Anything to Hide?
Studying Minified and Obfuscated Code in the Web

Philippe Skolka
Department of Computer Science
TU Darmstadt

Cristian-Alexandru Staicu
Department of Computer Science
TU Darmstadt

Michael Pradel
Department of Computer Science
TU Darmstadt

ABSTRACT

JavaScript has been used for various attacks on client-side web applications. To hinder both manual and automated analysis from detecting malicious scripts, code minification and code obfuscation may hide the behavior of a script. Unfortunately, little is currently known about how real-world websites use such code transformations. This paper presents an empirical study of obfuscation and minification in 967,149 scripts (424,023 unique) from the top 100,000 websites. The core of our study is a highly accurate (95%-100%) neural network-based classifier that we train to identify whether obfuscation or minification have been applied and if yes, using what tools. We find that code transformations are very widespread, affecting 38% of all scripts. Most of the transformed code has been minified, whereas advanced obfuscation techniques, such as encoding parts of the code or fetching all strings from a global array, affect less than 1% of all scripts (2,842 unique scripts in total). Studying which code gets obfuscated, we find that obfuscation is particularly common in certain website categories, e.g., adult content. Further analysis of the obfuscated code shows that most of it is similar to the output produced by a single obfuscation tool and that some obfuscated scripts trigger suspicious behavior, such as likely fingerprinting and timing attacks. Finally, we show that obfuscation comes at a cost, because it slows down execution and risks to produce code that changes the intended behavior. Overall, our study shows that the security community must consider minified and obfuscated JavaScript code, and it provides insights into what kinds of transformations to focus on. Our learned classifiers provide an automated and accurate way to identify obfuscated code, and we release a set of real-world obfuscated scripts for future research.

CCS CONCEPTS

• **Security and privacy → Web application security**; *Software reverse engineering*; *Intrusion/anomaly detection and malware mitigation.*

KEYWORDS

obfuscation; web security; empirical study; machine learning

ACM Reference Format:
Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to Hide? Studying Minified and Obfuscated Code in the Web . In *Proceedings of the 2019 World Wide Web Conference (WWW '19), May 13–17, 2019, San Francisco, CA, USA,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3308558.3313752

WWW '19, May 13–17, 2019, San Francisco, CA, USA
2019. ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3313752

1 INTRODUCTION

JavaScript has become the dominant programming language for client-side web applications and nowadays is used in the vast majority of all websites. The popularity of the language makes JavaScript an attractive target for various kinds of attacks. For example, cross-site scripting attacks try to inject malicious JavaScript code into websites [14, 17, 28]. Other attacks aim at compromising the underlying browser [6, 7] or extensions installed in a browser [11, 12], or they abuse particular web APIs [19, 26, 30].

An effective way to hide the maliciousness of JavaScript code are code transformations that preserve the overall behavior of a script while making it harder to understand and analyze. Such transformations affect both manual code inspection, e.g., because the code becomes harder to understand, and automated code analysis, e.g., because the malicious behavior is disguised as apparently harmless operations. There exist a variety of code transformations, ranging from renaming of local variables to more complex code changes that affect the control flow and data flow. We refer to transformations aimed at reducing code size, typically by renaming local variables to shorter names, as *minification.* In contrast, we refer to more complex transformations aimed at hindering the understanding and analysis of code as *obfuscation.* It is important to note that minification and obfuscation may be used for legitimate reasons, such as reducing code size or protecting intellectual property. However, independently of what the reason for transforming code is, it affects the ability of human and automated security analysis.

Despite the potential impact that minification and obfuscation may have on security analysis, little is currently known about how real-world websites use such transformations. A better understanding of what kinds of code transformations are applied in the wild could guide future efforts on making the web more secure. In particular, knowing how widespread minification and obfuscation are helps future analyses to focus on relevant problems. Moreover, understanding what kinds of transformation tools are the most popular enables the development of targeted defense techniques. Unfortunately, to the best of our knowledge, there currently is no comprehensive study of code transformations in the web.

This paper presents a large-scale empirical study of minification and obfuscation in client-side web code. The study involves 967,149 JavaScript files gathered from the top 100,000 websites. We analyze how many of these scripts are transformed through minification and obfuscation, respectively, and which tools are used for these transformations. Moreover, we study which kinds of scripts are transformed particularly often and inspect the runtime behavior of some obfuscated scripts. Finally, we analyze which costs code transformation may incur by assessing to what extent popular transformation tools influence the performance and correctness of JavaScript code.

Given the large-scale nature of our study, we rely, at least in parts, on automation to answer the above questions. To this end, we present a neural network-based classifier that identifies JavaScript code with particular properties. For example, the classifier can be trained to distinguish transformed from non-transformed code, minified from obfuscated code, and to identify code transformed with particular tools. We show that the classification has very high accuracy for these tasks, providing an effective way to identify particular kinds of scripts across all studied websites.

The study addresses six research questions.

RQ1: How prevalent are minification and obfuscation in client-side JavaScript code? Answering this question is important to determine whether code transformations should be considered by security analyses at all, and what kinds of transformations such analyses should focus on. We find that code transformations are very widespread, affecting 38% of all client-side scripts. The vast majority of the transformed code has been minified, whereas less than 1% of all code is obfuscated. Even though the percentage of obfuscated code is low, the absolute number of obfuscated scripts (2,842) still motivates work on de-obfuscation, as these scripts arguably are the most interesting for security analysis.

RQ2: Which tools are used to obfuscate code in the web? There is a variety of tools available for obfuscating JavaScript code. Understanding which tools and transformation techniques are used most often in practice helps prioritize efforts toward dealing with transformed code. Our study finds that a single obfuscation tool accounts for most obfuscated scripts in the web: 2,551 obfuscated scripts resemble the output of the *Daft Logic Obfuscator*, an online tool that is available free of charge, motivating future work to consider obfuscation techniques implemented by this tool.

RQ3: Does the prevalence of code transformations differ across different kinds of scripts or websites? Since there are many possible reasons for minifying and obfuscating code, it is interesting to ask whether specific kinds of scripts are transformed more often than others. We find that third-party scripts, i.e., code loaded from another website than the one visited by a user, is about twice as likely to be transformed than scripts loaded from the visited website. Possible reasons for this distribution include that content delivery networks minify libraries to reduce network traffic, and that advertisement and tracking code is transformed to protected intellectual property. Studying the prevalence of obfuscation in different categories of websites shows that some categories, e.g., sites with adult content, contain more obfuscated scripts than an average website.

RQ4: What kind of behavior do developers hide behind obfuscation? To further understand the reasons for obfuscating code, we analyze the execution behavior of obfuscated scripts and manually inspect a subset of them. We find that many of the obfuscated scripts access APIs that are typically used for tracking, fingerprinting, cookie syncing, or cookie theft. We also identify a script with an unusually high number of calls to `performance.now`, which could be because the script is exploiting some timing channel.

The final two questions are about potential costs that applying code transformations may incur.

