and as already observed at *level 1*, the transformed Alexa scripts are mostly minified, to improve website performance.

2) npm Top 10k Packages: As a second experiment, we focus on library-based JavaScript. To the best of our knowledge, such a study has not been performed before. We consider the 10,000 most downloaded npm packages, totaling between 332,946,417 (Top 1) and 136,395 (Top 10k) downloads between August and September 2020. Our *level 1* detector reports only 8.7% of the scripts as transformed (8.46% minified and 0.25% obfuscated), which is almost 8 times less than for Alexa. While Alexa samples are mostly minified, e.g., to reduce loading times and save bandwidth, it may not be necessary for npm packages. For example, when used with Node.js, `node` runtime has direct access to the `node_modules` folder. As no network access is

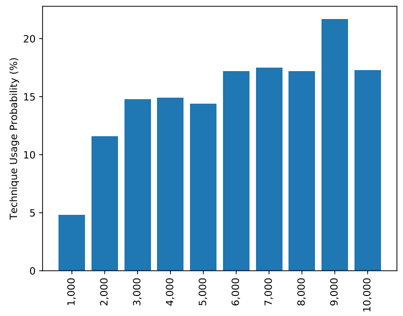
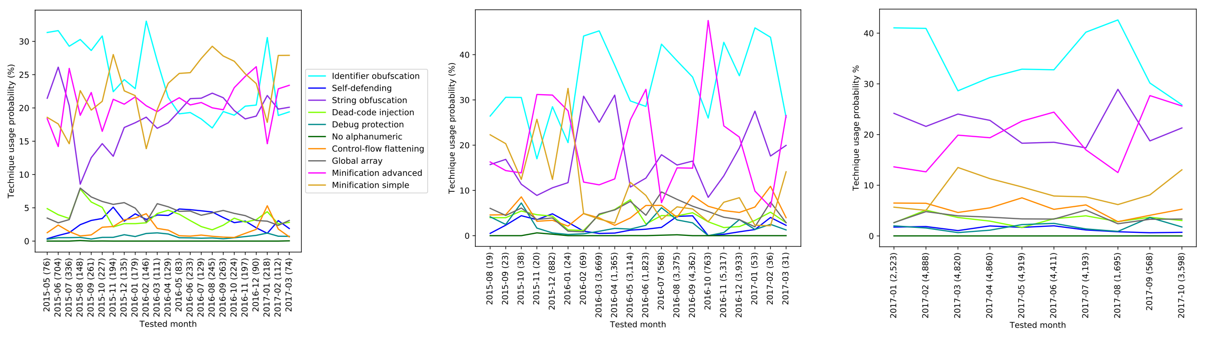


Figure 4: Evolution of the prevalence of transformed code in npm Top 10k packages depending on package popularity

required, transforming the code to, e.g., save bandwidth does not apply. Transformation techniques are rather applied to protect intellectual property (e.g., for proprietary packages), backward compatibility (which induces some transformations to use new syntax on older runtimes), or when the packages can directly be loaded in the Web. Overall, we found that 15.14% of the 10k most popular packages contain at least one transformed script, which is almost 6 times less than for Alexa.

Next, with *level 2*, we dive into the specific transformation techniques used on npm packages. Figure 3 sums up our main findings. As before, the two most prevalent techniques are related to minification, both basic (58.34%) and more advanced techniques (36.57%). As for Alexa, we computed these scores based on our detector's confidence for each prediction. Similarly to Alexa, we observe that when npm package developers transform their code, it is mostly with the aim of reducing its size. Still, we only report 8.7% of the scripts as transformed (compared to 68.60% for Alexa), meaning that minification is overall not very widespread on npm popular packages. Still, and contrary to Alexa, we observe that transformed scripts tend to be completely transformed, while Alexa samples combine regular with transformed code. In particular, out of the 100 minified Alexa samples we manually reviewed, 11 also include regular code, while we observed no such cases for npm. For this reason, we assume that our detector has a higher



(a) For DNC (b) For Hynek (c) For BSI

Figure 5: Transformation technique probability of being used in a malicious transformed sample

confidence in the prediction of npm minified samples (58.34% for minification simple) than of Alexa (45.96%).

