Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications

Jeremy Rack
michael.rack@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Cristian-Alexandru Staicu
staicu@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

ABSTRACT

In recent years, we have seen an increased interest in studying the software supply chain of user-facing applications to uncover problematic third-party dependencies. Prior work shows that web applications often rely on outdated or vulnerable third-party code. Moreover, real-world supply chain attacks show that dependencies can also be used to deliver malicious code, e.g., for carrying cryptomining operations. Nonetheless, existing measurement studies in this domain neglect an important software engineering practice: developers often merge together third-party code into a single file called *bundle*, which they then deliver from their own servers, making it appear as first-party code. Bundlers like Webpack or Rollup are popular open-source projects with tens of thousand of GitHub stars, suggesting that this technology is widely-used by developers. Ignoring bundling may result in underestimating the complexity of modern software supply chains.

In this work, we aim to address these methodological shortcomings of prior work. To this end, we propose a novel methodology for automatically detecting bundles, and partially reverse engineer them. Using this methodology, we conduct the first large-scale empirical study of bundled code on the web and examine its security implications. We provide evidence about the high prevalence of bundles, which are contained in 40% of all websites, and the average website includes more than one bundle. Following our methodology, we reidentify 1 051 vulnerabilities originating from 33 vulnerable npm packages, included in bundled code. Among the vulnerabilities, we find 17 critical and 59 high severity ones, which might enable malicious actors to execute attacks such as arbitrary code execution. Analyzing the low-rated libraries included in bundles, we discover 10 security holding packages, which suggest that supply-chain attacks affecting bundles are not only possible, but they are already happening.

CCS CONCEPTS

• **Security and privacy → Web application security**; *Software reverse engineering*; • **Software and its engineering → Software libraries and repositories**.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623140

KEYWORDS

software supply chain security, bundles, dependencies, JavaScript

1 INTRODUCTION

Recent high-profile security incidents show that both malicious (`SolarWinds`) and vulnerable (`Log4Shell`) dependencies can compromise the security and privacy of real-world applications. In response, practitioners proposed various techniques for reasoning about problematic dependencies, e.g., software composition analysis or software bill of materials. These approaches aim to make the reliance on third-party code more transparent. However, detecting all such dependencies is difficult, especially when dealing with closed-source applications.

In the context of web applications, dependencies on the client-side were extensively studied by researchers to identify potential security and privacy risks. Prior measurement studies [19, 31] often distinguish between first-party and third-party script inclusions to identify risks posed by external code. While useful, this approach misclassifies third-party code that developers serve from their own domains. To deal with this problem, Lauinger et al. [23] propose both static and dynamic analysis to precisely reidentify versions of popular libraries with known vulnerabilities.

All these studies ignore *bundling*, an emerging software engineering practice. To simplify the handling of dependencies in client-side JavaScript code, developers use specialized tools called bundlers. These tools merge different resources, often JavaScript files, into one or more output files, called *bundles*, which can be directly included in the web application. Thus, the generated bundles reflect all the included dependencies of an application and should, therefore, be considered when analyzing the software supply chain of web applications. An alternative way to think about bundles is as a mechanism for web applications to consume npm packages on the client-side. While convenient, such a mechanism exposes users of websites to the plethora of risks known to plague npm packages [52]. To the best of our knowledge, there is no prior work to explore this risk.

In this paper, we study the role of JavaScript bundles on the web and their potential effect on the security and privacy of websites. Concretely, we aim to detect if vulnerable or outright malicious code is included in popular websites via bundles. Two important challenges to be addressed by the study are how to detect bundles automatically and how to reverse engineer them to extract information about the bundled code. The main culprit for this difficulty is the fact that code transformations applied during the bundling process make the reversing process difficult. Prior work [23] uses mainly two methods to detect vulnerable libraries in websites: (i) Dynamic analysis that calls specific methods of a target library registered in

the global scope and observes the responses, (ii) Static analysis that compares certain features in the source code with the features in all publicly available versions of a target library. In the context of bundles, (i) is infeasible, because the libraries are not registered within the global scope. While (ii) is possible, we first need to compartmentalize the bundle by isolating the code of the included libraries from each other.

We propose a four-steps methodology to address these challenges and to study the prevalence of bundles. We first collect unique features in the output of popular bundlers to identify them as producers of candidate JavaScript files. Afterward, we use these fingerprints in a large-scale crawl to identify bundled code and save all detected bundles in a local database. Next, we investigate the reversibility of bundled code and use this acquired knowledge to design a *reversing suite* that extracts information about included libraries from a bundle. This enables us to quantify common characteristics of bundled code, e.g., number of included libraries. Finally, we gather intelligence about the security posture and reputation of the detected libraries to understand the security implications of consuming bundled code.

Concretely, our study answers the following research questions:

RQ1: *How prevalent are bundles in real-world websites?* We detect 1 086 368 bundles by crawling the Top1M Tranco list [34]. We show that bundles are very prevalent on the web by detecting at least one bundle on 40% of all crawled websites. Furthermore, we observe that Webpack is by far the most popular bundler, which is in line with its popularity on `npm`.

RQ2: *What information can be recovered by reverse engineering bundles?* Out of the collected bundles, we were able to extract all included libraries from 89 845 due to the presence of source maps, and another 257 due to a certain build option that was used. In addition to this, we compartmentalize 285 580 bundles into a module list and for another 84 920 we further derive a dependency graph between compartments. These results confirm that a great portion of bundles is indeed reversible. Studying bundled code, we notice significant differences between first-party and third-party bundles, starting with the fact that third-party bundles are much more common than first-party bundles. In contrast to this, first-party bundles contain more third-party libraries and include over three times as many unique `npm` packages as third-party bundles.

RQ3: *Which third-party libraries are contained in real-world bundles? Do any of them contain known vulnerabilities or other problematic code?* Investigating the exact libraries included in bundles, we show that there are specific libraries that appear disproportionately more in third-party bundles, e.g., the library `bowser`, which performs browser/platform/engine detection, might be used for compromizing users' privacy. We speculate that third-party bundles contain more tracking and advertisement functionality. Additionally, we discover 33 vulnerable `npm` packages included 1 051 times in bundled code. Among these vulnerabilities, there are 17 labeled critical and 59 labeled high severity, possibly exposing web applications to, e.g., arbitrary code execution. Additionally, we find that bundles sometimes include obscure npm packages that appear unmaintained, and we even identify ten security holding packages in real-world bundles. Since these packages were labeled as such in response to previous attacks and their publicly-accessible releases do not contain any JavaScript code, we conclude that real-world bundles were already affected by supply chain attacks.

Our results show that the software supply chains of web applications are much more complex than previously believed. Modern websites often consume npm packages via bundles, delivering both vulnerable and malicious code to their users. Moreover, the line between first-party and third-party code is very blurry, since bundles often contain mixed-origin code. In such conditions, blocking unwanted code, e.g., tracking scripts, is very difficult without breaking the benign functionality of websites. Overall, our work aims to raise awareness about an important development practice with deep implications for security and privacy studies of the web.

In summary, this paper contributes the following:

- The first large-scale empirical study of bundled code on the web. Additionally, we shed light on how npm packages are consumed on the client-side of web applications.
- An automated methodology for identifying and partially reverse engineering bundles. We explore different analysis techniques that can be used in the reversing process.
- Evidence that real-world bundles include vulnerable or even malicious code.

2 A PRIMER ON JAVASCRIPT BUNDLERS AND THEIR REVERSIBILITY

This section gives an overview of the JavaScript bundling process and the information about the original code that is still preserved in this process. We remind the reader that we aim to automatically detect bundles and reverse engineer them as much as possible.