RQ5: How do code transformations affect the performance of code? We find that most obfuscation tools negatively affect performance, i.e., they slow down the execution of the code. In contrast, most minification tools either have no effect on performance or speed up the execution of the code. These findings show that complex code transformations may come at a non-negligible cost, motivating future work on performance-invariant obfuscation.

RQ6: How do code transformations affect the correctness of code? Developers applying an automated code transformation tool may naively assume that the transformation preserves the overall semantics of the code. However, we find that existing tools for both minification and obfuscation often produce corrupt code that behaves differently from the original code. Only about half of the transformed code completely preserves the original semantics, motivating future work on more reliable transformation tools.

The results of our work are relevant for at least four groups of people. First, the study provides insights for developers of security-related program analyses, e.g., static malware checkers and de-obfuscators. In particular, we show that transformed code must be considered by any analysis aimed at real-world, client-side JavaScript code, and we show what kinds of transformations are the most important in practice. Second, the study affects developers of obfuscation and minification tools by highlighting the costs that using state-of-the-art tools imply. Third, the study informs users of transformation tools about the effects that using such tools may have on the performance and correctness of code. Finally, the classifier that we develop to conduct our study enables researchers interested in analyzing real-world JavaScript code to focus on code relevant for their research. For example, the classifier can accurately identify obfuscated code among hundreds of thousands of scripts.

In summary, this paper contributes the following:

- The first large-scale study of minification and obfuscation in real-world, client-side web application code.
- Insights about how code transformations are used in practice, including evidence that minification is widespread, that more complex obfuscation is rather rare yet non-negligible, and that particular obfuscation techniques clearly dominate.
- An automated classification technique that accurately identifies different kinds of transformed code. The technique is useful to select particular scripts, e.g., those with obfuscated code, for further analysis.
- A benchmark of obfuscated JavaScript files gathered from various popular websites, which we provide for future work on de-obfuscating, analyzing, and understanding obfuscated code.

2 CLASSIFICATION OF SCRIPTS

Addressing RQ1, RQ2, RQ3, and RQ4 at the scale of hundreds of thousands of scripts requires an automated technique to determine whether a script has been transformed, and if yes, in what way. While a skilled human could manually label files with high accuracy, that approach does not scale to the amount of JavaScript code considered in our study. One approach to address this challenge would be to define a set of heuristics, e.g., based on idiosyncrasies of specific transformation tools, and to determine the properties of code based on these heuristics. Unfortunately, the heuristics-based approach relies on human expertise for identifying code properties that are unique to specific transformation techniques, and it cannot be easily adapted to other transformation tools. Instead, we address the challenge of determining properties of code through machine learning. This section presents two machine learning-based approaches for classifying JavaScript code. We use the more effective of the two approaches as the basis of our study.

2.1 Classification Tasks

We address three classification tasks.

TRANS. The first task, called *TRANS*, is to determine whether a given piece of JavaScript code has been transformed by any minification or obfuscation tool. We train classifiers for this task with examples of regular code, which has not been processed by any transformation tool, and with examples of transformed code, which has been processed by a minification or obfuscation tool.

OBFUS. The second task, called *OBFUS*, is to determine whether a given piece of JavaScript code has been obfuscated. We train OBFUS classifiers with examples of regular code, examples of minified but not obfuscated code, and examples of obfuscated code.

TOOL-X. The third task, called *TOOL-X*, is to determine for a given piece of transformed JavaScript code what tool has been used to transform the code. Because popular minification tools apply very similar transformations, we focus on obfuscation tools for this task.

2.2 Training Data for Learning Classifiers

To train the classifiers, we start with a set of human-written, or *regular*, JavaScript files and then create transformed variants of these files.

2.2.1 Corpus of Regular Code. The regular code examples are a subset of a corpus of 150,000 JavaScript files provided by others [24], which consists of human-written, non-transformed code from open-source projects. We remove from this corpus all files with a size less than 1kB, as they provide very little information for the classifiers to make an informed decision, and files with a size greater than 10kB, to keep the memory consumption during training at a manageable level.

2.2.2 Program Transformation Tools. Our study is based on popular minification and obfuscation tools. We select tools that are publicly available and widely used, as reported, e.g., by publicly visible download numbers.

Minification. We consider seven minification tools (the names in parentheses are the abbreviation we use throughout the paper): UglifyJS (uglify), babel-minify (babel), Google Closure Compiler (closure), javascript-minifier.com (jsmincom), Matthias Mullie Minify (mmminify), and YUI Compressor (yui). These tools reduce the program size mainly by performing white spaces reduction and identifiers shortening. In addition, some tools, e.g., closure, also perform optimizations, such as inlining or constant folding.

Obfuscation. We consider five obfuscation tools: javascript-obfuscator (jsobf), javascriptobfuscator.com (jsobfcom), DaftLogic Obfuscator (daft-logic), jfogs, and JSObfu. The result of our tool search also included JSFuck and javascript2img.com, but we exclude them as they either are unable to process large code files or produce invalid JavaScript code. Table 1 shows which transformation techniques the tools use, as reported in previous work and in the tool's documentation. The two most common obfuscation techniques are identifier encoding, usually using HEX encoding, and storing strings in a global array.

Table 1: Obfuscation tools and transformation techniques.

Transformation techniques	Obfuscation tools				
	jsobf	jsobfcom	jfogs	JSObfu	daft-logic
String splitting	✓				✓
Keyword substitution					
String concatenation				✓	
Encoding the entire code					✓
Encrypting the entire code					
Identifier encoding	✓	✓	✓	✓	✓
String encoding	✓	✓		✓	
Dead code injection	✓				
Control flow flattening	✓				
String array	✓	✓	✓		✓
Code protecting techniques	✓				

Configurations. Most tools provide options to configure which transformation techniques to apply. Since different configurations may result in a different transformed code, we use multiple configurations for each tool. In total, we consider 15 configurations for the obfuscation tools and 31 configurations for the minification tools. Together with the regular version of a JavaScript file, this setup yields 47 variants of each file.

2.2.3 Generation of Training Data. As training data for a specific classifier, we randomly sample 10,000 files from the corpus of regular files and apply transformations tools to these files. For all three tasks, we train the classifiers with an even split of two classes of code, i.e., half of the training examples are expected to be classified as positive and negative, respectively. For the TRANS classifier, we apply the minification and obfuscation tools to each code example, using each tool equally often. For the OBFUS classifier, we obtain examples of obfuscated code by applying one of the obfuscation tools to each code example, using each tool equally often. To obtain examples of non-obfuscated code, we use regular and minified code. Since OBFUS gets trained to distinguish obfuscated code from both minified code and regular code, it can be used not only to identify obfuscated code among any code, but also to classify transformed code into minified versus obfuscated code. For the TOOL-X classifiers, we use examples of code transformed by tool X and code examples that are either not transformed or transformed by other tools.

2.3 Classification via Identifier Frequencies

The following describes the first of two approaches to learn classifiers. The approach exploits the fact that both minification and obfuscation tools replace the original identifiers, e.g., names of local variables, with other identifiers. Because many transformation tools use specific identifiers, a skilled human can determine whether a tool has been used and if yes, which tool. The learning approach is based on this observation.