Finally, we focus on the prevalence of code transformations depending on the rank of the considered packages. Similarly to the Alexa experiment, we split our 10k packages into 10 groups of 1k, ordered by popularity. As indicated in Figure 4, the 1k most popular packages are between 2.4 and 4.4 times less likely to contain transformed code than the remaining Top 10k packages. By diving into the specific transformation techniques used, we find that, while minification is the most prevalent technique, Top 1k packages equally use basic and advanced minification techniques (49% and 47%, respectively), while the Top 5k and Top 10k significantly privilege simpler techniques (58%) over more complex ones (37%).

C. *Code Transformations in Malicious JavaScript*

Previously, we showcased that both client-side and library-based benign JavaScript code can be transformed. Specifically, we focussed on the specific transformation techniques used and showed that their usage is similar between Alexa and npm, and that, in both cases, the most prevalent techniques are related to minification, for performance reasons. As a comparison, in this section, we focus on the transformation techniques used by malicious JavaScript samples.

The case of malicious JavaScript is slightly different from Alexa and npm Top 10k, as we cannot download popular malware for September 2020. In particular, our malicious JavaScript samples have been collected between 2015 and 2017 by DNC, Hynek, and BSI (Section IV-A). To avoid any bias due to a) the older character of our malware compared to the benign files and b) their collection over a two-year time-frame, we provide a comparative longitudinal study of benign code transformations in client-side and library-based JavaScript (between 2015 and 2020) in Section IV-D. In the current section, we analyze our malicious JavaScript datasets with our *level 1* and *level 2* detectors.

1) Level 1 Detector: First, and contrary to what we were expecting, most of our malicious samples are not necessarily detected as transformed. In particular, we classify only 28.93% of the BSI samples, 65.94% of DNC, and 73.07% of Hynek as transformed. We assume that these results differ across our three datasets, as they were provided by three different entities, which collected different kinds of malicious JavaScript, e.g., exploit kits vs. JScript-loaders. Given these results, we manually reviewed 100 randomly selected malicious instances recognized as regular and 100 as transformed (equally split across our three sources). Of the 100 regular files, we confirm that 25 are regular (the malicious logic stays in the open for 23, and 2 use conditional compilation [33], which Esprima parses as a large comment). Of the remaining ones, 57 are rather transformed but solely rely on identifier obfuscation, meaning that their syntactic structure looks very regular (i.e., only variable names are randomized). As for the 18 remaining ones, it was complicated for us to decide on a label because they generated code dynamically. Specifically, for each sample, we observed a few non-human readable fixed strings, which subsequently flew into `eval`. Regarding the transformed samples, our predictions are correct for 97 of them, while 3 are rather regular as their malicious logic is mostly staying in the open.

Overall, we assume that two reasons are responsible for the observed low transformed rates. First, malware samples are not necessarily completely transformed. For example, to evade detection, malware authors can combine a small and slightly obfuscated payload with a significantly larger amount of regular code. Due to the majority of the code being regular, our system would (correctly) classify the code as regular. Second, malicious actors can leverage specific transformation techniques, e.g., identifier obfuscation, to generate syntactically identical, but SHA-1 unique, malicious instances [15], which are broadcast in waves, one unique malicious script per victim, to, e.g., impede signature-based detection. This way, we observe similar classification results inside a wave (as a wave contains syntactically similar malicious instances), while the remaining samples may be very different. In particular, the proportion of transformed samples is very different from a month, and even a malware source, to another.

2) Level 2 Detector: For the malicious samples previously detected as transformed, we focus now on the specific transformation techniques they use. Out of the 100 samples classified as transformed that we manually reviewed, we stress that 97 are indeed transformed (and the 3 remaining ones are partially transformed but the malicious logic still apparent) so that we

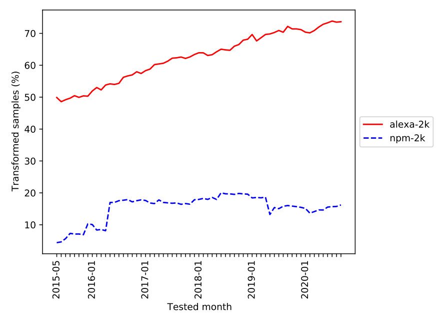


Figure 6: Evolution of the proportion of transformed scripts

are not introducing a bias in the following results. Figure 5 sums up our main observations. As the number of samples collected each month is not constant, we indicate the number of files next to each tested month in the x-axis. Also, we chose to separate the malware per source to avoid any transformation technique bias due to different providers, which collected different types of JavaScript files, e.g., exploit kits for DNC vs. JScript-loaders for BSI. We would also like to stress that malware is ephemeral, i.e., we are not comparing the same malware over time but different malicious waves, which may be independent or stem from different attackers.