2.1 Bundlers propose a paradigm shift

Traditionally, JavaScript libraries are consumed on the client-side using a `<script>` tag for each entry. The libraries can be delivered either from first- or third-party domains, and they usually register a handle in the global scope to make themselves accessible for all the code running in the same origin. For example, jQuery registers $ and Rambda *R* in the global scope. Prior to the adoption of bundlers, web developers had to manually manage these dependencies: The order in which the libraries are loaded on a website is significant, as many libraries depend on each other and, therefore, have to be loaded in the correct order. Moreover, Patra et al. [33] identify name clashes in the global scope, when loading multiple libraries, which can lead to serious software quality problems in websites.

Bundlers propose a radically different way of managing dependencies, enabling a new way of writing code for the client-side: developers write their JavaScript code using the modern JavaScript module systems (CommonJS, AMD, or ES6) and import all the functionality they need from such modules, e.g., contained in npm packages. The bundler then merges the developer's code with the relevant part of the imported dependencies, producing a self-contained JavaScript file called *bundle*, which can be directly loaded on the client-side. It is worth noting that such bundles preserve the order in which libraries are loaded in the developers' code and they do not necessary rely on the global scope for registering and consuming library code. Instead, they emulate modern JavaScript module systems, allowing for safer library consumption. Moreover, bundlers provide optimization features that increase the performance of a

```
1 (window.webpackJsonp=window.webpackJsonp||[]).push([[80],{
2 maj8:function(e,t,n){"use strict";/* object-assign, (c) Sindre Sorhus, @license MIT */ var r=Object.
      getOwnPropertySymbols,i=Object.prototype.hasOwnProperty,o=Object.prototype.propertyIsEnumerable;function a(e){if(
      null==e)throw new TypeError("Object.assign cannot be called with null or undefined");return Object(e)}...},
3 ZK3j:function(e,t,n){"use strict";var r=n("Y4pH"),i=n("qW1w");function o(e,t){return 55296==(64512&e.charCodeAt(t))&&(!(
    t<0||t+1>=e.length)&&56320==(64512&e.charCodeAt(t+1)))}function a(e){return(e>>>24|e>>>8&65280|e<<8&16711680|(255&e
      )<<24)>>>0}function s(e){return 1===e.length?"0"+e:e}t.toHex=function(e){for(var t="",n=0;n<e.length;n++)t+=s(e[n].
      toString(16));return t},t.rotr32=function(e,t){return e>>>t|e<<32-t}},
4 xy6B:function(e,t,n){"use strict";var r=n("ZK3j").rotr32;function i(e,t,n){return e&t^~e&n}function o(e,t,n){return e&t^
      e&n^t&n}function a(e,t,n){return e^t^n}t.ft_1=function(e,t,n,r){return 0===e?i(t,n,r):1===e||3===e?a(t,n,r):2===e?o
      (t,n,r):void 0},t.s1_256=function(e){return r(e,6)^r(e,11)^r(e,25)},t.g0_256=function(e){return r(e,7)^r(e,18)^e
      >>>3},t.g1_256=function(e){return r(e,17)^r(e,19)^e>>>10}}
5 });
```

Figure 1: A simplified version of a real-world bundle from `nytimes.com`. With dashed lines we highlight the compartments and with the arrow we show a direct dependency between the last two compartments.

website. Typical optimization goals for a website are: (i) reducing the initial load time, (ii) minimizing the number of necessary requests, and (iii) keeping the size of client-side code as small as possible. Let us consider a real-world example bundle to illustrate how code from different origins is mixed into a single bundle and what information is still available about the original code.

In Figure 1, we show a simplified, tiny fragment from a bundle from The New York Times' front page, available at the time of writing at https://www.nytimes.com/vi-assets/static-assets/main-888077be14ed646513fa.js. The original bundle has more than 1.45 megabytes of minified code JavaScript code that is hard to interpret manually. Moreover, many editors that we tried to use for its inspection crash when loading or searching inside the bundle.

The reversing pipeline we propose in Section 3 identifies three compartments in this example. They represent individual JavaScript files on the developer's machine, which were processed by the bundler. These compartments may originate from different libraries, but they can also correspond to a single, larger library. Our pipeline can also identify a dependency relation between compartments: the third compartment directly imports the second one to use the method `rotr32`. Moreover, by carefully inspecting the first compartment, we notice that there are important clues that can shed light on the code's provenance. For example, the comment in line 2 and the literal `"Object.assign cannot be called with null or undefined"` strongly suggest that the bundle includes a minified version of the `index.js` file in the `object-assign` npm package. It is worth reflecting on the complexity of this supply chain: (1) `object-assign`'s maintainer committed this code on GitHub and further published it on npm, (2) the developers of nytimes.com downloaded the npm package and bundled it together with other code into a large bundle of code, (3) readers of nytimes.com download the bundle and execute it inside their browser. Considering these complexities and the plethora of security problems in npm packages, we argue once again that it is important to perform a measurement study of bundled code to see if these risks propagate to the client side. We now proceed to discuss different aspects of the bundling process that enable our measurement methodology.

2.2 Support for JS modules and packages

One of our goals is to reidentify `npm` packages in real-world bundles. To avoid confusion between different terms, we start by clarifying our terminology. An `npm` package is a folder or file that has an associated `package.json` file, which describes it. These packages can be uploaded to the `npm` registry. An `npm` module is any file that is located in the `node_modules` folder of a project and can be imported in the developer's code. We call a module a *root module* if it is not imported by any other module and is thus, directly included in the code via, e.g., a `require` or `import` function call. It is thereby one root node in the dependency graph. We call a module a *sub module* if it is imported by a root module. We use libraries as a synonym for `npm` packages. If we know the names of all modules, and thus, can ensure that the code corresponds to a particular npm package, we also use root module as a synonym for library.

Currently, the two most popular module systems for JavaScript are CommonJS, in which dependencies are loaded using the `require` function, and ES6 modules, in which dependencies are loaded using `import` statements. All the bundles we analyzed in this study support both these modules systems.

```
1 (() => {
2    var cjs_modules = {[number:cjs_module#]*}
3    // Webpack "Require" function
4    function f(n){...}
5    // Webpack helper functions
6     [Helper Functions#]*
7    (() => {[ES6_modules#]*, entryPoint#})();
8 })();
```

Listing 1: The structure of a typical bundle

Listing 1 shows the structure of a typical Webpack bundle, which is representative for all the considered bundlers. The example bundle contains both CommonJS and ES6 modules. To improve readability, we use placeholders denoted by a pound (#), and an asterisk (*) to express that the enclosed expression is present zero or more times. The code starts with an immediately invoked function expression (IIFE) that wraps the entire bundled code. This is a very typical feature of bundled code and is used by all bundlers we investigated to avoid polluting the global scope. In line two, a variable is declared and initialized with a JavaScript object. This object contains a property for each CommonJS module in the bundle, where each key is a random number and the value is the code of the module. This particular listing of modules is very common for bundled code. Then, after some boilerplate code from the bundler itself, another IIFE is visible in line seven. The body of the IIFE is comprised of a list of all ES6 modules and the entry point of the bundle, consisting

of first-party code written by developers. This structure can vary based on additional resource types included, e.g., CSS files, but its general structure is a clear indicator for a Webpack bundle. Other bundlers use a similar structure, but different in subtle ways that we can use to detect the tool that produced the bundle.