The approach consists of two main steps. The first step is to extract a feature vector for a given JavaScript file. The feature vector

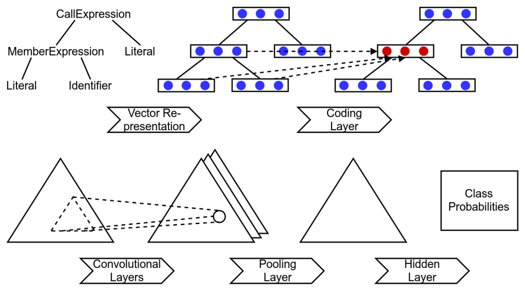

Figure 1: Tree-based convolutional neural network [18]. Rectangles with dots are vector representations of AST nodes; triangles represent trees.

summarizes what identifier names occur in the file and how frequent each name is. To this end, we tokenize the code and extract all identifiers. Based on all identifiers that occur in the training data, we determine a vocabulary of the 30,000 most common identifiers. Then, we transform each script into a feature vector of length 30,000, where each element represents a specific identifier. The element that represents a specific identifier is the *tf-idf* value of the identifier, i.e., the result of multiplying the *term frequency* with the *inverted document frequency* of an identifier. To compute the term frequency for a given script, we count the occurrences of each identifier and normalize these with the number of occurrences of the most frequent identifier in the script. For the inverted document frequency we divide, for each identifier, the total number of scripts in the training dataset by the number of scripts that contain the identifier and compute the logarithm of the resulting value.

The second step is to classify the feature vector of a JavaScript file using a support vector machine (SVM). We use SVMs because they are effective for binary classification problems. We achieve the best results for the SVM when using the *radial basis function kernel* and setting the penalty term C to 5. The classifier is implemented in Python using the machine learning library scikit-learn[1].

2.4 Classification via AST Convolution

The identifier-based classifier described above is conceptually simple but limited to a single feature of source code, i.e., identifier names. Our second classification approach addresses this limitation through a neural network that classifies abstract syntax trees (ASTs). ASTs are a useful representation of code because they preserve all relevant information while making the structural relationships between code elements explicit. To classify ASTs, we build upon a neural network architecture proposed by Mou et al. [18]. We adapt their approach by enriching traditional ASTs with additional information that proves useful for our classification tasks.

2.4.1 Background: Tree-based Convolutional Neural Network. We build upon an existing machine learning architecture for classifying trees, e.g., ASTs, based on a convolutional neural network. The network transforms a given tree into a continuous vector representation and then performs the actual classification task on the vector representation. The vector representation is learned in such a way that similar trees are represented by similar vectors. Figure 1 shows the five main steps involved in classifying trees. First, each node is transformed into a vector, where nodes with the same label, e.g., two CallExpression nodes, are mapped to the same vector. Second, a neural network layer ("coding layer") summarizes the children of a node and the node itself into the parent's vector. Third, several convolution layers extract features from the tree by sliding a feature detector over fixed-depth subtrees of the tree, which yields several trees of features. Fourth, a pooling layer summarizes these trees of features into a single tree again. Finally, all nodes of the new tree are passed through a fully connected hidden layer that outputs the classification result. During training with stochastic gradient descent, the parameters of the network are adapted to minimize a loss function that expresses how much the network's classification differs from the expected classification. Our implementation of the tree-based convolutional neural network is based on an implementation by Creston Bunch[2]. The architecture has previously been shown to be effective for identifying code that implements a particular algorithm [18]. We are the first to use it for identifying minified and obfuscated code.

2.4.2 Enriched ASTs. One possible approach is to apply the neural network to standard JavaScript ASTs. During initial experiments, we find this approach to provide a classifier with promising yet not fully satisfying accuracy. In particular, the classifiers struggle to learn from two features that are useful for a human but not well represented in standard ASTs: whitespace and specific properties of identifier names. Motivated by this observation, we enrich the standard JavaScript ASTs in two ways.

Whitespace. The first enrichment is to add information about whitespace into ASTs. Usually, this information is abstracted away as it is irrelevant for most scenarios where ASTs are used. To recognize minified and obfuscated code, though, whitespace is relevant because both minification and obfuscation tools often remove all or at least some whitespace. We enrich ASTs with whitespace information as follows: Whenever two nodes in the AST correspond to successive elements in the source code, we check whether any whitespace exists between the two code elements. If no such whitespace exists, then we insert a new "no whitespace" node between the two adjacent nodes; otherwise, we leave the nodes unchanged.

Length of identifiers. The second enrichment of ASTs is to add information about the length of identifiers, i.e., the number of characters of an identifier name. The motivation is that several obfuscation tools replace identifiers with less understandable identifiers of a fixed length that differs from tool to tool. In contrast, regular code usually consists of natural identifiers that have variable lengths. We encode this information by modifying every "Identifier" AST node by appending the length of the identifier to the node label, e.g., "Identifier3" for an identifier `foo`. To deal with unusually long identifiers, any length exceeding 30 characters is simply represented by a single, special label.

2.5 Accuracy of Classifiers

To decide which of the two classifiers to use for the study, we measure their classification accuracy on previously unseen sets of

[1]http://scikit-learn.org/stable

[2]https://github.com/crestonbunch/tbcnn

Table 2: Testing accuracies for different classifiers and classification tasks.

Classifier	Task		
	TRANS	OBFUS	TOOL-X
Identifier frequencies	76.40%	80.07%	64.44%–82.86%
AST convolution	95.06%	99.96%	99.68–100.00%

validation data. To this end, we randomly select 2,500 files from the non-transformed code (disjoint from the files used for generating training data), and create transformed variants of them, as described in Section 2.2.3. We then measure the accuracy for each classifier and task, i.e., the percentage of predictions that match the expected classification.

Table 2 summarizes the accuracy results for the two classifiers presented in Sections 2.3 and 2.4. Overall, the results show that both classifiers are effective (for comparison, a random decision would achieve 50% accuracy) and that the AST convolution-based classifier has the by far highest accuracy for all three tasks. In particular, the AST convolution-based classifier achieves more than 95% accuracy for all three tasks, and at least 99.68% accuracy for the OBFUS and TOOL-X tasks. In contrast, the classifier based on identifier frequencies performs poorly for some of the TOOL-X tasks, with an accuracy as low as 64.44% for one of the obfuscation tools.

To better understand why one classifier performs better than the other, we check how many of the different kinds of training examples the classifiers identify correctly. We find that the identifier frequency-based classifier is successful at identifying minified and obfuscated code but often fails to correctly identify regular code. For example, for the OBFUS task, the classifier correctly labels 99.92% of all obfuscated examples, but classifies only 59.47% of all non-obfuscated examples correctly. A detailed analysis of the identifiers that occur in obfuscated and non-obfuscated code explains these results: Many obfuscation tools use very characteristic identifiers, whereas regular code contains a wide range of natural identifiers. For example, jfogs creates many variables names `$fog$` followed by some number, and jsobf uses a similar pattern but also hex-encodes the identifiers.