Compared to Alexa Top 10k (Figure 2) and npm Top 10k (Figure 3), where *level 2* reported minification as the most prevalent transformation technique with a probability between 36-59%, malicious transformation techniques are both different and have a differing probability than for benign client-side and library code. Specifically, for our three malicious JavaScript sources, the most popular transformation technique is identifier obfuscation, with an average usage probability between 25-37%, compared to below 6.2% for benign scripts. Additional popular techniques include string obfuscation and minification advanced, which both have an average usage probability between 17-21%, compared to below 3.3% (string obfuscation) and over 36.5% (minification advanced) for benign inputs. For DNC, minification simple is also widespread (average usage of 22% vs. over 45.96% for benign scripts), which our manual analysis confirms. As for the remaining obfuscation techniques, dead-code injection, control-flow flattening, and global array appear between 5-10% of the time, which is again superior to their benign usage of mostly 1%. This way, we showcased that while both benign and malicious JavaScript files use transformation techniques, the former privilege minification to improve performance, e.g., by reducing loading times. In contrast, the latter favor other transformation techniques, combined with aggressive minification, to make malicious JavaScript code harder to understand and analyze.

D. Longitudinal Analysis of Code Transformations in the Wild

As a comparison with malicious samples that were collected between 2015-05 and 2017-10, we perform a longitudinal analysis of code transformations on benign samples. We focus on Alexa Top 2k websites and npm Top 2k packages, which we collected between 2015-05 and 2020-09.

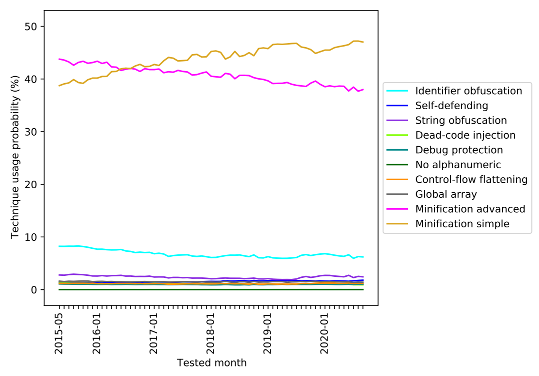


Figure 7: Transformation technique probability of being used in a transformed script from Alexa Top 2k over time

1) JavaScript From Alexa: First, we leveraged the Wayback Machine from the Internet Archive [21], [45] to collect the 2,000 most popular Alexa websites the first of each month between 2015 and 2020. As indicated in Figure 6, we observe a steady augmentation of the proportion of transformed scripts over time. By diving into the specific transformation techniques (Figure 7), we infer that this augmentation is solely due to a minification usage increase (from 38.74% in 2015 to 47.02% in 2020 for minification simple, while minification advanced slightly decreases from 43.77% to 40%). Therefore, we observe that web developers have become more concerned with the performance of their websites over time and are increasingly trying to save bandwidth and loading times. As for the remaining transformation techniques, they are staying more constant over time. For example, identifier obfuscation slightly decreases from 8.23% to 6.21%, while the other techniques have a usage probability under 2.4%, on average, meaning that benign websites do not really use them in practice. These results are also in line with our observations from Figure 2, even though we cannot directly compare Alexa Top 10k with the Top 2k websites.

2) JavaScript From npm Packages: Similarly, we collected the 2,000 most popular npm packages (based on the number of downloads at the end of each month) between 2015 and 2020.[8] Contrary to Alexa, npm packages are more ephemeral so that we are not necessarily comparing the same packages over time. In particular, we distinguish three phases in Figure 6. From 2015-05 to 2016-04, we classified on average 7.4% of the extracted scripts as transformed. The corresponding very high relative standard deviation of 24.22% indicates that data is spread out, which we explain by the fact that, on average, only 76.7% of the most popular npm packages on a given month are still popular on the next month. On the contrary, from 2016-05 to 2019-05, we observe a more consistent trend with 17.95% of the scripts classified as transformed, with a small relative standard deviation of 5.9%, and almost 93% of the packages are common between two consecutive

[8]We collected the packages to download from https://api.npmjs.org/downloads/point/{period}[/{package}], for example, https://api.npmjs.org/downloads/point/2018-05-01:2018-05-01/supports-color

