# Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage

Shaown Sarker
North Carolina State University
ssarker@ncsu.edu

Jordan Jueckstock
North Carolina State University
jjuecks@ncsu.edu

Alexandros Kapravelos
North Carolina State University
akaprav@ncsu.edu

## ABSTRACT

In this paper, we perform a large-scale measurement study of JavaScript obfuscation of browser APIs in the wild. We rely on a simple, but powerful observation: if dynamic analysis of a script's behavior (specifically, how it interacts with browser APIs) reveals browser API feature usage that cannot be reconciled with static analysis of the script's source code, then that behavior is obfuscated. To quantify and test this observation, we create a hybrid analysis platform using instrumented Chromium to log all browser API accesses by the scripts executed when a user visits a page. We filter the API access traces from our dynamic analysis through a static analysis tool that we developed in order to quantify how much and what kind of functionality is hidden on the web. When applying this methodology across the Alexa top 100k domains, we discover that 95.90% of the domains we successfully visited contain at least one script which invokes APIs that cannot be resolved from static analysis. We observe that `eval` is no longer the prominent obfuscation method on the web and we uncover families of novel obfuscation techniques that no longer rely on the use of `eval`.

**ACM Reference format:**
Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. 2020. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In *Proceedings of ACM Internet Measurement Conference, Virtual Event, USA, October 27–29, 2020 (IMC '20),* 14 pages.
https://doi.org/10.1145/3419394.3423616

## 1 INTRODUCTION

Source code obfuscation can be defined as making a program unintelligible while maintaining its functionality intact [5]. In case of JavaScript (JS), source code is considered obfuscated if the logic and meaning is transformed in a way intended to make it difficult for a human to understand or reverse-engineer [43]. Obfuscated JS is used not only for malicious purposes, like to deliver zero-day browser exploits [35], but also to hide the logic of legitimate web applications as part of software protection [11].

As the web becomes ever ubiquitous, there has been a considerable amount of increase in both client-side [27, 50], and server-side [49] JS-based attacks, along with vast amounts of intellectual property moving to the browser's client-side. Both of these increasingly involve JS obfuscation as a method to either evade and obstruct detection [27, 38], or to protect the intellectual property from being pirated [11]. Unfortunately, we currently lack the tools to automatically analyze the web and identify new obfuscation techniques, and this affects our ability to accurately measure and detect web threats.

Despite the prevalence of JS obfuscation, it is addressed briefly as a side-effect in a number of prior research work investigating malicious JS-based attacks [9, 13, 17, 34]. A number of studies explored JS obfuscation in combination with identifying JS maliciousness, the underlying implication being malicious JS should possess obfuscation properties [2, 16, 36, 40, 58]. There also exist studies that discriminate between JS obfuscation in benign and malicious source codes [14, 33].

Unfortunately, little research has been conducted into the topic of identifying JS obfuscation as a technique itself without the intent of it [3, 10, 30, 31, 33, 51, 53]. However, all of these prior studies almost exclusively depended on creating a model first (either with static or dynamic analysis), and used a small set of JS dataset for the purpose of training and validation - thus limiting their result for being used in a real world scenario.

It is notable that prior research into JS obfuscation as a transport mechanism explored dynamic code execution, mainly through `eval`, to the point that obfuscation of browser client-side code has become synonymous with `eval`, and the built-in function has garnered notoriety. However, with the advancement of both the web and along with it JS, we increasingly see obfuscated scripts that mangle the source code to conceal the browser API calls to hide the true motivation of the source code. Intuitively, this holds true as any JS source attempting to interact with the browser and/or the user's ecosystem has to pass through the browser API.

In this paper, we investigate the nature of JS obfuscation through its concealing effect of JS browser API features. We specifically focus on quantifying obfuscation that is tied to hiding the interactions of the script with the browser. Our approach does not cover all types of obfuscation, as there can exist obfuscated JS scripts in the wild that do not use any browser APIs. We argue that these scripts would be extremely limited, as any script with malicious intend or complex behavior, such as drive-by downloads, fingerprinting/tracking scripts, crypto-currency mining scripts, malicious advertisements, *have* to use several browser APIs to perform its tasks and achieve the goal of the attacker. Additionally, JS scripts that do not use any browser features are by definition not interesting due to their very limited capabilities.

Our proposed analysis system combines the runtime footprint of JS with the static analysis of its source code to identify JS obfuscation, which requires no ground truth for training purposes. We


*IMC '20, October 27–29, 2020, Virtual Event, USA*

ACM ISBN 978-1-4503-8138-3/20/10…$15.00
https://doi.org/10.1145/3419394.3423616

apply our system on pages collected over the Alexa top 100k domains to conduct a large-scale measurement study of JS obfuscation on the web ecosystem - including measuring traces of JS feature concealing obfuscation on the web, the origin of such obfuscated scripts throughout the web, most frequently obfuscated JS features through concealing, and state of the art obfuscation techniques used by real-world JS scripts that do not rely on `eval`. In summary, our main contributions include:

- We propose a novel approach of identifying JS obfuscation through feature concealing based on the intuition that the runtime behavior of code should be evident by static analysis on source code.
- We present results from a large-scale analysis of the JS feature obfuscation effect in the wild. We crawled the Alexa top 100k domains and quantified this obfuscation, pinpointed the sources of obfuscation, and measured the hidden functionality that lies behind such obfuscated API function calls and JS property accesses.
- We identify prominent new obfuscation techniques through a data-driven approach and present case studies that no longer rely on the notorious `eval` function.

## 2 DEFINING OBFUSCATION

Barak et al. defined program obfuscation as "The goal of program obfuscation is to make a program unintelligible while preserving its functionality" [5]. Extrapolating from this definition, JS obfuscation, in its simplest form, can be defined as when the intentional behavior of a script cannot be fully realized until execution.

Our hypothesis is that if *the interactions of code with its underlying system cannot be deduced from static analysis of its source code, then the code is concealing these interactions and thus can be considered obfuscated.*

Our goal with this new definition of obfuscation is to express the level of difficulty that static analysis tools and human analysts will have when analyzing code. Notice that our definition of obfuscation is more relaxed and can include unintended concealing of functionality, for example through heavy modification of source code via minification (i.e., rewriting the source code to be more compact without changing its functionality). However, dynamic analysis is dependent on the execution flow and thus is not exhaustive when it comes to reveal all the capabilities of the code. This can be alleviated through techniques such as force execution [37], which is out of scope for this paper.

In this work, we focus on obfuscation on the web where the code is JavaScript and the system is the browser. Interactions between the code (JS) and the system (browser) are defined as invocations of browser API features (property accesses or function calls). For the rest of the paper, when we refer to JS obfuscation, it is in terms of this hypothesis and this behavior of concealing browser API features.

We describe our data collection system architecture and the subsequent analysis to detect this obfuscation in §3 and §4 respectively, followed by a validation of our detection system in §5. We then apply our detection system to collect and identify obfuscation traces from the Alexa top 100K in §6 and present our findings in §7 and §8.

## 3 DATA COLLECTION ARCHITECTURE

To detect JS obfuscation traces in accordance with our definition, we implemented an automated web crawling system capable of collecting JS script execution traces as depicted in Figure 1 and a subsequent post-processing system for the collected traces.

### 3.1 Web Crawler

Our data collection worker from Figure 1 is a Dockerized container consisting of a web crawler and a log consumer. The web crawler, described in this section, uses a custom build of Chromium producing JS execution traces in the form of log files (§3.2). The log consumer is a Go-based tool to compress the trace logs and archive them after a page visit is completed (§3.3).

The web crawler is based on the NodeJS library Puppeteer [23], which automates the Chromium browser and communicates with the browser over the Chrome DevTools protocol [26]. The data collection worker pulls a job with a domain from a Redis queue and proceeds to visit the domain's webpage. For each domain, using Puppeteer, our crawler launches an instance of our custom build of Chromium in headless mode, opens a tab, and navigates to the URL concocted by prepending the domain with `http://`, while monitoring the progress of the visit. We set a fixed 15 second time limit for navigating to the page, and upon finishing navigation, we remain on the page for an additional amount of time to let the page load any resources as needed. In our system, both the navigation and the subsequent loitering on the page are also limited to a total of 30 seconds beginning from the start of the navigation, after which we tear down the page and the tab, followed by the closing of the browser instance.

During the visit, we collect a number of auxiliary information for each visit - including all network requests made, the headers and bodies of all HTTP resources downloaded, and all script sources parsed through the debugger using standard DevTools protocol commands and event handlers over Puppeteer. All of this information are stored into a MongoDB document storage in an asynchronous manner (as encountered) during the visit of the page.