To validate that enriching AST is beneficial over running the neural network on default JavaScript ASTs, we compare the accuracies of both variants of the AST-based classifier. We find that the default ASTs yield significantly lower accuracies than the enriched ASTs. For example, for the OBFUS task, the default ASTs give only 75.43% accuracy, which is not only much lower than with enriched ASTs but also lower than the classifier based on identifier frequencies.

Overall, we conclude from the accuracy results that the AST convolution-based classifier is highly effective at identifying transformed code, obfuscated code, and code obfuscated with a particular tool, making it a solid basis for studying JavaScript code at a large scale.

3 STUDYING DEPLOYED CLIENT-SIDE CODE

Based on the classifiers described above, this section presents the setup and results of our study of minification and obfuscation in deployed, client-side JavaScript code.

3.1 Study Data: Deployed, Client-Side JavaScript Code

To gather a representative set of JavaScript code used in real-world websites, we crawl the top 100,000 most popular websites, as listed by the Majestic Million[3] service. Our crawler visits each website, waits five seconds to enable dynamically loaded code to arrive, and then saves all scripts. We consider both code loaded via .js files and code loaded via inline scripts, i.e., via `<script>` tags without a `src` attribute. For the latter, the crawler copies the code between the tags into a new file. To speed up the loading of websites, we do not fetch resources other than scripts and HTML code.

The crawling yields 2,335,207 scripts from 85,001 websites in total. 14,999 websites are not accessible due to timeout errors and other reasons. Due to the limited size of ASTs that the classifiers can handle in reasonable time, we remove scripts with a size exceeding 40kB. This results in 1,861,489 scripts. We further remove scripts that are smaller than 512 bytes because we observe that such small scripts are hard to classify even for human subjects due to the limited number of clues that can aid the distinction between transformed and original scripts. Finally, we remove all those scripts for which the AST does not contain at least one "CallExpression" node. This is because such scripts are very often just configuration files in the JSON format. Applying these filters yields a set of 967,149 scripts. However, different websites may use the same JavaScript code, e.g., third-party libraries, thus the set of scripts contains duplicate files. In this study we are mostly concerned with the nature of the code on the web and not so much with its frequency on different websites, hence we remove all duplicates. This filtering leaves **a final set of 424,023 unique JavaScript files**, which we use for the study.

3.2 Accuracy of Classifiers on Study Data

Section 2.2.2 has established that our classifiers are highly accurate on code transformed by the tools used to generate the training data. To validate that the classifiers are effective also on the study data, for which we do not know which (if any) tools have been used, and to validate that the classification results match the classification that a skilled human could produce, we perform an experiment with JavaScript developers. The experiment involves five advanced developers, who have extensive experience in writing JavaScript and in understanding real-world JavaScript code.

The goal of the experiment is to gather ground truth to compare our classifiers against. During the experiment, each developer performs two tasks, which validate the TRANS and OBFUS classifiers, respectively. First, to validate the TRANS classifier, we show to each developer 50 scripts labeled by the classifier as transformed and 50 scripts labeled by the classifier as not transformed. We randomly sample the scripts from all 424,023 scripts in the study data. For each script, the developers are asked to answer the following question: "Is this human-written or generated/transformed code?" Second, to validate the OBFUS classifier, we repeat the same setup with 50 scripts labeled as obfuscated by the classifier and 50 scripts labeled as not obfuscated by the classifier. For these scripts, the developers are asked to answer the question: "Is this obfuscated or

[3]https://de.majestic.com/reports/majestic-million

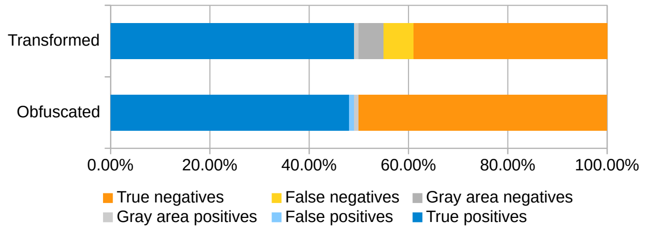

Figure 2: Effectiveness of the TRANS and OBFUS classifiers judged against the ground truth obtained from experienced JavaScript developers. The gray area depicts scripts for which the developers did not agree on the classification label.

not obfuscated code? If the file is transformed but you are not sure if it is obfuscated, try to decide whether the developer transformed it for hindering understanding." To avoid any influence from file names, such as "jquery.min.js", we hide the original file names from the developers and assign some generic names to the files.

After aggregating the results of the experiment, we compute the inter-rater agreement, which quantifies the extent to which the participants agree with each other. More precisely, we compute Cohen's kappa for each pair of participants. The pairwise agreement ranges between 0.64 and 0.93, with an average of 0.81, which is considered a very high agreement. To account for possible mistakes made by individual developers, we consider a 4-out-of-5 majority vote. That is, if at least four of the developers agree on whether a given script is transformed/obfuscated, then we consider this decision as the ground truth. If no such majority is reached, then we consider the scripts to be in a *gray area*, where even humans have a hard time judging whether the script has been transformed/obfuscated.

Figure 2 shows the results of the developer experiment. The blue area depicts true positives, i.e., scripts where the classifier and the developers agree on the scripts being transformed/obfuscated. Likewise, the orange area depicts true negatives, i.e., scripts where the classifier and the developers agree on the scripts being not transformed/obfuscated. Overall, the automated classifiers largely match the decisions by the developers. In addition, the TRANS classifier has 13% false negatives, i.e., it misses a few scripts that developers consider to be transformed, and the OBFUS classifier has 2% false positives, i.e., it sometimes incorrectly labels a non-obfuscated script as obfuscated. The gray parts in the middle of the figure, six scripts for TRANS and one for OBFUS, are the gray area, where the developers disagreed with each other.

Overall, we conclude from the experiment with developers that our automated classifiers match human classifications for the overwhelming majority of scripts. Given the 13% false negative rate of TRANS, one should interpret our results about the number of transformed scripts as a slight under-approximation of the actual number.

3.3 RQ1. Prevalence of Transformed Code

To address the question of how prevalent minified and obfuscated code is in client-side JavaScript code, we classify all downloaded scripts using the TRANS and OBFUS classifiers. Table 3 summarizes the results. We find that 38.5% of all scripts have gone through some kind of transformation, including both minification and obfuscation. In contrast, only 0.67% of all scripts (2,842 scripts) have been obfuscated. Applying the OBFUS classifier to those scripts that are classified as transformed confirms the above numbers: The vast majority of transformed scripts are not obfuscated, i.e., they have only been minified. These numbers show that minification is popular in the web. This finding is in line with the fact that many JavaScript libraries and frameworks include a minification step in their deployment pipeline to reduce the file size and hence the transmission time. A possible explanation for the low number of obfuscated scripts is that obfuscation comes at a cost (discussed in detail in Section 3.7 and 3.8), and therefore it is used only when developers want to hide some behavior. Despite the surprisingly low number of obfuscated scripts, these scripts provide an interesting target for further analysis, and we provide them as a benchmark for future work.[4]

Table 3: Prevalence of transformed code.