Considering the rigid structure of the bundles, we find that the most effective way to identify the bundler in use is to create fingerprints of the code that the bundler adds to almost every bundle. We refer to this code as bundler *boilerplate* code. This code can have different use cases, one of which is to emulate *Node's CommonJS* module system. Most bundles are intended to run in the browser, but this module system is not supported natively by browsers. Hence, every JavaScript bundler that supports *CommonJS* modules somehow emulates this module system to ensure that modules can import their dependencies via the *require* function. *CommonJS* modules are still very prevalent in the JavaScript space, thus, the code to handle them is equally likely to be found in bundles. For example, in the case of Webpack, a custom *require* function is used, called *"__webpack_require__"*.

```
function __webpack_require__(moduleId) {
  var cachedModule = __webpack_module_cache__[moduleId];
  if (cachedModule !== undefined) {
    return cachedModule.exports;
  }
  var module = __webpack_module_cache__[moduleId] = {
    exports: {}
  };
  t__webpack_modules__[moduleId](module, module.exports,
      __webpack_require__);
  return module.exports;
 }
```

Listing 2: Webpack require function

We show the code for this Webpack require function in Listing 2, while the minified version of the same function is visible in Listing 3. Revisiting the motivating example, one can observe that this minified require function is invoked both in the second and the third compartment to load dependencies.

```
function n(e) {
    var o = r[e];
    if (void 0 !== o) return o.exports;
    var u = r[e] = {
      exports: {}
    };
    return t[e](u, u.exports, n), u.exports
  }
```

Listing 3: Webpack require function minified

The fact that these functions appear at a similar position in the code makes it trivial to detect their usages. Other bundlers, such as Parcel have a different require emulation function called *parcelRequire*. We note that even after minification, the name of the function is still preserved and can be used to detect a Parcel bundle.

2.3 Module names and minification

Minification is an important code transformation step performed during the bundling process, which aims to reduce the code size of the produced code. Depending on the minifier in use and its configuration options, the minification process substitutes all identifier names with very short random names to reduce the page load time. Minification is non-reversible in the sense that we cannot recover the original identifiers from the transformed code, making the reverse engineering process difficult. However, most string literals or builtin invocations are not affected by minification, which means that they can be used in the reversing phase of the bundles.

In some configuration of the bundlers, the names of the modules are included in each compartment and are not affected by minification. For example, in the case of Webpack, to explicitly include the module names into the bundle, the `module_id` option needs to be set to "name". This is the default case if the `mode` configuration option is set to "development". In such a case, identifying the included library is trivial, since the module names almost always uniquely identify the associated library. One might expect that minification has an effect on the module names included in a bundle because it substitutes identifiers with short character sequences. However, this is not the case since module names are included as literals, hence are not affected by minification. In contrast, the "production" mode sets this option to "deterministic", which results in the module names being substituted by random numbers. By default, Webpack runs in *production* mode, which implies that minification is enabled and that the module names are not included in the bundle. However, our empirical evidence shows that some developers actively change the default mode and enable the module_id option.

2.4 Source maps

After the bundled JavaScript code of a web application is minified for optimization purposes, debugging the web application becomes inconvenient, due to identifier names being stripped. To revert this step, developers use source maps, which map the minified source code to its original counterpart. This mapping is parsed by the browser, which then allows smooth debugging of the application, as if it would be its original source code.

Having access to source maps exposes the original source code faithfully. In the case of a bundle, a source map gives insight into the list of modules that are included in the bundle, including their respective module names and original source code. Some source maps do not include the source code for each module, but the list of module names is still preserved. In both cases, we can extract the module names from the source map.

There are different ways to include source maps into the bundle:

(1) *Eval:* The eval option is a development option that improves build and rebuild times while separating and naming the bundled modules for ease of debugging. It can also be used in combination with source maps, where the source code is included after each module.
(2) *Inline:* The source map is appended to the source file itself, sometimes encoded with base64.
(3) *Reference:* A reference to the source map file is included at the end of the file in the form of a comment. We can further distinguish between cheap source maps and typical source maps. Cheap source maps do not map to the original code, but to the transpiled, not yet minified code.

In our work, we focus on *inline* and *external* non-eval source maps. Webpack does not recommend eval for *production* or *development* mode. Moreover, during our initial experiments, we have not found many usages of eval source maps.

2.5 Relevant information preserved in bundles

Considering all the information presented in this section, when analyzing JavaScript bundles, we are interested in extracting the following information:

(1) The **compartmentalization** of a bundle, i.e., splitting the source code to separate the included modules from each other, and thereby, assigning each module to its transformed code. This is essential for any further analysis and helps us understand how many modules are contained in a bundle.
(2) The **import relationship** between the detected modules is useful to group the related modules together. We can further leverage this information to infer the number of root modules in a bundle, which is a useful approximation for the number of imported libraries in a website.
(3) The **module names** are the most valuable property for our research because they associate a name to all the detected compartments of a bundle. We can use this information to precisely identify the included libraries. As we have seen, we can obtain module names either from the compartments themselves, when certain build options are used, or from the associated source maps.
(4) High-entropy **literals** that survived the minification process and are present in specific compartments can uniquely identify a specific version of a library.

Let us now proceed to describe our methodology that aims to use this information to reidentify problematic code inside bundles.

3 METHODOLOGY

In Figure 2 we show our measurement methodology, consisting of four steps. In the first step, we infer **fingerprints** (1) for each bundler from a sample set of generated bundles. We apply the fingerprints to decide which bundler was used to produce a given bundle. Afterward, we conduct a large-scale **crawl** (2) of the web, where we apply the fingerprints to all loaded JavaScript files. Each of the identified bundles is processed by a **reversing suite** (3) to extract information about the libraries included in the bundle. Lastly, we **analyze the security** (4) of the identified libraries. Below, we provide details about each of these steps.

3.1 Infer bundler fingerprints from samples

We start this phase by generating several bundles, using the most popular bundlers used by developers. To this end, we develop a sample web application and bundle it using different build options, resource types, and resources. We then analyze the generated samples to infer certain *code snippets* that each of the considered bundlers include in the generated code. As discussed in Section 2.2, these code snippets are often helper functions or boilerplate code that emulates JavaScript module systems. Our objective in this step is to generate a set of bundles that is as diverse as possible, to ensure that the produced fingerprints cover most developers' use cases and that our fingerprints are stable across different build options.

The entry point of our sample web application imports five of the most popular JavaScript libraries: `jquery`, `momentjs`, `uuid`, `underscore`, and `lodash`. It is important to note that to avoid these resources from being deleted as a consequence of the performed optimizations, we have to reference them in the entry point. For this purpose, we add some code that exercises some of their methods.

We first manually study all the available build options of the considered bundles and try it ourselves to see how it affects the output bundle. For each bundle, we then create a list of relevant build options that affect the structure of the bundle and we exhaustively explore each of them. The different build options available can be separated into two different categories:

- *Fully inclusive:* they can be used in combination with any other option.
- *Group inclusive:* the option belongs to a group of which only one can be set at a time.

Our pipeline takes care of these constraints and alternating between different combinations of build options, it outputs all the possible bundles that can be produced with each bundler, the sample application, and the relevant build options. The generated bundles serve as a test bed for our bundle detector: Since we pose the ground truth for each entry, we ensure that our pipeline correctly identifies the right tool in every case.

By manually analyzing the generated samples, we are able to extract fingerprints that are unique to each of the considered JavaScript bundlers, across the configuration options. Each of the considered fingerprint consists of several highly-specific lines of code, containing high-entropy identifiers or literals. Hence, we believe that the existence of false positives in our results is very unlikely, i.e., the chance that code that is not produce by bundles includes such code snippets is negligible. It is worth mentioning that since our samples include both minified and non-minified samples, the produced fingerprints support both these cases. Moreover, when producing fingerprints we ensure that they are not impacted by the non-deterministic transformations of the used minifier. The produced fingerprints serve as the basis for constructing the bundle detector, which we use for classifying JavaScript files in the crawling step. Considering how easy it is to generate new samples using off-the-shelf bundles, one can avoid the manual effort in this phase by training a machine learning classifier, in the style of Skolka et al. [40]. Nonetheless, we believe that the current fingerprints-based solution suffices for a first study in this domain, and we leave this opportunity for future work to explore.

3.2 Crawling phase

In this phase, we conduct a large-scale crawl of most popular websites on the internet. On every page visit, we intercept both loaded first- and third-party JavaScript files and use the fingerprints obtained in the previous step to detect bundled code. Concretely, we visit the landing page of each website using a headless browser and wait for six seconds for the scripts to load. We then parse each intercepted script into an abstract syntactic tree and attempt to locate each fingerprint inside the tree, using sub-tree matching. In case of a match, the corresponding JavaScript bundle is saved in a database for further study. Considering our straight-forward crawling strategy, i.e., only consider landing pages and no attempt to bypass login pages, our prevalence results should be considered a lower-bound for the actual usage of bundlers on the web.