### 3.2 Instrumented Browser

Based on our hypothesis, to identify the presence of obfuscation concealing browser API features in JS source code, we need to perform dynamic analysis to trace the runtime behavior of a JS script and compare that behavior to a static analysis of the source text. We use *VisibleV8* [32], a custom, open-source variant of the *V8* JS engine powering the Chromium browser, to capture browser API accesses. Much like the Linux *strace* tool logs Linux system calls made by native applications, VisibleV8 (VV8) logs browser API function calls and property accesses made by JS applications. VV8's in-browser architecture provides significant coverage and robustness advantages over alternative approaches to JS instrumentation that rely on JS script injection, prototype patching, or browser extensions. Furthermore, the fact that VV8's instrumentation is embedded invisibly inside the JS engine itself provides more intrinsic stealth than generic prototype patching. JS performance under VV8 can be significantly lower than that of stock Chromium,

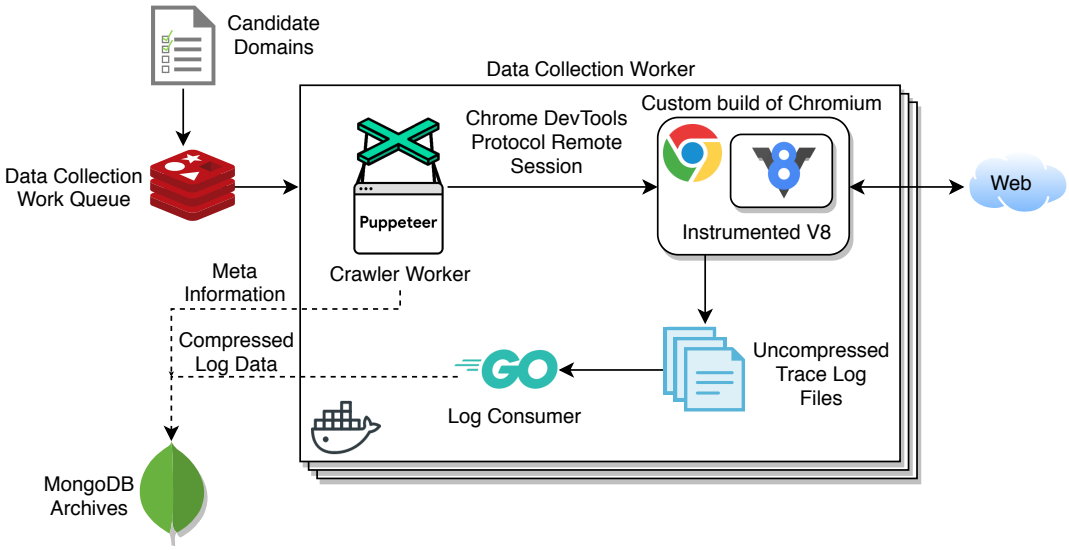


**Figure 1: Data collection pipeline for detecting JS obfuscation**

especially on micro-benchmarks[32], but general browsing performance of JS-intensive applications remained comfortably usable and scalable for our experiments.

VV8's instrumentation of **browser** APIs (e.g., `Window` and `Document`) to the exclusion of **builtin** APIs [44] (e.g., `Math` and `Date`) fits well with our hypothesis. Browser APIs constitute the interface between untrusted JS code and native browser functionality, in much the same way that operating system (OS) system calls constitute the interface between untrusted user applications and the trusted OS kernel. Since JS code cannot do anything "interesting" (e.g., any input or output of private user data or any other interaction with the user) without invoking browser APIs, they furthermore provide a "layer of truth" at which at least fragments of the script's intent becomes apparent regardless of how contorted its internal logic may be. To get the exact number of available browser API features, we analyzed and processed the WebIDL specification of the Chromium browser, identifying 6,997 unique API features for our analysis. We refer to the browser API features from this point on in the paper simply as "API features" or "JS features" interchangeably for brevity.

While VV8 has traditionally been used in stock Chromium, we embedded it into a custom build of the Chromium-based Brave [7] browser to take advantage of Brave's promising *PageGraph* technology. Still under heavy development, PageGraph is an open source [8] derivative of the technology behind Brave's AdGraph [28], which involves pervasive instrumentation of both V8 and the Blink rendering engine at the heart of Chromium. In our system, PageGraph complements VV8's low-level tracing of all browser API accesses with high level tracking of script provenance (§7), including via complicated DOM interactions and dynamic script injections.

Since both the VV8 and PageGraph instrumentation is built into a production browser (i.e., the Chromium-based Brave), data collection can be performed via standard browser automation. Automated browser visits to selected websites results in VV8 trace logs containing context information such as effective security origin URL, source code location of API accesses, names of the API function or property, and any data being passed into or written to logged functions or properties. PageGraph data can be extracted through an API exposed via the Chrome DevTools interface in Brave.

## 3.3 Log Consumer

The log consumer is a Go-based tool which has two responsibilities in our system. The first, as shown in Figure 1, is to compress and archive the trace logs produced by VV8 during our page visit into MongoDB. The log consumer is independent of the crawler so that the crawler can tear down the browser instance and the tab without it interfering.

Secondly, the log consumer is invoked once again as part of the post-processing of the VV8 trace logs. VV8 records the complete JS source code of every script processed through it via the trace logs and this record is kept exactly once per log for brevity. The post-processing extracts and archives all such scripts with a unique identifier of *script hash* into a PostgreSQL system for further analysis. The *script hash* serves as an identifier for the script in the database and it is derived by computing the SHA256 hash of the entire textual source of the script.

Additionally, the post-processing also extracts and records an API feature usage tuple consisting of a distinct combination of the following information:

- **Visit Domain:** the domain being visited by the page itself
- **Security Origin:** the runtime evaluation of `window.origin` which could differ from the visited domain based on whether the execution context was from an *iframe*
- **Active Script:** identified by its *script hash*
- **Feature Offset:** the character offset within the source for the API feature usage

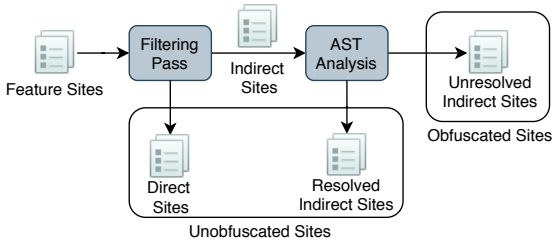


**Figure 2: Static analysis steps for detecting JS obfuscation**

- **Feature Usage Mode:** how the feature was used (i.e., a property get/set or a function call)
- **Feature Name:** a combination of the type of the native JS object and the accessed member (the property or the method name) of that object (eg. `Document.createElement`)

We commonly refer to the combination of feature name, feature offset, and feature usage mode on a particular script as a **feature site** of the script.

## 4 DETECTING OBFUSCATION

With our web crawling and subsequent post-processing of the VV8 trace logs providing us with feature usage information, we proceed to analyze the API feature sites as described in §3.3. We want to examine these feature sites within the scripts' source code and check whether these can be identified with a static analysis of the source. If any of the feature sites cannot be identified through this static analysis, we mark them as a trace of obfuscation concealing an API feature usage, since the usage cannot be deduced from the static analysis by itself without the trace information.

For our analysis, we conduct a two-step static analysis over the feature sites data as shown in Figure 2. We describe each step and the corresponding analysis in detail in the following sections.

### 4.1 Filtering Pass

The initial filtering pass over the feature site data is based on the intuition that the majority of the feature usages are not obfuscated and thus can be resolved simply through character offset examination in the textual script source. The intention of this pass is to filter out obvious non-obfuscated feature sites very quickly and mark the feature sites suspected of containing traces of obfuscation for further static analysis.

To achieve this, for each feature site (feature name, character offset, and usage) of a script, we extract the token at the character offset with the same length of the accessed member part of the feature name from the script's source, and then compare this token with the accessed member part. For example, if our feature site tuple has a feature name of `Document.write` with character offset of 100, we extract a token of length 5, starting from the textual source at character offset 100 and ending at offset 104. If this extracted token matches with the accessed member (`write` in case of our example) part of the feature name, we mark this feature site as a *direct site*. The underlying assumption here is that to use a native JS API feature (via call or property access) without any obfuscation of the source, one would have been likely to refer to the function call or property access by the accessed member part of the usage tuple in the source. In the case of a mismatch, we mark the feature site as an *indirect site* and as a candidate that requires more involved static analysis.

### 4.2 Abstract Syntax Tree Analysis

We subject the indirect feature sites from the filtering pass to a custom heuristic-based static analysis tool. This tool makes a best-effort attempt to resolve these sites through the analysis of Abstract Syntax Tree (AST) of the scripts' textual sources, combined with the variable scope information. The aim of this analysis tool is to identify certain basic human intelligible patterns in the script source through which these indirect feature usages could have been made. A successful resolve attempt indicates either no obfuscation at all or minor indirection that provides weak obfuscation at best, which can be understood through manual inspection by a human examiner. Failure to resolve indicates a certain level of obfuscation, whether deliberate (e.g., packed or mangled code) or accidental (heavy indirection and dynamism in the script's source). Either possibility is covered under our definition of obfuscation in terms of concealed behavior.

**Human Identifiable Patterns.** Consider the case of an indirect feature site involving a function call. There are two distinct, simple ways to invoke a function in JS given the function name identifier—we can either append a pair of parentheses with the arguments to be passed to the function enclosed within, or we can invoke any of the `call`, `apply`, `bind` methods defined on the prototype of the `Function` object. Since functions are first class objects in JS, they can be treated as any other variables. For feature sites making an indirect API function call, the feature offset should either contain a regular call on a source token that does not resemble the accessed property of the feature name, or an invocation of any of the `call`, `apply`, `bind` methods also on an non-resembling token. In either case, we need to trace back the token to the accessed property of the feature name in question.