TRANS on all scripts:	Regular	61.5%
	Transformed	38.5%
OBFUS on all scripts:	Regular or minified	99.33%
	Obfuscated	0.67%
OBFUS on transformed scripts:	Minified	98.31%
	Obfuscated	1.69%

Table 4: Tools used to obfuscate scripts.

Classifier	% detected scripts	% other scripts	# detected scripts
TOOL-JSObfu	0.01%	99.99%	3
TOOL-jsobfcom	0.04%	99.96%	149
TOOL-jfogs	0.02%	99.98%	0
TOOL-daft-logic	0.60%	99.40%	2,551
TOOL-jsobf	0.02%	99.98%	71

Finding 1. Around 38% of the scripts in the web are transformed, most of which are minified but not obfuscated. Overall, we find a total of 2,842 obfuscated scripts.

3.4 RQ2. Prevalence of Obfuscation Tools

Given the non-negligible number of obfuscated scripts, we next address the question which tools and techniques developers use for obfuscation. Table 4 shows for each tool X how many scripts are detected as obfuscated by this tool according to the corresponding TOOL-X classifier. The by far most popular obfuscator in the web is DaftLogic Obfuscator, with 2,551 scripts in total. As the only tool that encodes the entire code using the `eval` function, it clearly stands out among the other tools. The study data does not contain any scripts obfuscated with jfogs. A surprising fact from these results is that the apparent popularity on npm of tools like javascript-obfuscator and JSObfu does not transfer to client-side obfuscated code. One reason may be that these obfuscators are more popular for JavaScript code running on Node.js than for client-side code.

Finding 2. DaftLogic Obfuscator is the by far most popular obfuscation tool. The most popular obfuscation technique is to load code at runtime via `eval`.

[4]http://software-lab.org/projects/obfuscation_study.html

Table 5: Overlap between different classifiers.

Pair of classifiers	Number of scripts
TOOL-jsobfcom ∩ OBFUS	147 (98% of TOOL-jsobfcom)
TOOL-daft-logic ∩ OBFUS	2,474 (97% of TOOL-daft-logic)
TOOL-jsobf ∩ OBFUS	71 (100% of TOOL-jsobf)

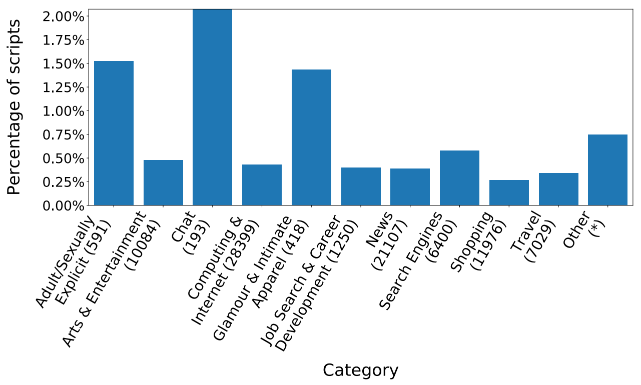

Figure 3: Prevalence of obfuscated code within categories of websites. In parenthesis on the x-axis we show the number of scripts in each category.

Having a set of classifiers related to obfuscation (OBFUS and several TOOL-X classifiers) raises the question to what extent the scripts detected by these classifiers overlap. Table 5 shows the overlap of scripts identified by the different obfuscation-related classifiers. We can make three observations. First, the TOOL-X classifiers do not overlap with each other, i.e., each of them precisely identifies scripts originating from a specific tool. This result is particularly remarkable for javascriptobfuscator.com and javascript-obfuscator, as these tools share a list of common obfuscation techniques. Second, almost all scripts detected as obfuscated by a specific tool are also detected as obfuscated by the general OBFUS classifier. Third, some scripts are classified as obfuscated but none of our TOOL-X classifiers can identify the tool used for obfuscating them.

Finding 3. Our generic obfuscation detection classifier successfully identifies most of the obfuscated scripts found by the individual tool classifiers and several additional scripts.

3.5 RQ3. Transformations vs. Kinds of Scripts

3.5.1 Categories of Scripts. We associate scripts with website categories, such as "News", "Travel", and "Education". To this end, we use the *Juniper Test-a-Site*[5] service, which yields a category for a given URL. We associate a category with a script based on the top-level domain from which the script was loaded.

Obfuscated code. We first analyze the prevalence of obfuscated code loaded by websites in specific categories. Figure 3 shows for ten of the categories the percentage of obfuscated scripts among all scripts loaded by a site and an additional entry with the average across all other categories. The results show that the prevalence of obfuscation differs significantly across website categories. Categories with a particularly high percentage of obfuscated scripts include "Adult/Sexually Explicit" and "Glamour & Intimate Apparel", i.e., sites with content that careful users may trust less than an average website.

[5]http://mtas.surfcontrol.com/mtas/JuniperTest-a-Site.asp

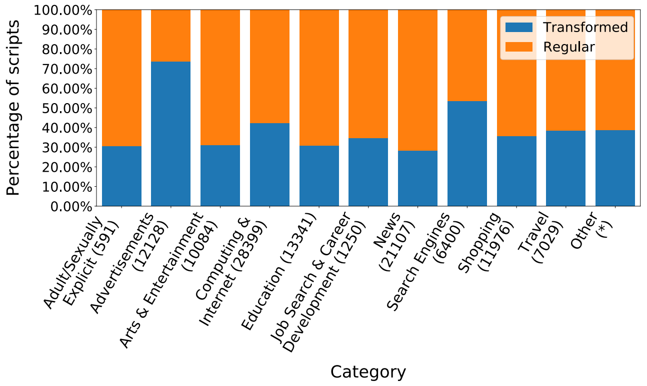

Figure 4: Prevalence of transformed code within categories of websites. In parenthesis on the x-axis we show the number of scripts in each category.

Transformed code. Figure 4 shows the ratio between transformed and regular code within all script categories in the study data. The percentage of transformed code ranges between 28% and 73%. The most significant categories are "Search engines" with 71.7% transformed code and "Advertisements" with 73.7%. We hypothesize that this is the case because websites in these categories serve scripts to many other domains, e.g. tracking scripts, and the size of the delivered scripts has a direct impact in the cost of using these services.

Finding 4. Code obfuscation is particularly common in certain website categories, e.g., adult content. Other code transformations occur particularly often in scripts delivered to many other websites.

3.5.2 Third-party Scripts. The following studies whether transformations are particularly common for scripts loaded from third-party sites. As first-party scripts, we consider all inline scripts and all scripts loaded from the top-level domain of the visited website itself. All remaining scripts are considered third-party scripts. [6] Based on this grouping of scripts, we analyze how the results of the TRANS and OBFUS classifiers relate to where a script is loaded from.
We find that third-party scripts, with a percentage of 55.38%, are almost twice as frequently transformed than scripts loaded directly from the visited website, which have a percentage of only 30.18% transformed code. These findings seem natural because providers of third-party scripts, e.g., content-delivery networks, often provide a minified version to reduce loading time. For scripts labeled as obfuscated, we do not find a significant difference between third-party and first-party scripts.