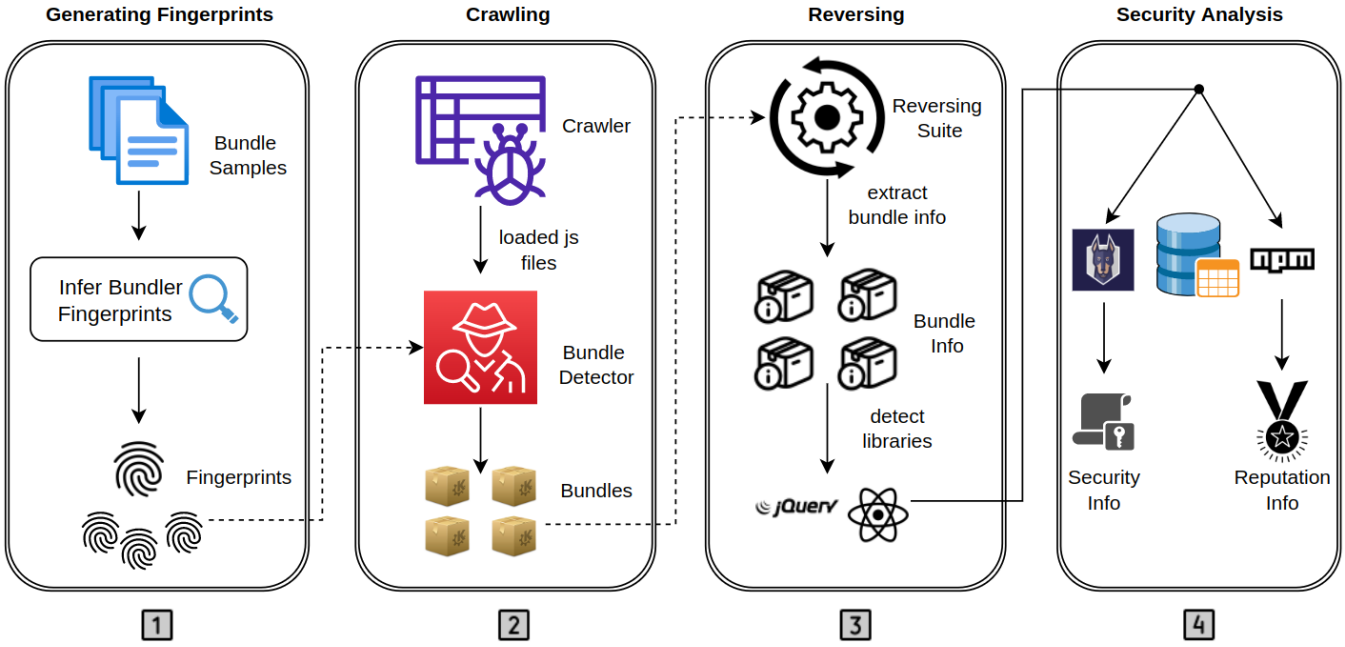

Figure 2: Our methodology for collecting JavaScript bundles from the web and reverse engineering them to obtain information about the provenance of the bundled code.

3.3 Reversing phase

The reversing suite extracts information about the included libraries, for each intercepted bundle. We describe the reversing process in Algorithm 1. The output of this algorithm is either (i) a compartments list, each corresponding to a module in the bundled code, (ii) a dependency graph (with or without module names for compartments), (iii) a list of libraries included in the bundle (with or without exact version), or (iv) nothing in case the bundle could not be parsed by our analysis pipeline. The algorithm consists of four steps, each building on the previous ones: extracting compartments (lines 1-4), building the compartments dependency graph (lines 5-16), identify embedded libraries (lines 17-27), and pinpoint the exact library version (lines 28-33). After each step, the algorithm may halt with only partial information about the bundle. Since this information suffices for answering our research questions, we leave for future work to fully reverse the majority of real-world bundles. Below, we describe in detail the four steps of the reversing phase.

In Step 1, we construct a compartment list, i.e., an array of unidentified modules where each compartment is represented by a number assigned to its source code. This step is bundler-specific in the sense that each bundler has a slightly different structure for aggregating the consumed code. Nonetheless, for all the analyzed bundlers, the compartments can be deterministicaly extracted using a simple AST-based static analysis. At this point in the reversing process, we do not know the name of the modules, nor do we know their *import* relationships.

Once we extract a module list, we further build a dependency graph (Step 2) from it. To this end, we need to first identify the emulated `require` function, which is included by each of the considered bundlers, as discussed in Section 2.2. We then locate each call site of this function and extract the literal used as argument. This literal allows us to locate the target compartment the require statement refers to. By relating the encompassing compartments with the required ones, we can construct the compartment graph, which represents the *import* relationship between modules. As noted in Section 2.5, we call each compartment that no other compartment imports a root compartment, and we note that such compartments usually correspond to an entry point in a dependent library.

In Step 3, we aim to infer the exact module names for each root compartment. We first check if a source map is available and if so, we download the source map and extract the module names, and if available, their code from the source map. Our analysis supports two types of source map inclusions[1]: directly included in the bundle (inlined) and external ones linked with a comment placed inside the bundle. Furthermore, we support both source maps that include the original code verbatim, but also lighter source maps that only include module names, paths, and line numbers. Upon successful retrieval of the source map, we group the modules by the library they belong to. This relationship is visible in the module names. For instance, let us consider the case of a bundle that includes the method `isEmpty` from the `lodash` library. The name of the `isEmpty` module is included as "/node_modules/lodash/isEmpty.js" in the source map, which reveals its relation to the `lodash` library, allowing our pipeline to conclude that the bundle contains this library. However, if no source map is available, for each module in the already detected module list, we extract its module id. As discussed in Section 2.3, in certain cases, the module names are included in the bundles as module ids. Our pipeline, thus, decides if the extracted module ids are short randomly generated identifiers,

[1]https://webpack.js.org/configuration/devtool/

or full-fledged module names, containing their relative paths in the enclosing dependent library. If the latter is the case, this allows our approach to label the constructed dependency graph with the correct module names, obtaining the same result as in the case when source maps are available.

Algorithm 1 Reversing algorithm

Input: Bundle under study $\mathcal{B}$
Output: Compartments C, compartments graph G, included libraries L, included library versions V

Step 1

$C \leftarrow \emptyset$
while $C_i \leftarrow$ MatchNewCompartment($\mathcal{B}$) **do**
  $C \leftarrow C \cup C_i$
end while

Step 2

req $\leftarrow$ GetRequireFunction($\mathcal{B}$)
if !req **then**
  return C ▷ Extraction of the compartments only
end if
$E \leftarrow \emptyset$
for $c_i \in C$ **do**
  while $r_i \leftarrow$ GetInvocationOfRequire(c_i) **do**
    $c_j \leftarrow$ GetTargetCompartment(r_i)
    $E \leftarrow E \cup \{c_i \rightarrow c_j\}$
  end while
end for
$G \leftarrow \langle C, E \rangle$

Step 3

$P \leftarrow \emptyset$
if ModuleIdsEnabled($\mathcal{B}$) **then**
  for $c_i \in C$ **do**
    $m_i \leftarrow$ GetModuleId(c_i)
    $P \leftarrow P \cup \{$TrimPackageName(m_i)$\}$
  end for
end if
if SourceMapAvailable($\mathcal{B}$) **then**
  $S \leftarrow$ RetrieveSourceMap($\mathcal{B}$)
  $P \leftarrow$ ExtractPackageNamesFromSourceMap(S)
end if
$V \leftarrow \emptyset$