In the case of property accesses, due to the dynamic nature of JS, there are myriad possible ways a property access can be performed. We focus on a limited subset of patterns a human examiner can resolve through the inspection of the script's source code. This subset included property accesses as part of a logical expression (e.g. `var a = false || "name"; ... window[a] = "value";`), an assignment redirection (e.g. `var p = "name"; ... q = p; ... window[q] = "value";`), or a member access expression on an object (`obj["p"] = "name"; ... window[obj.p] = "value"`). In the examples, the ellipsis designates other lines of source code within the script, and both the expression and the following reference shares a scope. In all these cases, again we need to trace back a non-resembling token to the feature name's accessed property.

**The Resolving Algorithm.** To perform our analysis, we parse the script source into an AST using the Esprima[4] JavaScript parser and identify the variable scopes, references, and binding expressions with the complementary EScope[21] analysis library. EScope provides all the variable scopes statically derived through the AST in nested form, and can provide the current scope for a given AST node with a reference to both the parent scope and the children scopes.

For each indirect feature site, we identify the originating AST node by first finding the AST leaf node that contains the target offset of the site. Next, we reach that leaf node's nearest parent node of the appropriate type: member access expression nodes for feature site involving property retrieval, assignment expression nodes for feature site involving property setting, and function call expression nodes for feature site involving function calls. At this point, our algorithm performs a recursive walk of the AST until we can resolve the expression under examination to the *accessed property* part of the feature name or a certain recursion level is reached (in our case this level was 50).

At each recursive AST walk, we expand the expression into AST nodes and kept reducing it by focusing on the particular node of interest in the expression - based on the expression subset we described previously, until we pinpoint either to a literal node or an expression node. If we reduce to a literal value, we check it against our accessed property of the feature site and report if there is a match or not.

However, if we reduce to an expression rather than a literal, we invoke an evaluation routine for the expression to resolve it to a literal value for checking with our accessed property. This evaluation routine is a JS interpreter for a subset of the AST structure which can potentially be resolved by a human examiner through inspection. This subset includes references to bound identifier variables, string concatenations, object member accesses, array literals, and method calls for which the receiver and all arguments can be evaluated statically.

When the evaluation routine reduce the expression to an identifier, we search for the variable corresponding to that identifier within the nearest enclosing scope as given by EScope. Upon finding the variable, we iterate over the variable's references within the current scope. If the variable has a *write expression*[1] of a literal value, we check the literal value with the accessed property. Otherwise, we invoke the evaluation routine recursively on the write expression.

We mark the indirect feature site as *resolved* when we could find a match to the resolved literal value for the accessed property. If we encounter an expression outside of our subset during the recursive walk or expression evaluation, or the recursion depth reaches the maximum level, or the resolved literal value does not match, we mark the indirect feature site as *unresolved*. These unresolved feature sites remaining after the AST-based analysis on our indirect sites contain feature usages that cannot not be deduced from static analysis of the source, and thus we identify the script to contain feature concealing obfuscation.

```
1 var global = window;
2 var prop = "Left Right".split(" ")[0];
3 global['client' + prop];
```

**Listing 1: Example for expression evaluation routine**

Listing 1 shows an example scenario to explain how our evaluation routine works. Given we are aware of the property access at line 3 (`window.clientLeft`) through our trace logs, we invoke our routine to evaluate the member access expression. Since this is a string concatenation expression with a literal and an identifier, we recursively invoke the evaluation routine on the identifier. From our variable scope analysis, we see that this identifier has a write expression of an array indexing expression, over a method call expression on a literal value. We recursively invoke our evaluation routine on these expressions until we reach the literal value and on our way back on the recursion, we evaluate each expression until we finally evaluate the `prop` variable to the literal value of `"Left"`. We finally evaluate our string concatenation expression with the value of `prop` to be `"clientLeft"` and since this matches our accessed property of the feature name, we mark the feature site as resolved.

However, the script excerpt in Listing 1 can be argued to be partially obfuscated—depending on the subjective judgment of the examiner. Due to the aggressiveness of our AST-based resolving algorithm, we can confidently claim that any unresolved feature site after passing though our analysis should be obfuscated, and our detection system provides a conservative bound of obfuscation traces.

## 5 VALIDATING THE HYPOTHESIS

If we assume our hypothesis (§2) to be true, then the following two sub-hypotheses should also hold: *1. for a script completely devoid of any obfuscation we should be able to identify all executing browser API calls and property accesses through static analysis of the source; 2. for an obfuscated script, we should be able to observe executing browser API calls and property accesses that we cannot resolve through static analysis of the code.*

To test these two sub-hypotheses, we selected a set of candidate scripts and a set of web domains on which they are statically included, set up a specialized web-crawling system through which we visited our selected domains, and performed the analysis described in §4. In the following sections, we describe the candidate selection process and the candidates themselves, followed by the analysis results of our detection system.

### 5.1 Candidate Selection

To test our first sub-hypothesis, we required scripts without any obfuscation, and also preferably without any compression or minification, as some compression or minification tools, such as UglifyJS[24], can perform a certain degree of optimization during the compression phase that can introduce obfuscation by altering the logical structure of the script. We refrained from using the minified versions of the scripts available, as there was no way to determine the tool and the level of minification used for these scripts. The best initial candidates we found were the developer versions of popular third-party JS libraries. These scripts are typically provided without any minification and are used for debugging purposes.

To identify our candidates, we focused on one of the biggest content delivery network (CDN) system for popular JS third-party libraries: Cloudflare's cdnjs [19], which serves 19% of the top one-million websites [54]. We picked the most-downloaded unique libraries through cdnjs based on the download statics for the month of September, 2019 [20] and filtered any libraries that are not JS (e.g. font-awesome), or libraries that do not provide developer versions of their source code (e.g. gsap). We came up with 15 libraries after this filtering listed in Table 7 in appendix A.

[1] In EScope terminology, write expressions are assignments to a bound variable within a scope

However, in most real-world websites developers opt to use the minified versions of third party libraries to improve network performance, and thus the developer versions of the script are barely, if at all, used in real-world websites. To select the domains using any of the third-party libraries of our selection, we retrieved both the developer and minified source code for all semantic versions available through cdnjs and computed the SHA-256 hash value for the minified source - giving us 545 distinct hash pairs. We then performed a search for these 545 hash values in our previously crawled dataset for the Alexa top 100K domains (crawled within the first week of October, 2019 - we describe this crawl in §6), which contains the DOM content and the scripts present on the root landing page of each of the domains along with their corresponding SHA-256 hashes. The matched domains constituted the set of candidate domains where we could use the developer versions of the scripts and have the page behave similarly. We found a total of 41,055 domains with hash value matches for 207 distinct semantic versions (obtained from cdnjs) of our 15 libraries (Table 8 in appendix B).

For each of the libraries, we took the 10 highest ranked domains (irrespective of the specific included semantic versions) as candidates, except for `lodash`, which had eight matches only, all of which were taken as candidates for our validation system. We excluded `popper.js` as we found only a single domain with a hash value match in our dataset, thus giving us 138 domains covering 64 semantic versions of these 14 libraries. After de-duplication, we ended up with 116 distinct domains as our candidate set for the validation system to visit with the developer version of the libraries.

For the second sub-hypothesis, we decided to use deliberately obfuscated versions of our developer version JS scripts. This way we can also demonstrate that obfuscation does result in concealed API calls or property accesses. To obtain the deliberately obfuscated versions of our developer version scripts, we used the widely popular JS obfuscation tool JavaScript Obfuscator [22], which has both a web-interface [48] and a command line tool through `npm` package manager, with weekly downloads averaging closer to 20k [45]. We based our tool selection by the comparative obfuscation tool study conducted by Skolka et al. in their work [53], where JavaScript Obfuscator was the top tool with the majority of highly used obfuscation features.

## 5.2 Record & Replay Webpages

For the validation system, we needed a way to visit the candidate domains twice - once with the developer versions of the candidate scripts and again with their tool-assisted obfuscated counterparts. However, the domains as we have seen, used the compressed versions of the candidate scripts. To circumvent this issue, we relied upon the Web Page Replay (WPR) tool [25], part of the catapult project [18] from Google. WPR is a Go-based tool that, when used in record mode, positions itself between the browser and the web as a proxy. The Chromium browser instance can connect to the WPR server over a proxy connection and any subsequent web page visit during the connected session is recorded in a compressed archive which contains all requests and the corresponding responses between the browser and the web for a specific session. Similarly, when WPR is run in replay mode with a prior recorded archive, the browser can reenact all request responses exactly as performed during the record session, given the requests are present in the archive—thus reconstructing the exact same web page in effect during the recording. We used the data collection system described in §3 in combination with the WPR server for our page visits during validation.