Finding 5. Third-party scripts are almost twice as frequently transformed than first-party scripts (55.38% versus 30.18%). Obfuscation is equally uncommon within both categories.

[6]This classification may be wrong for websites that split their files across multiple top-level domains. Finding more accurate ways to identify third-party scripts is left for future work.

3.6 RQ4. Runtime Behavior of Obfuscated Code

To better understand what the obfuscated scripts actually do, we run them in a custom environment and observe what APIs they try to access. The custom environment consists of self-replicating proxy objects that emulate the browser APIs and other well-known frameworks required by the analyzed scripts. For example, we create a globally accessible `document` proxy object that returns another proxy each time one of its properties is accessed. This environment provides a simple and effective technique for inspecting the behavior of obfuscated scripts.

In total, we analyze 2,924 unique obfuscated scripts which include all scripts classified as obfuscated by either OBFUS or one of the TOOL-X classifiers. We use 13 self-replicating proxies to mock the browser API and we run our analysis in Node.js. After each run, we collect a trace summarizing the property writes, property reads, and function calls observed via the proxies. In total, we collect 3.6 million property accesses, 10,400 writes and 1.4 million function calls. 2,231 scripts access at least one property via the proxies and 1,263 scripts set globally accessible properties. These numbers show that our setup is effective at running the obfuscated scripts and at extracting meaningful information about the APIs they call and the properties they access.

Out of the 2,924 scripts, 341 access the `cookie` object, 316 the `userAgent`, 287 the `location` and 101 the `referrer`. These are all privacy-sensitive APIs that may be accessed to perform cookie theft, cookie syncing, referrer sniffing, or browser fingerprinting. However, a more detailed analysis is needed to confirm this hypothesis. The most frequently invoked method is by far `document.createElement`, used by 346 scripts. This means that in order to fully understand a given obfuscated script, an analysis needs to reason about the HTML code injected in the page, which seem to be a popular idiom. Moreover, we observe that 295 scripts call `document.createElement('script')`, i.e., inject code at runtime, a technique commonly used by malware.

We manually inspect some of the traces and find two security-relevant behavioral patterns. First, several traces contain various API calls known to be used for browser fingerprinting, such as `location.href`, `screen.width`, `screen.colorDepth`, `navigator.plugins` or `window.devicePixelRatio`. There is even a trace in which after multiple such property accesses, a call to `document.createElement('img')` is made, which suggests that the obfuscated script is sending this information over the network. The second interesting case is of a trace containing 336,000 invocations to `performance.now`, which is likely to be part of a timing attack.

An additional observation we make after our sampling-based manual analysis is that some scripts set global properties, such as `SHA256_init`, `jQueryPath`, `mtTracking` or `Fingerprint2`. Their names suggest that some of the scripts register benign functionality, such as SHA hash functions or jQuery, while others register tracking and fingerprinting functionality.

Finding 6. Multiple obfuscated scripts access privacy-sensitive APIs and use dynamic code loading.

3.7 RQ5. Performance of Transformed Code

Code transformations may not only influence the understandability of code but also its efficiency. To better understand the cost-benefit tradeoff of transformations, we study to what extent transformations affect the performance of code.

Table 6: Libraries used to measure performance and correctness.

Library	Category	Number of tests
Bacon	Reactive programming	7,492
async	Asynchronous programming	513
immutable	Immutable data structures	557
lodash	General utility	6,685
math	Math utility	4,063
moment	Date utility	3,232
ramda	Functional programming	954
underscore	General utility	1,569
voca	String utility	409
when	Promise implementation	872

3.7.1 Benchmarks. Addressing RQ5 and RQ6 requires JavaScript code for which we can measure both the performance and the correctness. For this purpose, we gather a set of popular client-side JavaScript libraries that have extensive test suites. These tests include assertions to check the correctness of execution behavior, and they provide a reliable way to measure the execution time of code. Table 6 presents the ten libraries we consider, along with the number of tests they provide. All libraries are frequently used in client-side web applications. We execute their unit tests on Node.js, though, because it facilitates the performance measurements, as they are not influenced by opening and initializing a browser.

To study how transformations affect the performance and correctness of the libraries, we apply all 46 transformations (Section 2.2.2) to each library. To measure the performance of a given, possibly transformed, library, we execute its test suite 20 times and measure the overall wall-clock time of each execution. We do not perform a separate warm-up phase before measuring performance, as is common for longer-running benchmarks, to include the time for parsing code into our measurements, as this time affects the user experience on websites. Since closure injects code that is not compatible with our environment and prevents us from running the library tests, we omit this transformation tool for RQ5 and RQ6. We run all our performance and correctness measurements on an AMD Phenom II X6 1100T CPU with 16GB of RAM.

3.7.2 Results. Figure 5 shows the execution time of transformed code relatively to regular code for eleven different transformation tools. The figure presents results from the ten libraries listed in Section 3.7.1. Each data point shows the average execution time across 20 repetitions and the 95% confidence interval. The first six columns of data points are for minification tools, whereas the following four columns are for obfuscation tools. The last column shows the baseline, i.e. the performance of the original code.

For minification, the figure shows an overall improvement of efficiency. In particular, for two libraries, voca and math, minification causes performance improvements of over 20% and 8%, respectively. For most of the other libraries, minification causes a measurable but small performance improvement.

In contrast to minification, we observe an overall performance degradation after obfuscating code. For at least four libraries (ramda, moment, math, and Bacon), obfuscation significantly increases the

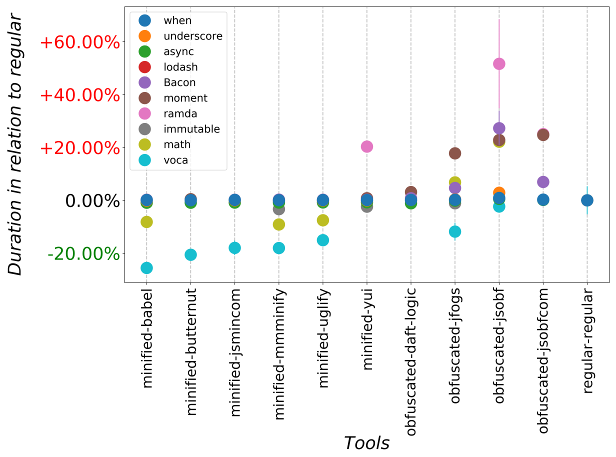

Figure 5: Execution time of transformed code relative to regular code. Each data point shows the average across 20 repetitions and the 95% confidence interval (which is too small to be visible for most libraries).

execution time, with average increases between 16% to 37%. For eight out of the ten libraries, the obfuscated code is measurably slower than the regular code. The only outlier is voca, which executes faster for two of the applied obfuscation tools.