Step 4

for $r_i \in C$ **do** ▷ Only iterate through root compartments
  $\sigma \leftarrow$ ExtractSignature(r_i)
  $v_i \leftarrow$ MatchSignature(σ)
  $V \leftarrow V \cup \{v_i\}$
end for
if $V \neq \emptyset$ **then**
  return V ▷ Extraction of included library versions
end if
if $L \neq \emptyset$ **then**
  return L ▷ Extraction of included library list only
end if
return G ▷ Extraction of the compartments graph only

In Step 4, we pinpoint not only the names of included libraries, but also their respective versions. At this point of our reversing suite, we are left with a dependency graph with named or unnamed modules. To identify the exact version of a library, we propose using static analysis to extract certain features (string literals, occurrences of certain keywords like "this", builtin calls, usage of class attribute) that remain untouched by the bundling process, as discussed in Section 2.3. We download each version of the target library from `npm`, bundle it, and extract these feature, persisting them in a database. First, for each of the unnamed root modules identified earlier, we extract the features from all its sub-modules, aggregate them, and compare the resulting set with the ones saved in the database to compute a similarity score. We apply a certain threshold (80%) to the score to declare the compared modules as *similar* to a particular version. This yields a range of potential versions, which we further decrease by applying a unique feature analysis. This analysis involves comparing the feature groups of each candidate version to identify features unique to that particular version or a smaller version range. Lastly, we search for these unique features in the module we are assessing to further reduce the number of possible versions.

We note that our reversing algorithm does not support the usage of submodules, so we only report complete inclusions of libraries. That is, if a bundle only includes a submodule from a library, e.g., `lodash/isEqual`, we do not count this as an inclusion of the encompasing library. That is because this would require a more precise downstream security analysis that locates the vulnerabilities we aim to reidentify, in the affected submodule. Since we are not aware of any vulnerability database that provides this data, we decided to adopt a conservative, coarse-grained library identificaiton.

3.4 Security analysis

To determine the security posture of bundles, we perform a software composition analysis at bundle level, for bundles for which we could identify their composing libraries. More specifically, we aim to reidentify known vulnerabilities or malicious code inside bundles, and to analyze the reputation of each detected bundled library. To this end, we leverage publicly available vulnerability databases to reidentify problematic code. These databases associate npm packages with their known vulnerabilities, a set of affected versions, and a severity score for each vulnerability. For cases when we can precisely identify the bundled version for a particular package, we verify if this version is affected by any vulnerability. Otherwise, we take a conservative approach and only report packages for which there are vulnerabilities known that affect all their versions. To evaluate the reputation of the found libraries, we use `npms`[2], an open-source API that exposes the official analyzer used by npm internally[3]. `npms` assigns to each package a score between 0 and 100, using multiple signals: quality, maintenance, and popularity. This score is used to sort the search results on npmjs.com, hence, we believe it is a reliable score that reflects the community's perception of reputation. After obtaining this score, we manually inspect low-reputation packages for potential security risks.

[2] https://npms.io/
[3] https://docs.npmjs.com/searching-for-and-choosing-packages-to-download

4 RESULTS

In this section, we present and discuss the results we collected for each research question, together with details about our setup.

4.1 Experimental setup

In our study, we consider the five most popular bundlers at the time of writing, namely, Webpack, Browserify, Rollup, Parcel, and Esbuild. We create a total of 36 fingerprints to identify the bundles created using different configurations of these tools. There are other JavaScript bundlers available, however, these have either not gained much popularity or internally use one of these five bundlers for the bundling process. For the reversing process, we focus on Webpack, as it is by far the most popular bundler, according to our prevalence results in Section 4.2. For the library version detection we aim to identify `lodash`, as it is a representative popular library with multiple known vulnerabilities.

To assist the replication of our study and to foster future work in this domain, we open source our measurement framework, containing the inferred fingerprints, the crawling scripts, and the detection and reversing pipeline:

https://github.com/zenoj/BundlerStudy

We crawl the Tranco Top1M list [34] from 10th of May 2022, on the same day. We use `pupeteer` with `Chrome` as the headless browser, and its plugin `puppeteer-extra-plugin-stealth` for avoiding bot detection. Out of the target one million websites, a total of 155 698 (15,56%) websites do not load any JavaScript programs, hence have no bundles either. We consider them as "unresponsive" and exclude them from our study.

We use Snyk[4] as vulnerability database. We run all our experiments on a server with 64 Intel Xeon E5-4650L@2.60GHz CPU cores, 768GB of memory, running on Debian GNU/Linux 10.

4.2 RQ1: How prevalent are bundles in real-world web applications?

To measure the prevalence of bundlers, we first examine how many bundles are included on websites and which bundler is the most popular. In total, we detect 1 086 368 bundles, on the crawled websites. Figure 3 shows the average percentage of websites that use bundles over the crawled website ranges. For instance, over the most popular 50K websites, we see that an average of about 47% of websites use bundles. Moreover, we see a correlation between the decrease in this average and the rank of the websites. This is especially visible for the first 280K websites where the decrease is most dramatic. We believe this is mostly due to the decreasing number of third-party bundles, as discussed below in Figure 5. Overall, around two out of five websites load at least one bundle on a site visit, which underlines the high prevalence of bundles on the web. Nonetheless, some websites include much more than that, e.g., gameskeys.net includes 55 bundles on its landing page. At a closer inspection, most of these bundles come from the domain kumo.network-n.com, which is flagged as suspicious by certain malware scanning services[5] or it is included in blocklists[6]. This suggests that bundles might be abused for delivering malicious code, possibly due to the limited analysis support for reasoning about bundled code.

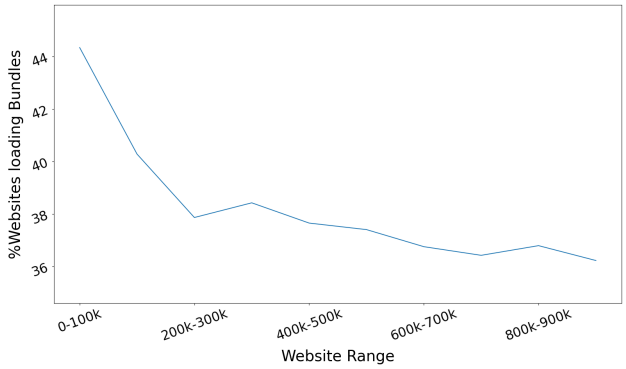

Figure 3: The prevalence of bundles

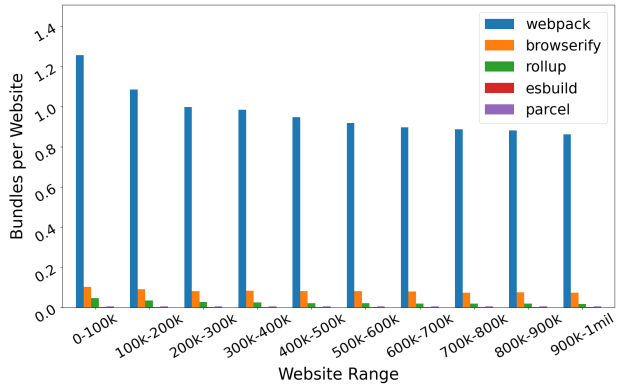

Figure 4: The prevalence of bundlers

Next, we show in Figure 4, which tools generated the detected bundles. In the most popular 100K websites, we found an average 1.27 (Webpack), 0.11 (Browserify), 0.05 (Rollup), 0.008 (Parcel), and 0.009 (Esbuild) bundles per website. We observe that the vast majority are Webpack bundles, followed by a fraction of Browserify, and some Rollup bundles. The quantity of Esbuild and Parcel is so small that their relevance is almost negligible. This result is in line with the download trends on npm. These results further show the high relevance of JavaScript bundlers on the modern web.