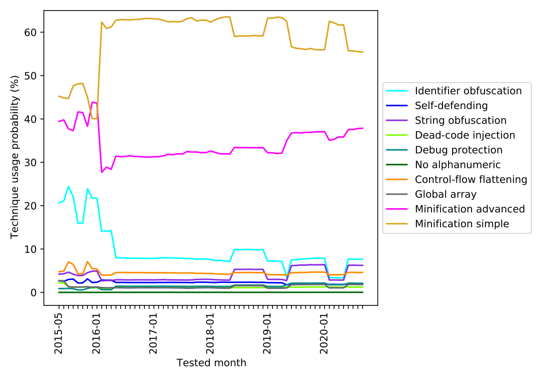


Figure 8: Transformation technique probability of being used in a transformed script from npm Top 2k over time

months. Similarly, from 2019-06 to 2020-09, we also observe a consistent trend of 15.17% of transformed scripts and 87.48% of common packages. We manually reviewed a subset of our predictions (for the different phases), where we retain a high accuracy with 50 / 50 files accurately classified as regular and 47 / 50 files accurately classified as transformed (more precisely as minified). We could not infer any relationships between the number of transformed scripts and the increasing/decreasing popularity of npm minifiers, though. We rather assume that the three phases we observed are typical of the ephemeral state of npm packages, which depend on current trends. Still, our observed 7-18% of transformed script rate confirms our observation from Section IV-B2 in the sense that npm packages mostly do not abuse code transformations as they do not need to, e.g., reduce bandwidth or loading times.

Regarding specific transformation techniques, as shown in Figure 8, we observe the same three stages as previously for the two most prevalent techniques, namely minification simple and advanced (with an average probability of 58.62% and 34.28%, respectively), as well as for the less popular identifier obfuscation technique (9.71%). As for the less common ones, they rather stay consistent over time, with probabilities mostly below 3%. This way, we do not observe a specific evolution in the transformed code landscape of npm packages over time. While the most prevalent transformation technique stays minification, its usage depends on current package trends.

E. Summary of Code Transformations

In the previous sections, we confirm that both benign and malicious JavaScript use code transformations. In particular, the most prevalent technique for both client-side and library-based benign code is minification. While we detected minification in 68.20% of the scripts extracted from websites, we found only 8.46% of minified npm files. In fact, web developers choose to optimize website performance to save bandwidth and loading times, whereas it does not apply to npm packages, which generally do not need any network access. As for malicious JavaScript samples, they abuse and combine many transformation techniques including, but not limited to, aggressive minification. While we also reported on benign obfuscated files, e.g., with identifier obfuscation, our detector predicted this technique with an average usage below 6.2% for benign scripts (both Alexa and npm), compared to over 25-37% for our malicious samples. Similarly, the probability that malicious files use string obfuscation lies between 17-21%, compared to less than 3.3% for benign scripts. This highlights the fact that the main reason for transforming malicious samples is to make them harder to understand. While benign code can also be transformed to impede its analysis, this is less prevalent than for malicious samples. As for the remaining obfuscation techniques that we monitored, benign developers have a probability of using them below 3%, whereas more than half of the techniques have a usage probability between 5-10% in malware.

Regarding the evolution of the transformed code landscape over time, we noticed that both client-side and library-based JavaScript tend to get more transformed. For example, the linear increase of minification in websites suggests that web developers are getting more concerned with website performance over time. Regarding malicious JavaScript, the trend is very different, as malware is ephemeral, so that we are not comparing the same samples over time. Still, we note that the three most popular malicious transformation techniques stay the same over time, namely identifier and string obfuscation, and aggressively minifying the code, as discussed previously. This way, we quantified the fact that both benign and malicious JavaScript samples are transformed. Still, developers have different reasons that motivate the use of differing code transformation techniques, with a different usage probability, for benign vs. malicious JavaScript code.

V. Discussion

In this section, we first examine the limitations our approach might have before introducing potential improvements.