Finding 7. Minification either improves performance or has no noticeable effect. In contrast, obfuscation can cause significant performance degradations.

3.8 RQ6. Correctness of Transformed Code

Besides affecting the performance of code, there is another potential cost of applying code transformations: the impact of transformations on the correctness of the code. To assess this impact, we run the test suites of the benchmarks from Section 3.7.1 before and after applying different transformations and measure the percentage of tests that still succeed after the transformation. We say that transformed code is *correct* if this percentage is 100%, i.e., the transformed code passes all tests.

Figure 6 shows the percentage of correct code among all code transformed with a specific tool. Overall, only about 70% of the minified code is correct, and even worse, less than 50% of the obfuscated code is correct. The original code, shown in the right-most column, is by definition 100% correct. A manual analysis of transformed code that fails test cases shows two root causes. First, some transformation tools have implementation-level bugs that get triggered by some code. For example, JSObfu, which consistently creates incorrect code extensively uses the `String.fromCharCode()` method to encode constant string occurring in the code. However, in some cases the method is applied on an undefined object instead of a string, which causes an exception. Second, some transformation tools change the semantics of code in ways that affect rather subtle corner-cases of the JavaScript language. For example, some configurations of UglifyJS replace non-global function names with short and meaningless names. If such a function is a constructor function, this transformation affects code that creates an object with

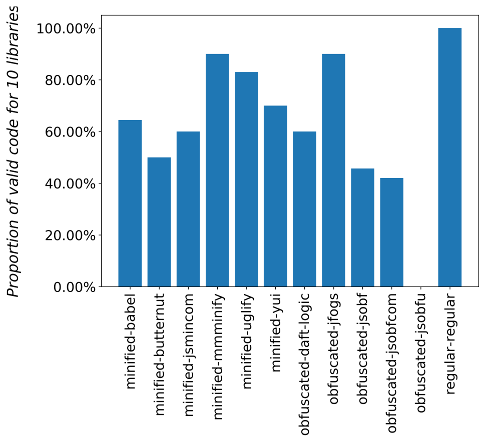

Figure 6: Percentage of correct code among all code transformed by a specific tool.

this function and then checks the name of the constructor using JavaScript's reflection APIs. Some of the tests trigger this corner case, e.g., a test of the "immutable" benchmark checks whether objects of type `Record` have a constructor with the same name.

Finding 8. A large portion of code transformed both by minification and obfuscation does not fully preserve the correctness of the code.

We conclude from these results that transformation tools may not only impose a performance cost, but even worse, risk to change the semantics of code. A practical take-away from this finding is that users of such tools must carefully check the correctness of transformed code, instead of blindly relying on the transformation tool. Our results also motivates future work on validating and improving minification and obfuscation tools, e.g., through automated testing.

4 RELATED WORK

This work relates to two active research fields: (i) detection of obfuscated and malicious code and (ii) empirical studies of the web.

4.1 Obfuscated and Malicious Code Detection

Tellenbach et al. [33] propose multiple classifiers for detecting obfuscation. However, they manually specify features of code files, such as the Shannon entropy or the number of characters per line. Similar, Likarish et al. [15] manually extract 60 features from code and propose four different classifiers. In contrast to both, our AST-based classifier does not require any feature engineering.

Wang et al. [36] present a neural network-based classifier for detecting malicious JavaScript code. Since attackers often hide the malicious intent of their scripts using obfuscation, a significant proportion of the dataset used for training their classifier consists of obfuscated code. Comparable to our approach, instead of defining explicit features, they learn the features automatically from the code with the help of multiple layers of stacked denoising autoencoders in the neural network. They transform the code files by replacing each character with a unique binary vector. The vectors require 20,000 dimensions to represent all characters of the dataset. Therefore, Wang et al. reduce the dimensionality to 480 using a dimensionality reduction algorithm. In contrast, we learn a vector representation for AST nodes which only requires 45 dimensions.

Al-Taharwa et al. [1] present a Bayesian-based obfuscation detector aimed at manually performed obfuscation. In contrast, we consider automatically transformed code. Kaplan et al. [10] and Curtsinger et al. [7] extract features from AST nodes while preserving the context of nodes. Their classifiers specialize on obfuscated and on malicious code, respectively. The context-based feature extraction mechanism is comparable to the concept we follow by processing entire ASTs of code files to preserve the context of all AST nodes. However, all three approaches limit the number of allowed contexts and reduce the number of features by applying feature selection. In our case, we do not discard any information from the ASTs so that the neural network can extract the most descriptive features.

There are further static classifier-based approaches which are less related to our approach. Jodavi et al. [9] address the detection of obfuscation with an ensemble of multiple one-class SVM classifiers which are trained to recognize non-obfuscated code using a set of structural and lexical features, such as the number of dynamic code evaluations and the maximum entropy of strings.

Certain researchers address the problem of detecting obfuscated code by using techniques other than machine learning. Xu et al. [38] propose an approach to detect malicious obfuscated code using both static and dynamic analysis. They mostly focus on obfuscation techniques which require the usage of the `eval` function, the `unescape` function, or other related functions. Using static analysis, they gather information about function definitions and invocations in the code. At runtime, they examine if new function definitions and invocations are present to reveal the potentially malicious part of the code. By comparison, we detect if code is obfuscated without assuming a malicious intent because obfuscation allows for different goals, e.g. the protection of intellectual property of code. Furthermore, we include all techniques offered by the obfuscation tools we use for this study instead of focusing only on `eval` based obfuscation. As discovered in Section 2.2.2, most JavaScript obfuscators do not implement `eval` based techniques.

Ceccato et al. [5] propose using out of the box obfuscators for hindering portability of attacks. We also use readily available obfuscators, but for generating training data, not for software diversification.

Deobfuscation is the process of reverse engineering obfuscated code. Techniques for purpose include learning-based approaches [3, 25] and semantics-preserving rewriting of the obfuscated code [16, 39]. This work is complementary to our work and can be applied after detection obfuscated code.

4.2 Empirical Studies

There are a few obfuscation related studies in other programming language contexts. Ceccato et al. [4] analyze the impact of different obfuscation patterns on Java code in terms of the difference between the obfuscated and the original code with the help of metrics. Wang et al. [35] examine the characteristics of obfuscated iOS Apps, the popularity of different obfuscation patterns, and the difficulty of reverse engineering the obfuscated apps. Hammad et al. [8] perform a study concerning the effect of obfuscated Android apps on anti-malware products. They find that the performance of most anti-malware products is significantly impacted by obfuscated apps. Moreover, they report that the tools used for obfuscating the apps frequently result in corrupt apps. Depending on the tool, only 0% to 62% of all 250 apps are runnable after the obfuscation is applied. We observe a similar outcome in the case of JavaScript obfuscation.