To increase confidence in the measurements reported in this section, the two authors of this paper manually inspected 100 detected bundles, 20 for each bundler. The raters agree that none of the analyzed samples are in fact false positives, i.e., they all represent real-world bundles. Thus, these results support the assertion that

[4]https://snyk.io/vuln/npm:npm

[5]http://any.run/report/9450c116014894fc766697192e9b0c05002affe39782da6dc27a3a6f9b7540fa/ce0d6648-0cd1-41f8-9753-c98a2489503b

[6]https://github.com/NethServer/dns-community-blacklist/blob/master/adguarddns.dns

the highly-specific fingerprints employed by our approach are very unlikely to produce false positives.

Figure 5 illustrates the average number of bundles originating from first-party or third-party, over the whole scanned website range. We can see that in the first 50 000 domains the average is about 1.25 bundles for the third-party, while the first-party is at around 0.46 bundles per domain. In general, we can see that there are a lot more third-party than first-party bundles. Even more interesting is the development of the graphs with decreasing rank. While the blue line, representing the number of first-party bundles, stays relatively constant between approximately 0.5 and 0.35, the number of third-party bundles declines to less than half of its starting value. This shows that bundlers are predominantly used for third-party scripts and that the decrease in third-party bundles is responsible for the overall decrease in bundles' prevalence in lower-ranked websites.

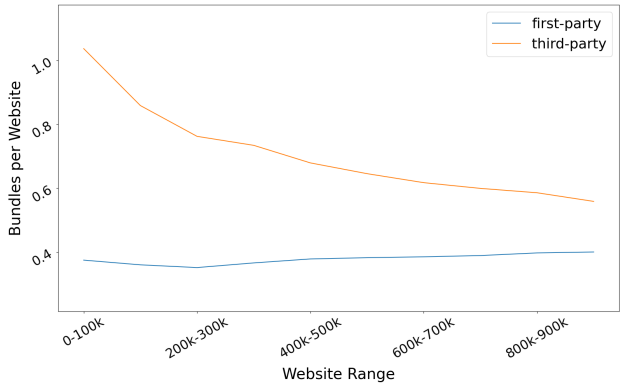

Figure 5: The origin of bundles

In contrast, the bundles' size behaves the opposite way. On one hand, the average first-party bundle size diminishes from 210 kB in popular websites to almost half this value in lowest-ranked ones. On the other hand, the third-party bundles' size stays steadily between 180 kB and 160 kB. This might indicate that higher-ranked websites bundle more self-written code than lower-ranked websites.

4.3 RQ2: To what degree can bundles be reverse engineered?

We start by investigating the number of accessible source maps, which we remind the reader provide a transparent view inside the associated bundle. Figure 6 shows the relationship between the number of Webpack bundles and their number of accessible source maps. The graph indicates that source maps are quite common, given that between every eighth and every tenth bundle includes an accessible source map that exposes the entire source code of the bundle. This finding has strong security implications: The prevalence of source maps enables us to examine third-party dependencies of about 80K websites. As discussed in Section 4.4, some of these dependencies have known vulnerabilities that might render the website vulnerable to attacks.

Altogether, we were able to parse and further investigate 46.84% of all found Webpack bundles, as shown in Figure 7. From 62.70% of

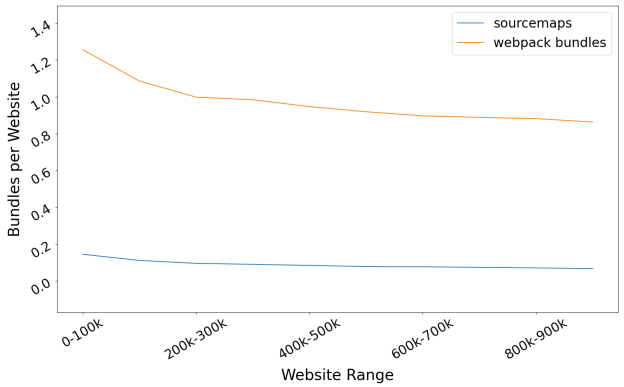

Figure 6: The prevalence of source maps

these, we can build a module list, from 18.64% we can further infer a dependency tree. We can fully reverse 18.63% of these due to accessible source maps and less than 0.03% due to the module_id build option. For the most popular 100k websites, out of approximately 126K bundles, about 12K have accessible source maps, from another 12K we can extract a dependency tree without module names, and from around 70K we can pull a list of modules without module names. Moreover, *module ids* are very rarely used, as they are not even visible in the graph. Altogether we are able to parse and to some extent reverse around half (47,35%) of all detected Webpack bundles. This is an important result that reflects the effectiveness of our analysis approach, and further implies that reversing bundles is feasible.

Additionally, we precisely identify the exact `lodash` version in 299 bundles: 281 bundles with 4.17.21, 15 with 4.17.20, and one each for 4.17.16, 4.17.15, and 4.17.10, respectively. This shows that most developers bundle the most recent version of popular libraries.

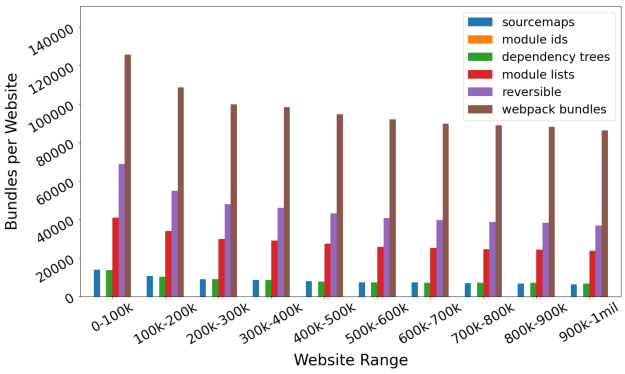

Figure 7: Reversibility of bundles. The bar labeled *reversible* is the sum of all other reversing methods that are placed to the left of it in each data point.

Below, we show some representative examples of bundles reversed engineered using our pipeline. We use the *treemap* graphical representation to visualize the diversity of the code that was merged into the bundle. In the case of module lists, the visualization looks

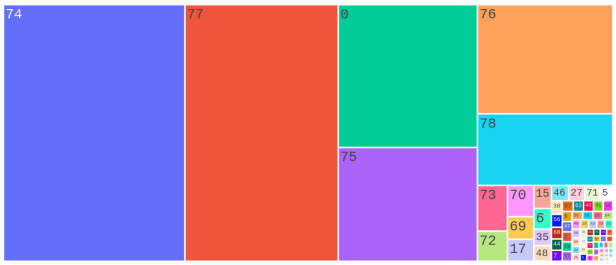

Figure 8: An opaque view of the compartments of a bundle captured on purefishing.jp.

as depicted in Figure 8. Every compartment represents a module, while the reserved space for each compartment is an indicator of its size. In total, we count 79 compartments with the biggest module `jquery` having a size of 280 kB, while the smallest module `core-js/internals/hidden-keys.js` measures only 21 B. In the case of this particular bundle, we are also able to build a dependency tree and label the included libraries correctly, due to an accessible source map. This is shown in Figure 9. For this reason, we can further group the code by first-party and third-party. We decided to combine the first-party modules into one compartment, as we are more interested in the third-party components. The comparison between Figure 8 and Figure 9 shows how multiple components can be merged using their dependency relationships. After the merging process, we can see that there are in fact only seven third-party libraries included in the bundle. Moreover, we note that this bundle includes significantly more third-party than first-party code.

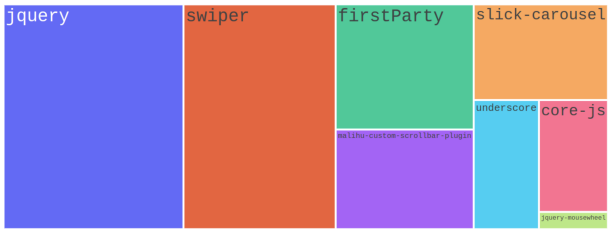

Figure 9: A transparent view of the libraries included in a bundle from purefishing.jp, whose compartments are shown in Figure 8.