A. Limitations

We chose to perform a static analysis of JavaScript files to directly see the traces left by different transformation techniques in the files' syntax. By design, we cannot analyze dynamically generated code nor transformation techniques used at runtime, e.g., several *eval* unfolding layers. Similarly, we cannot detect samples transformed with HideNoSeek [13], as it rewrites malicious JavaScript to reproduce an existing benign syntax, which does not leave any trace in the code structure. Still, we are not trying to infer the behavior of a script but to study transformation traces directly accessible in its syntax. In particular, we chose to focus on ten transformation techniques, which we selected based on the transformation capabilities of available JavaScript minifiers and obfuscators. For this reason, our *level 2* detector is limited to the ten techniques we monitor (while we discussed additional existing techniques in Section II-A). Still, our *level 1* detector can recognize samples as transformed, even if they use techniques that we do not monitor. Similarly, our approach is not limited to the tools we considered to generate our training sets but generalizes to other tools, as it is tailored to recognize generic transformation techniques through syntactic patterns. This is confirmed in Section III-E3 and with our manual analyses in Sections IV-B1, IV-B2, and IV-C.

B. Potential Improvements

In this paper, we measure the prevalence of JavaScript code transformations in the wild. In particular, we showcase that both benign and malicious JavaScript transform their code, meaning that code transformation is no indicator of maliciousness. In particular, we show that specific techniques and their usage probability differ between benign and malicious scripts. For this reason, we believe that our large-scale analysis and measurement study are useful to the community and will contribute to implementing new malware detection systems. To this end, we could consider patterns that are solely present in malicious files to reduce the number of false positives due to benign obfuscated samples. However, this extension is outside the scope of this paper, and we leave it for future work.

VI. Related Work

In this paper, we perform a large-scale analysis of code transformations in the wild. While obfuscation has been analyzed before, we provide an in-depth study of specific transformation techniques and highlight the fact that code transformations used in malicious code differ from the prevalent benign transformation techniques. We first present works related to transformed JavaScript before focussing on malicious JavaScript and finally on obfuscation beyond JavaScript.

Detecting Transformed JavaScript — In the literature, several approaches have been proposed to recognize transformed, or at least obfuscated, JavaScript code. Specifically, Kaplan et al. [26] introduced NoFus, their bayesian classifier trained over the AST to distinguish obfuscated from non-obfuscated code. Similarly, with JSOD, Blanc et al. [3] proposed an anomaly-based detection system over the AST to detect obfuscated scripts including readable patterns. Likarish et al. [30] defined specific features based on detecting obfuscation to recognize malicious JavaScript. With JStill, Xu et al. [52] leveraged the fact that malicious JavaScript code has to be deobfuscated before performing its intended behavior. Thus, they detected obfuscation given specific deobfuscation functions. Similarly, Sarker et al. [40] rely on the fact that obfuscation aims at hiding a script's behavior, meaning that if the results of a dynamic analysis differ from a static analysis, then the script's behavior is obfuscated. Still, none of these approaches focus on the specific transformation techniques that developers use nor on benign vs. malicious transformation techniques. While Xu et al. [51] analyzed the usage of some obfuscation techniques, they manually reviewed 100 malicious samples. In contrast, our approach is automated and can analyze JavaScript code from the wild at scale. Finally, Skolka et al. [44] performed an empirical study of transformed code on the Web. In particular, they focus on the tools that developers used to obfuscate or minify their code, while we dive into (and detect) the specific transformation techniques. Besides, they analyzed client-side JavaScript, while we also consider library-based code with npm, and malicious samples, to identify any difference in the transformation process, depending on the intent of the JavaScript samples, as well as their evolution over time.

Detecting Malicious JavaScript — Code transformations, more specifically obfuscation, should not be confounded with maliciousness. In the following, we present some tools that statically detect *malicious* JavaScript inputs. Rieck et al. introduced CUJO [39], which contains a static detector leveraging n-grams upon lexical units. With ZOZZLE, Curtsinger et al. [9] combined a Bayesian classifier with features extracted from the AST to detect malicious JavaScript. Similarly, Fass et al. proposed JAST [15], an n-gram approach based on features from the AST. Later, they introduced JSTAP [14] to go beyond the syntactic structure and consider control and data flows.

Detecting Obfuscation — Finally, code transformations are not limited to JavaScript. Specifically, Wermke et al. [50] focussed on software obfuscation on Android applications, while Wang et al. [49] studied software obfuscation techniques on mobile applications from Apple App Store.