In our system, we visited each of our candidate domains three times: once in record mode and twice in replay mode. The visit in record mode was to create the archive for the candidate domains' webpages with the candidate scripts' requests later to be used for replaying the page. Our crawler launched the WPR server and added the proxy details to the Chromium browser's launch options. After the page visit was completed, the crawler closed the WPR server, triggering it to write the recorded archive in a file on disk.

During our recording visit[2] for the 116 candidate domains, we found 57 distinct semantic versions of our 14 candidate libraries out of the initial 64 distinct versions matched in our Alexa crawl data - this is most likely due to the websites updating their third-party scripts during the time between our Alexa crawl and the validation crawl.

We visited each of the candidate domains twice with this recorded archive data - once for the developer versions of the scripts and again for the tool-assisted obfuscated versions of the scripts. We wrote a Go-based tool named *wprmod* to alter the WPR record archives' request-responses given the SHA-256 hash of the response body to replace and the actual content in textual format to replace it with. With this altered archive, WPR replays provided the replaced content for the same request performed during recording. Our replay runs replaced the used versions of the scripts with our developer and obfuscated versions respectively, and visited the candidate pages with this altered archive.

However, we observed that in our recorded archives some of the requests' responses had compression encoding scheme mismatches. For example in the request header, it mentioned `gzip` as compression encoding and then provided a response body with encoding `utf-8`. These were clearly server configuration errors from the site developers and caused a few decompression error in our wprmod tool where we did not rewrite these the request's response body in such cases. So, during our replay for the developer versions of the scripts, we replaced 51 distinct semantic versions of the 57 compressed versions.

To generate our deliberately obfuscated scripts, we passed the developer versions of the scripts through the command-line version of the JavaScript Obfuscator tool. The authors of the tool provide three preset configurations for the tool, and we used the most popular configuration with medium obfuscation and optimal performance for generating our deliberately obfuscated versions of the candidate scripts. We refrained from using the maximum obfuscation setting to keep script breakage to a minimum - only 34 out of 51 versions of developer scripts did not result in either a time-out or exeception with maximum settings. We replaced 50 distinct semantic versions out of the 51 versions of developer scripts - only a single library (`json3`, version: 3.3.2) failed to parse with the Javascript Obfuscator tool.

[2] Done in October, 2019

| | Developer | Obfuscated |
|---|---|---|
| Direct | 3,050 | 250 |
| Indirect - Resolved | 15 | 757 |
| Indirect - Unresolved | 20 | 2,009 |
| Total | 3,085 | 3,012 |

**Table 1: Breakdown of feature sites after the two-step analysis on candidate scripts**

| Page Abort Category | Category Count |
|---|---|
| Network Failures | 5,431 |
| PageGraph Issues | 4,051 |
| Page Navigation (15s) Timeout | 3,706 |
| Page Visitation (30s) Timeout | 1,305 |
| Total | 14,493 |

**Table 2: Categories for Alexa top 100K domains visit errors breakdown**

### 5.3 Validation Results

In our post-processed data, we had 3,085 and 3,012 distinct API feature sites from the 51 developer version scripts and the 50 obfuscated scripts, respectively. We applied our two-step detection system as described in §4 on these features sites. Table 1 shows the breakdown of the feature sites over our developer and obfuscated versions of the candidate scripts after our analysis.

We manually checked the 20 indirect feature sites on the developer versions of the candidate scripts that were marked unresolved by our analysis. We found that all of these feature sites were property accesses through a wrapper function of the form: `f = function (recv, prop) {... recv[prop] ...}`. Using this the property accesses were made by invoking this function, e.g. `f(window, "location")`, where the function invocations were not necessarily part of the script itself. This was an indirection mechanism, which could only be resolved by a human examiner if the examiner had access to the entire call stack. However, static analysis of variable scope is incapable of evaluating callee argument values through the call expressions. Based on this, we classified these as legitimate unresolved feature sites.

Given the sheer number of unresolved feature sites present in obfuscated candidate scripts (2,009—66.70% of total sites) compared to the developer versions (20—0.64% of total sites) after our analysis system, we concluded that both of our sub-hypotheses hold, and establish that *comparing dynamic trace information with static analysis for API features is a viable way to reveal feature concealing obfuscations in JS scripts.*

## 6 ALEXA TOP 100K DATA COLLECTION

To measure the effect of JS obfuscation through concealed API usage on the web, we reused our crawling system from §3 to collect data from the Alexa top 100K domains.[3] We deployed this crawl over a Kubernetes cluster with 80 CPU cores and 512GB of memory, connecting to the internet from our university network.

We queued the Alexa top 100k domains, excluding the 37 Punycode-encoded [12] domain names that our queuing logic did not process, to our web crawler as described in §3.1. Of the queued domains, 85,470 completed successfully; i.e., the crawler completed the visit without error. Table 2 shows the major broad categories of the 14,493 page visit failures. Most of the network failures occurred due to the domain name not being resolved, indicating presence of stale domains on the Alexa list, followed by a variety of ephemeral network errors including DNS lookup failures, TLS/SSL errors, and transport level errors (connection reset/refused). The vast majority of PageGraph issues were triggered by PageGraph's conservative internal correctness assertions aborting the page load, with a few breakages encountered due to the page not having the necessary content to generate a PageGraph. The page navigation and visitation timeouts happened when our preset time limits during page visits exceeded, as described in §3.1.

After the post-processing run on the collected trace logs, we extracted and archived 11,120,829 JS scripts encountered in our instrumented Chromium browser from 84,260 distinct domains of the 85,470 domains successfully visited, out of which 3,222,053 had a unique script hash. Our use of dynamic analysis and instrumentation focused tightly on browser APIs limited the population of scripts for which we had feature site data for 1,083,803 distinct scripts from 77,423 distinct domains.

[3] http://s3.amazonaws.com/alexa-static/top-1m.csv.zip - as retrieved on Sep 24, 2019

| Category | Distinct Scripts |
|---|---|
| No IDL API Usage | 177,305 |
| Direct Only | 787,599 |
| Direct & Resolved Only | 43,048 |
| **Unresolved** | **75,851** |
| Total | 1,083,803 |

**Table 3: Breakdown of all unique scripts from Alexa top 100K crawl after the analysis**

## 7 RESULTS

Table 3 summarizes the volume of scripts at each level of our analysis workflow. In this case "No IDL API Usage" means our instrumentation detected native function or property access (e.g., accessing the global object), but that no standard, IDL-defined browser features were invoked. *Direct only* includes scripts in which all feature sites were cleared through the filtering pass of our analysis. *Direct & resolved only* scripts include both direct and some indirect feature sites - all of which were successfully resolved through our AST-based analysis. The remaining *unresolved* scripts show at least one unresolved indirect feature site, constituting to our set of *obfuscated scripts*. Note that in the following discussion, when we mention *resolved* scripts, we refer to the set of scripts that do not contain any unresolved indirect sites after our analysis. Similarly, when we refer to *obfuscated* scripts, we refer to the set of scripts with unresolved feature sites.

### 7.1 Obfuscation Prevalence

To approximate the extent of obfuscated JS scripts through feature concealing in the wild, we consider the two values from the analysis of our collected data: the number of scripts with unresolved

| Alexa Rank | Domain | Unresolved | Total |
|---|---|---|---|
| 79,633 | 11alive.com | 55 | 220 |
| 57,593 | sportune.fr | 49 | 250 |
| 64,969 | racingjunk.com | 49 | 296 |
| 75,354 | kron4.com | 48 | 223 |
| 47,511 | ovaciondigital.com.uy | 47 | 254 |

**Table 4: Top 5 domains by number of obfuscated scripts vs. total scripts loaded on that site**

indirect feature sites out of the total analyzed script population, and the popularity of such scripts across all visited domains. We find that the majority of the scripts are without obfuscation traces in accordance with the intuition, and a relatively small population of scripts containing obfuscated feature sites is encountered on almost all top websites.

We found that of the 77,423 domains for which we had script data, out of the Alexa top 100k, only 3,178 (4.10% ) did not load obfuscated scripts. The vast majority of these, 74,245 domains (95.90% ), contained at least one obfuscated script. In Table 4, we show the top five web domains loading the most obfuscated scripts along with the total number of scripts loaded by that site, ordered by the site's Alexa rank. We note that four out of the five sites are news media sites (local news, sporting events). Online news sites are, of course, notorious for heavy use of aggressive advertising (and associated tracking) content - that such sites are the most prolific users of observed obfuscated scripts is unsurprising.

## 7.2 Context and Origin of Scripts

To understand how and from where obfuscated scripts are loaded, we leverage metadata from our instrumentation to compare the execution contexts, source origins, and loading mechanisms of both obfuscated and resolved scripts. We find that obfuscated scripts typically come directly from 3rd-party sources despite executing in a 1st-party security context at nearly the same proportional rate as resolved scripts.