This is not the case for all bundles: A bundle we captured on obo.de includes 17 third-party libraries, and a significant amount of first-party code, which represents two-thirds of the entire bundle. There are also more scattered bundles containing possibly hundreds of modules as visible in Figure 10. Merging the modules that belong to the same library (Figure 11), the same bundle (Figure 10) looks a lot different, showing the importance of grouping related modules together when trying to identify bigger code chunks like libraries.

4.4 RQ3: Which libraries are included in bundles? Are they security relevant?

In this section, we thoroughly inspect bundled libraries detected inside bundles, and attempt to study their security implications.

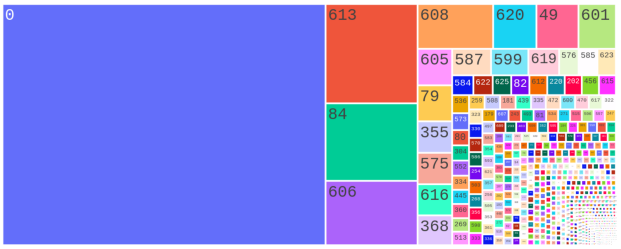

Figure 10: An opaque view of the compartments of a bundle captured on readshop.nl.

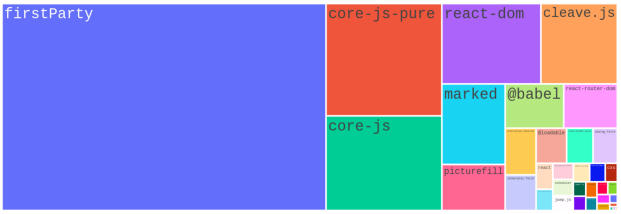

Figure 11: A transparent view of the libraries included in a bundle from readshop.nl, whose compartments are shown in Figure 10.

Bundle analysis. The relationship between the number of included third-party libraries and the scanned website ranges is depicted in Figure 12. First-party bundles include six third-party libraries on average over the first 100K websites, while third-party bundles include only 3.4 in the same range. Overall, the average bundle contains between three and six third-party libraries with first-party bundles containing significantly more than third-party ones. This high number of third-party code supports the importance of bundlers for supply-chain security.

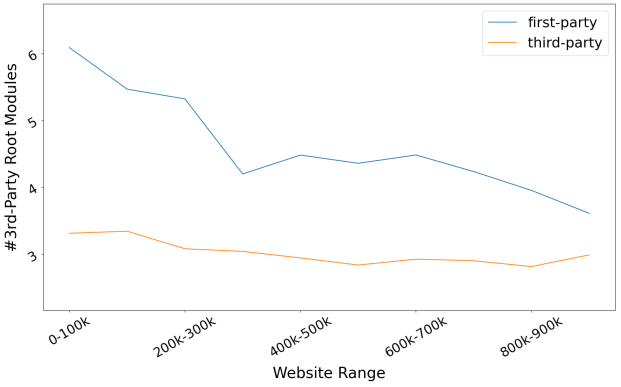

Figure 12: The number of third-party libraries

Included libraries. Our results show that about 8.3% of all bundles can be reversed due to accessible source maps and another 0.01% of bundles because the module _id build option is enabled. This leaves us with 8.3% bundles that we can assess regarding their security

posture. Altogether, we find 7 821 unique npm packages included in bundled code. Furthermore, we find 6 778 different third-party libraries in first-party bundles, whereas only 4 156 in third-party bundles. These results are in line with our earlier observations that first-party bundles include more different libraries per bundle, which leads to a more diverse bundle composition, even though there are twice as many third-party bundles.

Most bundled libraries. Figure 13 shows the ten most frequently included libraries in all bundles. `tslib` is the most popular among bundled libraries with 14 270 usages, which is not surprising considering the recent interest in gradual typing for JavaScript. A particularly interesting result, though, is the high prevalence of the library `bowser`, which performs browser's version detection. It appears in 10 520 third-party bundles, but in less than 1 000 first-party ones. This suggests that many third-party bundles include tracking code that might harm the privacy of the users. However, we note that there are many benign uses of the `bowser` library, such as specializing a given web page for a particular device or operating system. Another library with disproportionate usages in third-party bundles is `jsonp`, which suggests that third-party bundles actively attempt to perform cross-domain requests. On the contrary, around 89.4% of the usages of the `lazysizes` library can be attributed to first-party bundles. This library is a "SEO friendly lazy loader for images", which suggests that developers often bundle code for improving the performance of their website.

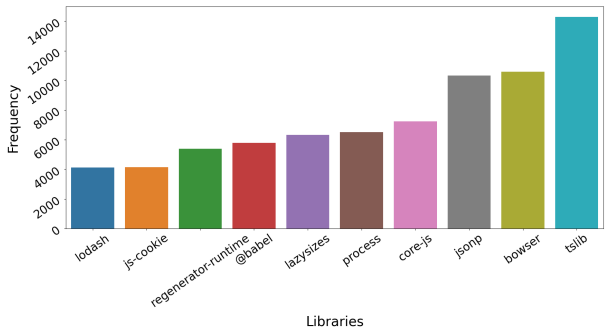

Figure 13: Top10 bundled libraries

Vulnerability assessment. Figure 1 summarizes the number of known vulnerabilities we reidentified in bundled libraries, grouped by vulnerability class. In total, we find 33 npm packages with known security issues that lead to 1 051 vulnerabilities in bundles. Out of these, 773 are *low* severity, 202 are of *medium* severity, 59 are of *high* severity, and 17 are classified as *critical* vulnerabilities. These vulnerabilities include: *arbitrary code execution*, *prototype pollution*, and *cross-site-scripting*. Additionally, three `lodash` versions (4.17.5, 4.17.10, 4.17.16) we identified in one bundle each contain known prototype pollution vulnerabilities. This shows that bundled code contains JavaScript libraries with known vulnerabilities that might have a severe impact on the security of the application that relies on it.

One year after our initial study, on 9th of August 2023, we revisited all the 1 051 affected bundles and observe that only 673 are still

Class	Affected Packages	Occurrences in Bundles
XSS	13	151 (M:128, H:8, C:15)
ReDoS	8	48 (M:37, H:11)
Prototype Pollution	4	36 (M:5, H:31)
Information Exposure	1	1 (L:1)
Improper Input Validation	2	10 (M: 8, C:2)
Command Injection	1	3 (H:3)
Arbitrary Code Execution	1	3 (H:3)
DoS	1	3 (H:3)
Insecure Credential Storage	1	12 (L:12)
HTML Injection	1	24 (M:24)
Insufficient Input Validation	1	760 (L:760)
Total	**33**	**1051**

Table 1: Vulnerabilities found in bundles. In paranthesis we show the severity of the reidentified vulnerabilities: low (L), medium (M), high (H), or critical (C).

available. Only 110 still have accessible source maps and 32 are still flagged by our approach as being vulnerable. Ten of these bundles are third-party, while the others are first party. We manually confirmed that all these bundles include an unpatched, vulnerable npm package. Finally, we reported these findings to the affected websites. On one hand, these results show that bundles evolve significantly over time and that developers tend to patch the vulnerabilities in their bundles. On the other hand, there are still bundles affected by vulnerabilities published more than 10 years ago.