VII. Conclusion

This paper measures the prevalence of JavaScript code transformations in the wild. In particular, both well-intentioned and malware developers use code transformations on JavaScript. Due to the inherently different intent of their scripts, they transform their code for different reasons, e.g., performance improvement vs. analysis impediment. These different expectations from code transformations naturally lead to distinct techniques, which leave different traces in the source code syntax. In this paper, we study how code transformations used in malicious JavaScript code differ from transformations typically used in benign scripts, and how code transformations evolve over time. To this end, we define a learning-based pipeline to 1) distinguish regular from transformed scripts and 2) recognize the specific transformation techniques used.

In practice, code transformations are widespread, both for malicious and benign JavaScript, e.g., 89.4% of Alexa Top 10k websites contain at least a transformed script. In addition, we observe that client-side JavaScript is getting more transformed (minified) over time, meaning that web developers are getting more concerned about reducing loading times and improving website performance. In contrast, the most popular transformation techniques used by malicious JavaScript do not change over time. Finally, we showcase that transformation techniques used by malicious JavaScript differ from the prevalent benign techniques, which are all the more common between client-side and library-based code.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. In particular, we thank our shepherd Yinzhi Cao for his guidance and support in preparing the final version of this paper. We also thank the German Federal Office for Information Security (BSI) and Kafeine DNC, which provided us with malicious JavaScript for our experiments. Special thanks also go to Ben Stock and Cris Staicu for their appreciated feedback.

References

[1] Alexa Internet, "Alexa Top Sites," http://www.alexa.com/topsites. Accessed on 2020-10-17.

[2] anseki, "Obfuscate string literals in JavaScript code," https://github.com/anseki/gnirts. Accessed on 2020-06-13.

[3] G. Blanc, D. Miyamoto, M. Akiyama, and Y. Kadobayashi, "Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts," in *International Conference on Advanced Information Networking and Applications Workshops*, 2012.

[4] BSI, "German Federal Office for Information Security (BSI)," https://www.bsi.bund.de/EN. Accessed on 2021-03-18.

[5] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages," in *International Conference on World Wide Web (WWW)*, 2011.

[6] R. Caruana, "Multitask Learning," in *Machine Learning*, 1997.

[7] chencheng, "awesome-javascript," https://github.com/sorrycc/awesome-javascript. Accessed on 2020-04-29.

[8] A. Chilton, "JavaScript Minifier," https://javascript-minifier.com. Accessed on 2020-12-08.

[9] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection," in *USENIX Security*, 2011.

[10] Daft Logic, "Daft Logic: Online Javascript Obfuscator," https://www.daftlogic.com/projects-online-javascript-obfuscator.htm. Accessed on 2020-12-08.

[11] Ecma International, "ECMAScript 2020 Language Specification," https://www.ecma-international.org/ecma-262. Accessed on 2020-12-08.

[12] D. Edwards, "dean.edwards.name/packer/," http://dean.edwards.name/packer/. Accessed on 2020-06-15.

[13] A. Fass, M. Backes, and B. Stock, "HIDENOSEEK: Camouflaging Malicious JavaScript in Benign ASTs," in *CCS*, 2019.

[14] ——, "JSTAP: A Static Pre-Filter for Malicious JavaScript Detection," in *ACSAC*, 2019, code repository: https://github.com/Aurore54F/JStap.

[15] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "JAST: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript," in *DIMVA*, 2018.

[16] Google Developers, "Closure Compiler," https://developers.google.com/closure/compiler. Accessed on 2020-12-08.

[17] ——, "Closure Compiler: Advanced Compilation," https://developers.google.com/closure/compiler/docs/api-tutorial3. Accessed on 2020-12-08.

[18] A. Hidayat, "ECMAScript Parsing Infrastructure for Multipurpose Analysis," http://esprima.org. Accessed on 2020-12-08.

[19] F. Howard, "Malware with your Mocha? Obfuscation and Anti Emulation Tricks in Malicious JavaScript," Sophos, Tech. Rep., 2010.

[20] Hynek Petrak, "Javascript Malware Collection," https://github.com/HynekPetrak/javascript-malware-collection. Accessed on 2020-06-16.

[21] Internet Archive, "Search the history of over 486 billion web pages on the Internet," https://archive.org. Accessed on 2020-12-08.

[22] K. Järvelin and J. Kekäläinen, "IR Evaluation Methods for Retrieving Highly Relevant Documents," in *ACM SIGIR Conference on Research and Development in Information Retrieval*, 2000.