**Script Loading Mechanisms.** PageGraph[28] provides script type annotations, which indicate how a script was loaded - via external URL, inline inclusion in static HTML, or dynamic inline injection via various DOM APIs. From these annotations, we discovered contrasting differences in how resolved and obfuscated scripts were loaded during our experiment. We saw obfuscated scripts loaded overwhelmingly (98%) via external URL (i.e., script tags with explicit, http(s) URL `src` attributes). Conversely, resolved scripts showed more diversity of loading mechanism: 59% from external URLs, 26% from inline code in HTML documents, 7% generated inline via `document.write`, 5% generated inline via DOM API calls, etc. Obfuscated feature sites are thus heavily concentrated in external scripts (e.g., advertising and tracking frameworks, compressed libraries from CDNs loaded from other 3rd parties), compared to application specific or bootstrapping script source code directly embedded in (or injected into) HTML documents.

**1st vs. 3rd Party Defined.** We consider two axes of distinction when analyzing how obfuscated scripts are loaded and executed. Most obviously, scripts may be loaded from the 1st-party domain (i.e., the domain we are visiting) in contrast to some other 3rd party domain. This 1st vs. 3rd party categorization of a script's origin URL (the URL the script was loaded from - if any exists) is distinct from the script's security *execution context* which is the security origin as extracted from the dynamic traces mentioned in §3.3.

The browser enforces a Same Origin Policy (SOP) to prevent documents and sub-documents (such as `iframes`) loaded from different *origins* (i.e., *URL scheme*, *host name*, and *port number*) cannot access each others' DOM trees. In our case, we compare only the eTLD+1 (public suffix plus one subdomain; e.g., "example.com" for "sub.example.com") of domain names, not the full origin for 1st vs 3rd party association. This approach differs from the browser's official SOP, but it has the advantage of revealing close relationships between related subdomains.

**Execution Context.** We consider a script to have 1st party execution context if the eTLD+1 of the runtime evaluation of `Window.origin` matches that of the visiting domain. Based on this, among resolved scripts, 49.11% were loaded in 1st party context compared to 50.75% in 3rd party context. Obfuscated scripts showed nearly the same breakdown: 48.47% 1st party to 51.27% 3rd party loading. That obfuscated scripts are loaded and executed with 1st party privileges at nearly the same proportional rate as more readable code raises questions about the amount of trust afforded to JS scripts deliberately concealing its range of potential activity.

**Source Origin.** For our scripts, we categorized scripts with source origin URL that have the same eTLD+1 as the visiting domain to be 1st party source origin scripts. In the case of the script not having a source origin URL, we recursively look for the parent script responsible for this script either though dynamic DOM manipulation or `eval`, and use the source origin URL of the parent script. During this ancestral recursive walk, if we encounter that the parent is not a script, rather a document or sub-document (eg. iframe) - indicating the the child script was included in the document in an inline manner, we simply fall back to the security origin URL of the document.

We found obfuscated scripts to have 3rd party source origins more frequently than the resolved scripts. In our dataset, 78.55% of obfuscated scripts had 3rd party source origins compared to 61.77% for resolved scripts. This disparity in favor of 3rd party source origins again corroborates that obfuscated JS scripts typically originates from 3rd party domains, rather than created and hosted locally.

## 7.3 Feature Site Obfuscation and eval

Given the long association of `eval` with obfuscated and malicious code [10, 57, 58], we wanted to explore the relationship between feature site obfuscation and use of `eval` in the wild. Note that our methodology does not attempt to classify `eval` use as obfuscated or unobfuscated at all, so we are not attempting a direct comparison between obfuscation mechanisms. But we report on the overall volume of `eval` activity observed, to compare it with the volume of unresolved feature sites and obfuscated scripts.

In this context, if a script uses `eval` to load another script, we refer to the script performing the `eval` as the *parent* and the script loaded via `eval` as the *child*. Out of the 1,083,803 distinct scripts analyzed, we found 69,163 total distinct `eval` children scripts from

| Feature Name | Obfuscated Perc. Rank | Direct Perc. Rank |
|---|---|---|
| Element.scroll | 96.11% | 45.90% |
| HTMLSelectElement.remove | 87.15% | 50.76% |
| Response.text | 89.96% | 55.72% |
| HTMLInputElement.select | 88.23% | 56.26% |
| ServiceWorkerRegistration.update | 87.90% | 57.67% |
| Window.scroll | 92.12% | 64.36% |
| PerformanceResourceTiming.toJSON | 82.61% | 55.08% |
| HTMLElement.blur | 96.54% | 69.76% |
| Iterator.next | 84.99% | 58.64% |
| Navigator.registerProtocolHandler | 91.04% | 65.01% |

**Table 5: Top 10 API functions accessed via obfuscation**

| Feature Name | Obfuscated Perc. Rank | Direct Perc. Rank |
|---|---|---|
| UnderlyingSourceBase.type | 98.45% | 30.89% |
| HTMLInputElement.required | 94.91% | 56.89% |
| Navigator.userActivation | 88.77% | 52.42% |
| StyleSheet.disabled | 92.00% | 56.95% |
| CanvasRenderingContext2D.imageSmoothingEnabled | 84.68% | 50.00% |
| Document.dir | 89.76% | 56.76% |
| HTMLElement.translate | 86.79% | 54.65% |
| HTMLTextAreaElement.disabled | 86.66% | 54.65% |
| Document.fullscreenEnabled | 95.97% | 65.20% |
| BatteryManager.chargingTime | 91.07% | 60.73% |

**Table 6: Top 10 API properties accessed via obfuscation**

21,380 total distinct `eval` parents. When we considered only the set of obfuscated scripts, we found 1,901 distinct obfuscated scripts resulting from `eval` (2.75% of all distinct children) and 5,028 distinct obfuscated scripts performing `eval` (23.52% of all distinct parents). Strikingly, while in the general population of analyzed scripts, `eval` children outnumber parents more than 3 to 1, among obfuscated scripts the relationship is reversed, with `eval` parents outnumbering children more than 2 to 1. In other words, *obfuscated scripts are more likely to use* `eval` *to load scripts, than they are to be loaded by* `eval` *in the first place.*

Recall that these statistics reflect not necessarily *obfuscated* `eval` usage, but *all* `eval` usage observed in our dataset. While we make no effort to classify `eval` usage as obfuscation or not, the numbers provide a comparative upper bound on how much `eval`-based obfuscation could have existed in our dataset. Even if every `eval` parent observed were assumed to be a case of obfuscation (which is almost certainly not true), *we still observed significantly more distinct instances of feature site obfuscation (75,851 vs. 21,380 ).* This is in accordance with the intuitive observation that `eval` is a well-known footprint of obfuscation, even among static detection systems, possibly resulting in this shift away from `eval` by the obfuscation tools and actors.

## 7.4 Frequently Obfuscated APIs

To gain insight into what browser API interactions tend to be obfuscated in the wild, we compare API function popularity across resolved and unresolved (i.e., obfuscated) feature sites.

**Popularity Comparison Mechanism.** To identify distinct obfuscated features (i.e., both function calls and property accesses more likely to be accessed from unresolved than resolved feature sites), we first computed each distinct API feature name's percentile rank (i.e., popularity) among resolved ($P_r$) and unresolved ($P_u$) feature sites. We then compute the percentile ranks difference ($|P_u - P_r|$) for each feature name, giving a score that is higher when the feature is more popular among unresolved than resolved feature sites. APIs showing the highest percentile rank differences were more likely to be used in an obfuscated way in our data. However, since low frequency outliers in either list can radically skew the scores, we further filtered out the feature names with highest scores by removing all API features with total global access count below 100.

**Distinctly Obfuscated APIs.** We initially identified 923 distinct API functions and 1,608 distinct API properties accessed via resolved feature sites, while 320 distinct functions and 639 distinct properties were accessed in an obfuscated manner. In Table 5, we present the top 10 functions by gain in percentile rank, ordered by descending rank gain. Out of these 10 functions, 5 are associated with simulating user-interaction or manipulating user input forms. Among the rest are APIs associated with performance profiling, JS-initiated network requests, ServiceWorkers, and registration of custom URL scheme handlers (which requires user consent). In Table 6, we present the top 10 properties by gain in percentile rank, also ordered by descending rank gain. Of these, 4 are associated with user interaction detection or other user interface manipulation, 4 with obscure DOM metadata, and 1 with media streaming. The last is part of the infamous `BatteryManager` API that was hastily deprecated for privacy reasons [42, 47].

# 8 OBFUSCATION TECHNIQUES IN THE WILD

Having identified the scripts with obfuscation traces, we focused on the obfuscation techniques used for concealing the feature usages. To extract and identify the most prominent obfuscation techniques used in real-world obfuscated scripts, we build an automated system that processes unresolved feature sites and automatically identifies groups of similar feature sites. Our system is capable of assigning each cluster a score, which highlights clusters that are indicative of representing a specific obfuscation technique.

## 8.1 Feature Site Clustering

**Clustering Process.** To apply clustering on our unresolved feature sites, we needed to extract feature vectors from our 491,909 unresolved feature sites over 75,851 scripts from our analysis of the Alexa top 100K crawl data. Since we were clustering the feature sites instead of the entire scripts, we wanted to extract feature vectors from only the relevant portion of the script contiguous to the feature site, which we termed a *hotspot*. For each unresolved feature site of a script, we parsed the script into tokens using Esprima[4] and identified the token containing the character offset of the feature site. The hotspot of the feature site consisted of $r$ tokens before and after the containing token, including the containing token itself, where $r$ was the *radius* of the hotspot. We then proceeded to create a vector from the $2r + 1$ tokens of the hotspot in terms of token type frequencies, resulting in a vector of 82 dimensions for each of the unresolved feature site.