Visaggio et al. [34] compare obfuscated and regular JavaScript code using several metrics, including n-gram, entropy, and word size. They find that the two kinds of code differ from each other, in particular, when combining multiple metrics. XU et al. [37] study 510 samples of malicious JavaScript code, its use of obfuscation, and how obfuscation influences whether an anti-virus checker detects the code as malicious. In contrast, our study considers many more scripts, goes beyond obfuscated code, and addresses different research questions.

There have been various JavaScript and web security related studies recently, including a study on the prevalence of the `eval` function [27], on trust relationships between websites that include remote libraries and their corresponding library providers [21], on the communication between websites and embedded frames with 3rd-party content [30], on outdated libraries in the web [13], on XSS vulnerabilities [17], on ReDoS vulnerabilities in JavaScript-based web servers [31], and on performance issues in JavaScript [29]. However, there has been no study focusing on obfuscation and minification in the web outside of malicious code. We conduct the first comprehensive study examining the prevalence, the performance, and the validity of obfuscated and minified JavaScript code in the web.

4.3 JavaScript Analysis

Our work relates to program analyses for JavaScript [2], e.g., to detect conflicts between libraries [22], type inconsistencies [23], code injection attacks [32], or data races [20]. In contrast to that work, we here address an analysis task by training a machine learning instead of implementing an analysis by hand.

5 CONCLUSION

This paper presents the first large-scale study of minified and obfuscated JavaScript in client-side website applications. Our study leads to several findings that should be of interest for both the web community and the security community. In particular, we show that transformed code is surprisingly common, whereas obfuscation aimed at hindering understanding remains an exception. Yet, there is a non-negligible number of obfuscated scripts, which typically occur in specific website categories and which sometimes expose security-relevant behavior, e.g., fingerprinting or timing attacks. Moreover, we show that obfuscation not only provides benefits to its users but also imposes a cost by negatively impacting both the performance and the correctness of the code. Besides these findings we are releasing the obfuscated code detected during the study to stimulate future research in this area.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConcSys and Perf4JS projects, and by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

REFERENCES

[1] Ismail Adel AL-Taharwa, Hahn-Ming Lee, Albert B. Jeng, Kuo-Ping Wu, Cheng-Seen Ho, and Shyi-Ming Chen. 2015. JSOD: JavaScript Obfuscation Detector. *Sec. and Commun. Netw.* 8, 6 (April 2015), 1092–1107. https://doi.org/10.1002/sec.1064

[2] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *Comput. Surveys* (2017).

[3] Rohan Bavishi, Michael Pradel, and Koushik Sen. 2018. Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts. *CoRR* arXiv:1809.05193 (2018).

[4] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. 2015. A large study on the effect of code obfuscation on the quality of Java code. *Empirical Software Engineering* 20, 6 (01 Dec 2015), 1486–1524. https://doi.org/10.1007/s10664-014-9321-0

[5] Mariano Ceccato, Paolo Falcarin, Alessandro Cabutto, Yosief Weldezghi Frezghi, and Cristian-Alexandru Staicu. 2016. Search Based Clustering for Protecting Software with Diversified Updates. In *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*. 159–175. https://doi.org/10.1007/978-3-319-47106-8_11

[6] Marco Cova, Christopher Krügel, and Giovanni Vigna. 2010. Detection and analysis of drive-by-download attacks and malicious JavaScript code. *International Conference on World Wide Web (WWW)*.

[7] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=2028067.2028070

[8] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2017. A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products. *Proceedings of the 2018 IEEE 26th International Conference on Program Comprehension - ICSE 2018*.

[9] Mehran Jodavi, Mahdi Abadi, and Elham Parhizkar. 2015. JSObfusDetector: A binary PSO-based one-class classifier ensemble to detect obfuscated JavaScript code. *Proceedings of the International Symposium on Artificial Intelligence and Signal Processing, AISP 2015* (2015), 322–327. https://doi.org/10.1109/AISP.2015.7123508

[10] Scott Kaplan, Ben Livshits, Ben Zorn, Christian Siefert, and Charlie Cursinger. 2011. *"NOFUS: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code"*. Technical Report.

[11] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting Malicious Behavior in Browser Extensions. *Proceedings of the 23rd USENIX Conference on Security*.

[12] Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-chieh Shan. 2012. An Analysis of the Mozilla Jetpack Extension Framework. *ECOOP 2012 - Object-Oriented Programming - 26th European Conferenc*.

[13] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[14] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[15] Peter Likarish, Eunjin Jung, and Insoon Jo. 2009. Obfuscated malicious JavaScript detection using classification techniques. In *2009 4th International Conference on Malicious and Unwanted Software, MALWARE 2009*. 47 – 54.

[16] Gen Lu and Saumya K. Debray. 2012. Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach. *International Conference on Software Security and Reliability (SERE)*.

[17] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. *Network and Distributed System Security Symposium (NDSS)*.

[18] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press, 1287–1293. http://dl.acm.org/citation.cfm?id=3015812.3016002

[19] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. *Web 2.0 Security & Privacy, (W2SP)*.

[20] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races that Matter. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*.

[21] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[22] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. 2018. ConflictJS: Finding and Understanding Conflicts Between JavaScript Libraries. In *ICSE*. 741–751.

[23] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *International Conference on Software Engineering (ICSE)*.

[24] Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*. ACM.

[25] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code".. In *Principles of Programming Languages (POPL)*. 111–124.

[26] Andreas Reiter and Alexander Marsalek. 2017. WebRTC: your privacy is at risk. *Proceedings of the Symposium on Applied Computing, SAC 2017*.

[27] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg.

[28] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. *Network and Distributed System Security Symposium (NDSS)*.

[29] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *International Conference on Software Engineering (ICSE)*. 61–72.

[30] Sooel Son and Vitaly Shmatikov. 2013. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[31] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security Symposium*. 361–376.

[32] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. 2018. Understanding and Automatically Preventing Injection Attacks on Node.js. In *Network and Distributed System Security Symposium (NDSS)*.

[33] Bernhard Tellenbach, Sergio Paganoni, and Marc Rennhard. 2016. Detecting Obfuscated JavaScripts using Machine Learning . *International Journal on Advances in Security* 9, 3 & 4 (2016), 196–206.

[34] Corrado Aaron Visaggio, Giuseppe Antonio Pagin, and Gerardo Canfora. 2013. An empirical study of metric-based methods to detect obfuscated code. *International Journal of Security and its Applications* 7, 2 (2013), 59–74.

[35] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, and Tao Wei. 2018. Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018*. ACM, New York, NY, USA, 11.

[36] Yao Wang, Wan-dong Cai, and Peng-cheng Wei. 2016. A deep learning approach for detecting malicious JavaScript code. 9 (02 2016).

[37] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In *7th International Conference on Malicious and Unwanted Software, MALWARE 2012, Fajardo, PR, USA, October 16-18, 2012*. 9–16.

[38] W Xu, F Zhang, and S Zhu. 2013. JStill: Mostly static detection of obfuscated malicious javascript code . *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)* (2013), 117–128. http://www.cse.psu.edu/{~}szhu/papers/JStill.pdf

[39] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. *IEEE Symposium on Security and Privacy (SP)*.