[23] Jscrambler, "Control Flow Flattening," https://docs.jscrambler.com/code-integrity/tutorials/control-flow-flattening?utm_source=blog.jscrambler.com&utm_medium=referral&utm_campaign=101-cff. Accessed on 2020-06-07.

[24] T. Kachalov, "JavaScript Obfuscator Tool," https://obfuscator.io. Accessed on 2020-12-08.

[25] Kafeine, "MDNC - Malware don't need coffee," https://malware.dontneedcoffee.com. Accessed on 2020-06-15.

[26] S. Kaplan, B. Livshits, B. Zorn, C. Siefert, and C. Curtsinger, ""NOFUS: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code"," in *Microsoft Research Technical Report*, 2011.

[27] M. Kleppe, "JSFuck," https://github.com/aemkei/jsfuck. Accessed on 2020-12-08.

[28] J. Z. Kolter and M. A. Maloof, "Learning to Detect and Classify Malicious Executables in the Wild," in *Journal of Machine Learning Research*, 2006.

[29] P. Laskov and N. Šrndić, "Static Detection of Malicious JavaScript-Bearing PDF Documents," in *ACSAC*, 2011.

[30] P. Likarish, E. Jung, and I. Jo, "Obfuscated Malicious JavaScript Detection Using Classification Techniques," in *International Conference on Malicious and Unwanted Software (MALWARE)*, 2009.

[31] MarM15, "Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild," https://github.com/MarM15/js-transformations.

[32] Mozilla Developer Network, "Conditional (ternary) operator," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator. Accessed on 2020-06-07.

[33] ——, "JavaScript Conditional Compilation: cc_on," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Microsoft_Extensions/at-cc-on. Accessed on 2020-06-13.

[34] ——, "Property accessors," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors. Accessed on 2020-06-07.

[35] npm, "npm: download-counts," https://github.com/npm/download-counts. Accessed on 2020-10-17.

[36] P. Palladino, "Non alphanumeric JavaScript," http://patriciopalladino.com/blog/2012/08/09/non-alphanumeric-javascript.html. Accessed on 2020-12-08.

[37] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning Programs from Noisy Data," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016.

[38] J. Read, B. Pfahringer, G. Holmes, and E. Frank, "Classifier Chains for Multi-Label Classification," in *Machine Learning*, 2011.

[39] K. Rieck, T. Krueger, and A. Dewald, "CUJO: Efficient Detection and Prevention of Drive-by-Download Attacks," in *ACSAC*, 2010.

[40] S. Sarker, J. Jueckstock, and A. Kapravelos, "Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage," in *ACM Internet Measurement Conference (IMC)*, 2020.

[41] scikit-learn developers, "Scikit-learn: ClassifierChain," https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.ClassifierChain.html. Accessed on 2020-06-08.

[42] ——, "Scikit-learn: Multiclass and Multilabel Algorithms," https://scikit-learn.org/stable/modules/multiclass.html. Accessed on 2020-06-08.

[43] ——, "Scikit-learn: MultiOutputClassifier," https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.MultiOutputClassifier.html. Accessed on 2020-06-08.

[44] P. Skolka, C.-A. Staicu, and M. Pradel, "Anything to Hide? Studying Minified and Obfuscated Code in the Web," in *The World Wide Web Conference (WWW)*, 2019.

[45] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security," in *USENIX Security*, 2017.

[46] O. Trekhleb, "Top 33 JavaScript Projects on GitHub," https://itnext.io/top-33-javascript-projects-on-github-ad9d1dc822f7. Accessed on 2020-04-29.

[47] W3Techs, "Historical trends in the usage statistics of client-side programming languages for websites," https://w3techs.com/technologies/history_overview/client_side_language/all. Accessed on 2020-10-18.

[48] ——, "Usage statistics of JavaScript libraries for websites," https://w3techs.com/technologies/overview/javascript_library. Accessed on 2020-12-07.

[49] P. Wang, Q. Bao, L. Wang, S. Wang, Z. Chen, T. Wei, and D. Wu, "Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation," in *International Conference on Software Engineering (ICSE)*, 2018.

[50] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A Large Scale Investigation of Obfuscation Use in Google Play," in *ACSAC*, 2018.

[51] W. Xu, F. Zhang, and S. Zhu, "The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study," in *International Conference on Malicious and Unwanted Software (MALWARE)*, 2012.

[52] ——, "JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code," in *ACM conference on Data and application security and privacy (CODASPY)*, 2013.