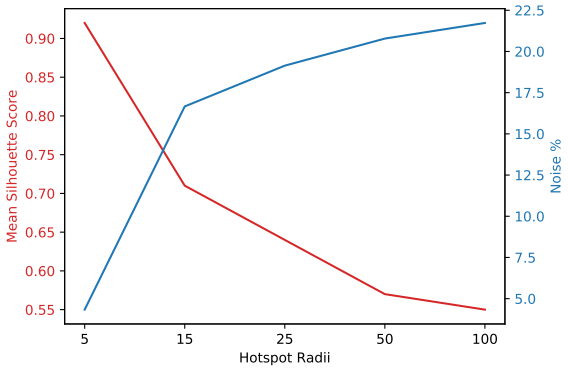


**Figure 3: Mean silhouette score vs noise percentage of DBSCAN runs over different radii of feature site hotspots**

We then applied off-the-self DBSCAN (epsilon = 0.5, min samples = 5, distance metric = euclidean) [52], a density based partial clustering algorithm, on our hotspot vectors to generate the clusters. Since our vectors depended upon the value of the radius $r$, we ran the clustering with different radius values. Figure 3, shows the clustering results in terms of noise percentages (percent of feature sites not belonging to any cluster; lower is better) and mean silhouette scores (average measure of all cluster cohesiveness and isolation, out of 1.00; higher is better) of different radii. As can be seen, smaller radius values performed better as they were likely to exclude tokens non-relevant to the obfuscated feature site into the hotspot. We selected the clustering labels for radius value 5, resulting in 5,741 clusters with a noise percentage of 4.33% and a mean silhouette score of 0.9212.

**Ranking Clusters.** After the initial clustering, we wanted to manually inspect clusters containing prominent obfuscation techniques. To select the candidate clusters, we assigned each cluster a *diversity score* and ranked the clusters based on this score. The idea behind the diversity score was that the popular obfuscation techniques were applied to a large number of scripts, concealing also a large variety of API features. The diversity score of a cluster was the harmonic mean[56] of the distinct number of scripts and the distinct number of feature names within the cluster for all feature sites belonging to the cluster, thus the value of diversity score for a cluster would be higher when both constituting values were larger. The top 20 clusters by diversity score contained 65,595 unique scripts with unresolved feature sites (86.48% of total unique scripts with unresolved sites). With our clusters ranked, we manually inspected 20 randomly sampled scripts from each of the top 20 clusters having the highest diversity scores with no false positives. We describe our findings in the following section.

## 8.2 Observed Obfuscation Techniques

In this section we describe the prominent obfuscation techniques we identified through our inspection of the top-ranked clusters of the unresolved feature sites. All of these techniques described here rely on complex logical structures and convoluted string manipulations to hide the API functionalities being invoked. We named each technique based on the term we gave the core logical structure of the technique. None of these techniques make use of the notorious `eval`, the function almost synonymous with JS obfuscation, marking a shift in how JS code is obfuscated that went unnoticed. The excerpts shown here are from our dataset—we removed the white-space minification for clarity and some excerpts were curtailed for brevity (designated by the use of the ellipsis).

**Technique 1: Functionality Map.** This was by far the most prevalent obfuscation technique we observed through our inspection. This obfuscation technique begins with an array containing all possible invocations throughout the script in string literal form—the *functionality map*—followed by a rotation routine that manipulates the order of the array, so that the actual indexes are only known during runtime. This rotated mapping is then leveraged by the *accessor* function, which performs the actual lookup into the map to retrieve the particular functionality to invoke (Listing 2 in appendix C). Using the combination of these two, the API functionalities are invoked throughout the script. For example, this is appending a DOM node to the document body using the `Document.append` API function:

```
document[_0x5a0e('0x3a')][_0x5a0e('0x17')](...).
```

We found a number of variations of this technique during our manual inspection: 1) no use of a rotating routine, 2) the accessor function was a simple index lookup into the rotated functionality map, and 3) no use of an accessor function, the functionality map was accessed using direct indices in octal form. We identified four off-the shelf JS obfuscation tools capable of one or more variations of this technique [15, 29, 46, 59], which coincides with the findings of Skolka et al [53], which they termed *String Array* feature of the tools. According to our clustering, 36,996 scripts contained at least one variation of this technique.

**Technique 2: Table of Accessors.** This technique relies upon a string manipulation decoder function that can create the function names or property accesses used in the script in string literal form from an encoded string and an adjustment offset (Listing 3 in appendix D).

Using this decoder function, `b("nslcLe", 15)` gets translated to `charAt`. Then, using this function, a table is created which consists entirely of calls to this accessor function with the specific arguments to concoct the functionalities to be used in the script as `a = ["", b("nslcLe", 15), b("msvvy", 19), b("enaqbz", 13), b("Qejp32Wnnwu", 4), ...];`. The rest of the script invokes API calls or accesses properties based on the table indices. For example, this is retrieving the document cookies through the global `window` variable: `window[a[130]][a[868]]`. We identified 22,752 unique scripts with this technique in our manually inspected clusters.

**Technique 3: Coordinate Munging.** This core of this technique is a decoder function and a table of numerals (*coordinates*) to feed into it (Listing 4 in appendix E).

The technique then creates multiple instances of the wrapper functions to use throughout the script: `var f = (new N).d, c = (new N).d, ...;`. Using these wrapper functions, all the API invocations and property accesses are performed through the rest of the script. For example, (we are using ellipsis to avoid using very long identifiers in code) `f("dR5...")` translates to `setTimeout` and is invoked using `window[f("dR5...")]`. There were 1,452 unique scripts with this technique in our inspected clusters.

**Technique 4: Switch-blade Function.** This technique is centered around a function consisting of a switch-case function that is returned using logical obfuscation that performs the duty of a decoder function (Listing 5 in appendix F). The technique then involves creating multiple wrapper functions that act as executor for the switch-blade function (Listing 6 in appendix F). Using these executor functions, invocations are made through the rest of the script. For example, `window.document` can be accessed as `window[Z4EE.x7K(28)]`. The number of unique scripts with this technique in our inspected clusters was 1,123.

**Technique 5: Classic String Constructor.** This is one of the classical string manipulation techniques - which relies on a decoder function to translate numerical values to concoct a string literal. The variations of this we observed included an offset argument passed in for the numerical values to translate. Here are two variations of the function that we observed in our clustering (both of them do the exact same thing with minute variations), as shown in Listing 7 in appendix G.

With these decoder functions `setTimeout` can be concocted through the call `z(36, 151, 137, 152, 120, 141, 145, 137, 147, 153, 152)` and subsequently the function can be invoked using `window[z(36, 151, ...)]`. We found 3,272 unique scripts with both variations of this technique in our inspected clusters.

## 9 LIMITATIONS & DISCUSSION

Our definition of JS obfuscation and the subsequent detection system were intrinsically dependent on dynamic analysis, which itself is limited by the particular execution flow taken, and thus does not reveal all the capabilities of the code. In our collection methodology, we did not generate inputs or simulate human browsing behavior, so the script execution through the trace logs was not exhaustive. However, JS code that executes on page load would always be observed in our system, and it is this code that performs interesting and security-relevant behaviors like loading third-party widgets and ads. Exhaustive JS code coverage through execution [37] is a tangential problem and out of scope of this paper.

The VV8 instrumentation system explicitly traces only browser API interactions and global object manipulations. This restriction suits our hypothesis and the presented obfuscation definition and analysis well, but it imposes the limitation that we cannot readily compare the obfuscation of browser API feature sites with feature sites for built-in JS APIs (e.g., `String.fromCharCode`). Furthermore, to the best of our knowledge, there does not exist a tool that can give us full stack trace of execution for the JS API triggered, which denies us context information that limits our static analysis. However, the former limitation has no impact on the validation of our core hypothesis, and the latter affected only a negligible fraction of feature sites in our validation experiment. We do not consider either of these external limitations to impact our methodology significantly.

Due to the hybrid nature of the analysis used in this paper, it is hard to establish any concrete comparison to prior works in the field of obfuscation detection. However, because of our system's not relying on ground truth data or specific trained models, our system does not suffer from the limitations of the large body of prior work, as we are able to detect obfuscation without having knowledge of it in a ground truth set or even requiring retraining of the model(s) involved.

## 10 RELATED WORK

**Obfuscation Identification.** Prior work in this area falls into two major categories: 1) identifying JS obfuscation as part of the malicious JS footprint, and 2) identifying JS obfuscation by itself as a transport. In the first category, a number of studies used trained classifier(s) on statically extracted features from the JS source [16, 30, 31, 40]. But, there also exist systems using non-static techniques: casual relations finding [2], string pattern based analysis [36]. A few studies combined both static or dynamic features to identify obfuscated malicious activity: trained classifier on static and dynamic features [14], ensemble of classifiers as a filter for malicious URLs [9], anomaly detection to identify drive-by download attacks [13]. Our system, in comparison, did not rely on any classifier training, thus avoiding the requirement for ground truth set of already labeled JS source. This enables us to detect unknown obfuscations while prior systems could only identify obfuscation traces seen in the labeled set.

There are some studies which attempted to identify JS obfuscation by itself. Choi et al. performed static string pattern analysis to identify JS obfuscation automatically on a web page level [10]. The `NOFUS` [33] system was built on the ideas similar to `ZOZZLE` [14] to classify JS obfuscation based on static hierarchical features from the AST. Similarly, `JSOD` [3] used a Bayesian classifier on context based features from the AST to classify readably obfuscated JS scripts, and Skolka et al. [53] used neural network based approach on enriched ASTs to identify obfuscation and minification footprints by specific tools in JS sources. Our system did not solely rely on static analysis and our detection process did not use any trained classifier, but instead we relied on the fundamental property that API usage that we see dynamically from a script should be evident in its source code. This also removes any requirement of retraining the models with newly available data unlike a large portion of the existing work.

**Hybrid Analysis.** There are a few prior studies utilizing the combination of static and dynamic analysis. `JStill` [58] combined runtime bytecode analysis with static information to identify obfuscated malicious functions in JS source. In contrast, we used trace footprints of JS API function calls and property access to identify obfuscation. Li et al. combined both static and dynamic analysis to detect malicious JS contained in pdf files [39]. Xu et al. conducted a measurement study [57] of obfuscation techniques over 1039 malicious samples from VirusTotal[55]. In our measurement study, we performed on a much larger scale (Alexa top 100k domains) and we did not consider the intent of the scripts we processed in terms of maliciousness.

**Deobfuscation.** A modest amount of research exists on removing JS obfuscation from scripts. This includes `Maude`, a static rule-based JS rewriter to deobfuscate JS source[6], and the dynamic semantic-based JS deobfuscator by Lu et al.[41]. However, `JSDES` attempted to perform automated deobfuscation on only malicious JS by first identifying obfuscation set manually, followed by tracing the functions capable of generating code dynamically within the obfuscated set, and simulating their execution to have deobfuscated

code [1]. In contrast, our system used both static and dynamic analysis to identify obfuscation without any detection of its intent, and was not concerned with any levels of obfuscation removal.

## 11 CONCLUSION

In this paper, we have presented a novel definition of JS obfuscation in terms of browser API features concealing. We centered our definition on the fundamental hypothesis that if we observe some runtime functionality in a script, then we should be able to statically identify the responsible code that triggered that functionality, otherwise the script is hiding its runtime behavior. We presented a system to detect JS obfuscation based on our definition and validated our hypothesis using this system. Additionally, we crawled the Alexa top 100k domains to measure the prevalence of feature concealing JS obfuscation in the concurrent web ecosystem. Our results showed that a significant number of domains (95.90% of the domains we visited) contained at least one script which hides functionality. We observed that this feature concealing effect is more widespread than `eval`, we demonstrated a data-driven system to discover several previously unseen obfuscation techniques.

Our work indicates a shift in how JS code conceals functionality on the web, which significantly affects current security analysis tools and the effort needed from human analysts to study the web. Obfuscation that is based on code generation through `eval` is fading away as more sophisticated obfuscation techniques are being employed. Future web security measurements and tools would need to take this shift into consideration in order to deal with evolving web threats.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Zubair Shafiq for their helpful feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541 and by the National Science Foundation (NSF) under grant CNS-1703375.

## REFERENCES

[1] Moataz AbdelKhalek and Ahmed Shosha. JSDES: An Automated De-Obfuscation System for Malicious JavaScript. In *Proceedings of the 12th International Conference on Availability, Reliability and Security - ARES*, 2017.
[2] I.A. Al-Taharwa, C.H. Mao, H.K. Pao, K.P. Wu, C. Faloutsos, H.M. Lee, S.M. Chen, and A.B. Jeng. Obfuscated malicious javascript detection by causal relations finding. In *Advanced Communication Technology (ICACT)*, 2011.
[3] Ismail Adel AL-Taharwa, Hahn-Ming Lee, Albert B. Jeng, Kuo-Ping Wu, Cheng-Seen Ho, and Shyi-Ming Chen. JSOD: JavaScript obfuscation detector. In *Security and Communication Networks*, 2015.
[4] Ariya Hidayat. ECMAScript parsing infrastructure for multipurpose analysis. https://esprima.org/. Accessed: 11-12-2019.
[5] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs. In *Annual International Cryptology Conference*, 2001.
[6] G Blanc, R Ando, and Y Kadobayashi. Term-Rewriting Deobfuscation for Static Client-Side Scripting Malware Detection. In *Mobility and Security (NTMS)*, 2011.
[7] Brave Software. Features | Brave Browser. https://brave.com/features/. Accessed: 11-15-2019.
[8] Brave Software. PageGraph Âů brave/brave-browser Wiki. https://github.com/brave/brave-browser/wiki/PageGraph. Accessed: 11-15-2019.
[9] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler : A Fast Filter for the Large-Scale Detection of Malicious Web Pages Categories and Subject Descriptors. In *Proceedings of the International World Wide Web Conference (WWW)*, 2011.
[10] YoungHan Choi, TaeGhyoon Kim, SeokJin Choi, and Cheolwon Lee. Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. In *International Conference on Future Generation Information Technology*, 2009.
[11] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on Software Engineering*, 2002.
[12] A. Costello. Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA). RFC 3492, RFC Editor, March 2003.
[13] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the International World Wide Web Conference (WWW)*, 2010.
[14] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proceedings of the USENIX Security Symposium*, 2011.
[15] DaftLogic. daftlogic. https://www.daftlogic.com/projects-online-javascript-obfuscator.htm. Accessed: 05-30-2020.
[16] Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2018.
[17] Ben Feinstein and Daniel Peck. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. In *Black Hat USA*, 2007.
[18] Github. Catapult Project. https://github.com/catapult-project. Accessed: 11-12-2019.
[19] Github. CDNJS - the best front-end resource CDN for free! https://cdnjs.com/. Accessed: 11-12-2019.
[20] Github. cdnjs September 2019 Usage Stats. https://github.com/cdnjs/cf-stats/blob/master/2019/cdnjs_September_2019.md. Accessed: 11-12-2019.
[21] Github. EScope. https://github.com/estools/escope. Accessed: 11-12-2019.
[22] Github. JavaScript Obfuscator. https://github.com/javascript-obfuscator/javascript-obfuscator. Accessed: 11-12-2019.
[23] Github. Puppeteer. https://github.com/GoogleChrome/puppeteer. Accessed: 11-12-2019.
[24] Github. UglifyJS âĂŞ a JavaScript parser/compressor/beautifier. https://github.com/mishoo/UglifyJS. Accessed: 11-12-2019.
[25] Github. Web Page Replay. https://github.com/catapult-project/catapult/tree/master/web_page_replay_go. Accessed: 11-12-2019.
[26] github.io. Chrome DevTools Protocol Viewer. https://chromedevtools.github.io/devtools-protocol/. Accessed: 11-12-2019.
[27] F. Howard. Malware with your Mocha? Obfuscation and antiemulation tricks in malicious JavaScript. In *Sophos Technical Papers (2010)*, 2010.
[28] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2020.
[29] Javascript Obfuscator. Javascript Obfuscator. https://javascriptobfuscator.com/default.aspx. Accessed: 05-30-2020.
[30] W Jiang, H Wang, and K Wu. Method for Detecting Javascript Code Obfuscation based on Convolutional Neural Network. In *International Journal of Performability Engineering*, 2018.
[31] Mehran Jodavi, Mahdi Abadi, and Elham Parhizkar. Jsobfusdetector: A binary pso-based one-class classifier ensemble to detect obfuscated javascript code. In *International Symposium on Artificial Intelligence and Signal Processing*, 2015.
[32] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC)*, 2019.
[33] Scott Kaplan, Benjamin Livshits, Benjamin Zorn, Christian Siefert, and Charlie Curtsinger. " NOFUS: Automatically Detecting"+ String. fromCharCode (32)+" ObFuSCateD". toLowerCase ()+" JavaScript Code. In *Technical report, Technical Report MSR-TR 2011–57, Microsoft Research*, 2011.
[34] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Proceedings of the USENIX Security Symposium*, 2013.
[35] Kaspersky. Chrome 0-day exploit cve-2019-13720 used in operation wizardopium. https://securelist.com/chrome-0-day-exploit-cve-2019-13720-used-in-operation-wizardopium/94866/. Accessed: 11-11-2019.
[36] Byung-Ik Kim, Chae-Tae Im, and Hyun-Chul Jung. Suspicious malicious web site detection with strength analysis of a javascript obfuscation. In *International Journal of Advanced Science and Technology*, 2011.
[37] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. J-force: Forced execution on javascript. In *Proceedings of the International World Wide Web Conference (WWW)*, 2017.
[38] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.
[39] Min Li, Ying Zhou, Min Yu, and Chao Liu. Combining static and dynamic analysis for the detection of malicious JavaScript-bearing PDF documents. In *Computer Science, Technology and Application*, 2016.

[40] Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, 2009.
[41] Gen Lu and Saumya Debray. Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012.
[42] Mozilla. Battery Status API. https://developer.mozilla.org/en-US/docs/Web/API/Battery_Status_API. Accessed 11-14-2019.
[43] Mozilla Web Docs. Source code submission. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/Source_Code_Submission. Accessed: 11-09-2019.
[44] Mozilla Web Docs. Standard Built-in Objects. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects. Accessed: 11-09-2019.
[45] NPM. JavaScript Obfuscator. https://www.npmjs.com/package/javascript-obfuscator. Accessed: 11-12-2019.
[46] Obfuscator.io. Obfuscator.io. https://obfuscator.io/. Accessed: 05-30-2020.
[47] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. The leaking battery - A privacy analysis of the HTML5 Battery Status API. *Lecture Notes in Computer Science*, 2015.
[48] Online Version. JavaScript Obfuscator. https://obfuscator.io/. Accessed: 11-12-2019.
[49] Brian Pfretzschner and Lotfi ben Othmane. Identification of dependency-based attacks on node. js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017.
[50] Niels Provos, Panayiotis Mavrommatis, Moheeb Rajab, and Fabian Monrose. All your iframes point to us. In *Proceedings of the USENIX Security Symposium*, 2008.
[51] Paruj Ratanaworabhan, V Benjamin Livshits, and Benjamin G Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the USENIX Security Symposium*, 2009.
[52] Scitki Learn Documentation. Scikit Learn DBSCAN. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#sklearn.cluster.DBSCAN. Accessed: 11-12-2019.
[53] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. Anything to hide? studying minified and obfuscated code in the web. In *Proceedings of the International World Wide Web Conference (WWW)*, 2019.
[54] Technology Lookup. CDN Usage Distribution in the Top 1 Million Sites. https://trends.builtwith.com/cdn. Accessed: 11-12-2019.
[55] VirusTotal. Analyze suspicious files and URLs to detect types of malware, automatically share them with the security community. http://mathworld.wolfram.com/HarmonicMean.html. Accessed: 11-13-2019.
[56] Wolfram MathWorld. Harmonic Mean. http://mathworld.wolfram.com/HarmonicMean.html. Accessed: 11-12-2019.
[57] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In *International Conference on Malicious and Unwanted Software (MALWARE)*, 2012.
[58] Wei Xu, Fangfang Zhang, and Sencun Zhu. Jstill: Mostly static detection of obfuscated malicious javascript code. In *Proceedings of the third ACM conference on Data and application security and privacy - CODASPY*, 2013.
[59] ZS Wang. jfogs. https://github.com/zswang/jfogs. Accessed: 05-30-2020.

# APPENDIX

## A. cdnjs Libraries after Filtering

| Library Name | Semantic Version | Specific File | Downloads |
|---|---|---|---|
| jQuery | 3.3.1 | jquery.min.js | 43,749,305 |
| jQuery Mousewheel | 3.1.13 | jquey.mousewheel.min.js | 36,966,724 |
| lodash.js | 4.17.11 | lodash.core.min.js | 28,930,715 |
| jQuery Cookie | 1.4.1 | jquery.cookie.min.js | 13,208,301 |
| json3 | 3.3.2 | json3.min.js | 8,570,063 |
| modernizr | 2.8.3 | modernizr.min.js | 8,404,457 |
| popper.js | 1.12.9 | popper.min.js | 6,781,952 |
| underscore.js | 1.8.3 | underscore-min.js | 6,714,896 |
| twitter-bootstrap | 3.3.7 | bootstrap.min.js | 4,960,813 |
| mobile-detect | 1.4.3 | mobile-detect.min.js | 4,638,880 |
| jquery-ui | 3.1.1 | jquery-ui.min.js | 4,321,998 |
| postscribe | 2.0.8 | postscribe.min.js | 4,240,441 |
| swiper | 4.5.0 | swiper.min.js | 4,202,031 |
| jquery.lazyload | 1.9.1 | jquery.lazyload.min.js | 4,190,760 |
| clipboard.js | 2.0.0 | clipboard.min.js | 4,131,558 |

**Table 7: Top 15 cdnjs libraries by download after filtering**

## B. Library Hash Matches in Alexa Top 100K Data

| Library | Matching Domains |
|---|---|
| jquery | 27,366 |
| twitter-bootstrap | 8,077 |
| jqueryui | 1,253 |
| Swiper | 1,094 |
| jquery-mousewheel | 599 |
| modernizr | 581 |
| clipboard.js | 513 |
| jquery-cookie | 477 |
| underscore.js | 452 |
| mobile-detect | 338 |
| postscribe | 148 |
| jquery.lazyload | 112 |
| json3 | 36 |
| lodash.js | 8 |
| popper.js | 1 |
| **Total** | **41,055** |

**Table 8: Libraries with their corresponding SHA-256 hash matches for all semantic versions in our Alexa top 100K dataset**

## C. Code Listing for Obfuscation Technique 1

```
var _0x3866 = ['object', 'date', 'forEach',...];
// The rotaion routine
(function(_0x1d538b, _0x59d6af) {
    var _0xf0ddbf = function(_0x6dddcd) {
        while (--_0x6dddcd) {
            _0x1d538b['push'](_0x1d538b['shift']());
        }
    };
    _0xf0ddbf(++_0x59d6af);
}(_0x3866, 0xf4));

// The accessor function
var _0x5a0e = function(_0x31af49, _0x3a42ac) {
    _0x31af49 = _0x31af49 - 0x0;
    var _0x526b8b = _0x3866[_0x31af49];
    return _0x526b8b;
};
```

**Listing 2: The functionality map and following rotation routine, and the accessor function for technique 1**

## D. Code Listing for Obfuscation Technique 2

```
var Ha = /[a-z]/,
Ia = /[A-Z]/,
b = function(a, b) {
        if (null == b && (b = 13), b = Number(b), a = String(a),
    ↪ 0 == b) return a;
        0 > b && (b += 26);
        for (var c, g, f, h = a.length, e = -1, d = ""; ++e < h;)
    ↪  c = a.charAt(e), Ha.test(c) ?
             (g = "abcdefghijklmnopqrstuvwxyz".indexOf(c),
              f = (g + b) % 26, d += "abcdefghijklmnopqrstuvwxyz".
    ↪ charAt(f)) : Ia.test(c) ?
                (g = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".indexOf(c),
                 f = (g + b) % 26, d += "
    ↪ ABCDEFGHIJKLMNOPQRSTUVWXYZ".charAt(f)) : d += c;
        return d
    }
```

**Listing 3: The decoder function for technique 2**

## E. Code Listing for Obfuscation Technique 3

```
var a = [17, 95, 94, 86, 24, 63, 2, 0, 1423857449, ...];
! function() {
    function N() {
        var c = "WI1R2D5dv3Xzw8Cs".split("");
        this.d = function(d) {
            if (null == d || void 0 == d) return d;
            if (d.length % a[6] != a[7]) throw Error("1100");
            for (var e = [], f = a[7]; f < d.length; f++) {
                f % a[6] == a[7] && e.push("%");
                for (var g = c, K = a[7]; K < g.length; K++)
                    if (d.charAt(f) == g[K]) {
                        e.push(K.toString(a[50]));
                        break
                    }
            }
            return decodeURIComponent(e.join(""))
        }
}
```

**Listing 4: The decoder function for technique 3**

## F. Code Listing for Obfuscation Technique 4

```
Z4EE.m7K = function() {
    var u7K = 2;
    for (; u7K !== 1;) {
        switch (u7K) {
            case 2:
                return {
                    B6Q: function(h6Q) {
                        var C7K = 2;
                        for (; C7K !== 10;) {
                            switch (C7K) {
                                case 5:
                                    var t6Q = 0,
                                        c6Q = 0;
                                    C7K = 4;
                                    break;
                                ...
}();
```

**Listing 5: The switch-blade function for technique 4**

```
Z4EE.O7K = function() {
    return typeof Z4EE.m7K.B6Q === 'function' ? Z4EE.m7K.B6Q.
    ↪ apply(Z4EE.m7K, arguments) : Z4EE.m7K.B6Q;
};
...
Z4EE.x7K = function() {
    return typeof Z4EE.m7K.B6Q === 'function' ? Z4EE.m7K.B6Q.
    ↪ apply(Z4EE.m7K, arguments) : Z4EE.m7K.B6Q;
};
```

**Listing 6: The executor functions**

## F. Code Listing for Obfuscation Technique 5

```
function Z(I) {
    var l = arguments.length,
        O = [],
        S = 1;
    while (S < l) O[S - 1] = arguments[S++] - I;
    return String.fromCharCode.apply(String, O)
}
function z(I) {
    var l = arguments.length,
        O = [];
    for (var S = 1; S < l; ++S) O.push(arguments[S] - I);
    return String.fromCharCode.apply(String, O)
}
```

**Listing 7: The string decoder function variations**