Let us now proceed to show how such a long-known vulnerability can compromise the security of the embedding website. Let us consider the website https://www.gob.pe/, the official digital platform of the Republic of Peru. This website includes the npm package `awesomplete` via the bundle https://www.gob.pe/packs/js/application_base-d54f73c7964935d9242f.js. This package is known to be affected by a HTML input injection[7]. The package provides auto-complete functionality for HTML input fields, by accepting a list of texts to complete the input towards. The vulnerability report states that `awesomplete` contains a XSS gadget that takes parts of the input list and renders it into the DOM without sanitization. We note that the threat model of this vulnerability is very strong, assuming that attackers can control the auto-complete suggestions. By running code in the context of the page, we can verify that indeed the website includes the handle `Awesomplete` in the global scope, registered by the problematic npm package. Let us now add a malicious auto-complete option to the main search field on the page:

```
new Awesomplete($("#input-search-home-gobpe")[0], {
    minChars: 1,
    list: ["Fox: blah<img src=x onerror=alert('bad!')>"]
});
```

Subsequently, if the user types into the input field the word "Fox", the XSS gadget from the `awesomplete` package get triggered, and the alert is shown. This confirms that the problematic code is

[7] https://security.snyk.io/vuln/SNYK-JS-AWESOMPLETE-174474

indeed present on the website. However, in this demonstration, we assume attackers can get control over the input list to the package, an assumption we could not fullfill under a traditional web attacker model. Nonetheless, we note that attackers can attempt to leverage other vulnerabilities to take control of that input. For example, if we assume a prototype pollution vulnerability, an attacker can pollute the `Object.prototype.list` property and hijack inputs for call sites for which no list is provided as input.

The example above shows that vulnerabilities in bundles are indeed present on the affected website, but that triggering them may prove difficult and require a deep understanding of the embedding website and its threat model.

Lowest-rated packages. To investigate the reputation of bundled code, we fetch the statistics from `npm` for each package and manually analyze the 100 lowest-rated bundled `npm` packages. Overall we find that the average `npm` package score of all bundles is 0.577, while the average score of the 100 least rated libraries is 0.179. We analyze packages that originate from the same library together to avoid redundancy. This leaves us with 76 out of 100 packages, which in total are included 472 times in the examined bundles. Furthermore, these packages have an average weekly download count of 105 and were, on average, last published 3.45 years ago. However, 43 (56.58%) of them are downloaded even less than once per day. In addition to that, we find five (6.58%) officially deprecated packages. Because of this label, we recommend that developers should look for alternatives for this packages, and the research community should consider developing automatic approaches to assist this migration.

Studying the low-rated packages even more closely, we find ten (13.16%) *security holding* packages. These are placeholders for packages that were removed by `npm` due to containing malicious code. All of these ten packages are still downloaded on a daily basis, which indicates that some of the victims are still unaware that they might have included a malicious package in the past. However, these placeholder packages do not contain any JavaScript code, but only a readme file saying that nobody should use the package. All the releases of these packages containing actual (malicious) code were removed from the npm repository immediately after detection. Hence, since we have concrete evidence that the JavaScript code of these packages is contained in real-world bundles, but none of the public releases contain any JavaScript code, we conclude that the bundles might actually include the malicious version. However, pursuing this hypothesis further is extremely difficult, as it requires us to safely execute the affected bundles and uncover or at least judge the malicious behavior contained in these packages. Public advisories for npm do not include any information about what type of malicious action a flagged package performs. Thus, we leave for future work to further track the effect of supply chain attacks on real-world bundles.

5 DISCUSSION

In this section, we discuss the impact and limitations of our measurements study and suggest next steps for future work.

Results' impact and possible methodology improvements. Our results show that developers carelessly consume npm packages on the front-end, sometimes depending on vulnerable or malicious code. Since this code ultimately runs inside the JavaScript sandbox, its security impact is limited to XSS-like attacks. However, considering that a single package compromise can affect thousands of websites and the million of users using them, such an attack would have devastating consequences, e.g., attackers can harvest credentials or steal cookies at scale. Moreover, we find that security- and privacy-relevant, third-party packages are often bundled together with first-party code. This suggests that script-level blocking is not effective when privacy-sensitive libraries like `bower` are included in bundles. Future work should present more sophisticated techniques to deal with this reality.

While our methodology shows the feasibility of reverse engineering bundles, the number of bundles completely analyzed with our pipeline is relatively low. This is because we use relatively simple analysis techniques that aim to show that reversing of real-world bundles is possible and that security-relevant code is present in such bundles. Nonetheless, future work should explore more advanced program analysis techniques like dynamic analysis of bundles or machine learning-based reverse engineering.

Finally, our measurements correspond to a particular moment in time and do not provide insights into the evolution of bundles. Future work should explore how often bundles' composition changes and how fast vulnerability fixes are adopted.

Inaccuracies in bundles detection. While both false positives and false negatives are possible in our measurement setup, we took several measures to decrease both these inaccuracies. First, it is very unlikely that our highly-specific fingerprints with tens or hundreds of code tokens would flag scripts that are not bundles as such. To provide additional evidence that false positives are rare, we perform a manual analysis of 100 flagged bundles and find zero false positives. To reduce false negatives, we consider a wide variety of tools, with different configuration options. Nonetheless, we do not support every single legacy version of the considered bundlers or custom plugins available for them. Hence, we advise the reader to consider our prevalence results as a lower bound for the actual bundlers usage in the wild.

Crawling considerations. Performing a large-scale crawl poses great challenges. An important limitation is bot detection measures enforced by websites. Website administrators do not want to waste bandwidth on bots such as our crawler because it is not in their business interest. To mitigate the misuse of the website they implement bot detection heuristics that flag and ultimately block traffic originating from sources that "behave" differently than benign users. There are already complete third-party solutions available for this purpose, which monitor traffic across websites and put IP addresses on a blocklist on bot suspicion. As the presented crawl was performed from a single IP address, these blocking solutions are quite effective against crawlers like ours. We suspect that many of the unresponsive websites are actually behaving like this because the website refuses to answer with a legitimate request to our crawler.

Related work [51] identifies a disparity in loaded resources between a user and a crawler visiting the same website. This difference is visible in the number of loaded third-party scripts, which might impact a crawl like ours. We acknowledge that our measurements might suffer from this effect and thus, we advise the reader once again to consider our findings in Section 4.2 as a lower bound for

the actual prevalence of bundles. There might be other causes for under-estimating the usage of bundles in the wild, e.g., our crawl is limited to the landing page and does not explore the deep functionality of websites.

Vulnerability disclosure. Our results are in line with previous work on vulnerable library detection [5, 12, 23], in the sense that we show that vulnerable code is present on websites, but we do not show that it can actually be exploited for nefarious purposes. Considering the challenges posed by large-scale notification campaigns [45, 46] and the cost-benefits of such endeavor in our specific case, we decided not to disclose our findings to all the affected websites. While we strongly believe that the problematic dependencies should be updated as soon as possible, we think that without presenting a concrete payload and the harm it might cause, notifying all the websites that include a problematic bundle would do more harm than good. Nonetheless, to test if website maintainers are interested in such reports, we report the 32 bundles for which we could manually verify the presence of the problematic code. Nonetheless, at the time of writing, several weeks after these reports, none of the website maintainers responded to our disclosure.

Potential solutions for improving bundles security. To reduce the security impact of bundles usage, we recommend a couple of actions to be taken by the research and development community. First, we recommend that bundlers include a software composition analysis at bundling time, to alert the users about problematic packages that are about to be bundled. This approach would decrease the chance of creating new bundles with legacy vulnerable code, but it cannot serve as a reactive measure for when new vulnerabilities are discovered. To that end, we recommend that researchers build a bundlers observatory that aims to regularly verify packages included in bundles on popular websites. Companies specializing in software supply chain assurance, e.g., Snyk or Chainguard, may want to move into this space to provide their clients with a more comprehensive view of their dependencies. Nonetheless, all these solutions lack a holistic view of the bundles usage, so we also advocate for program analysis tools that can address this shortcoming. Such tools should give insights into how the independent compartments into the bundle interact with the DOM, with each other, and with security-relevant web APIs.

6 RELATED WORK

In this section we survey the closest related work, insisting on JavaScript security, empirical studies of the web, and software supply chain security.

JavaScript bundlers. Despite being used since 2013, the research community did not study in detail JavaScript bundlers, specifically with respect to security. Laurila [24] discuss software engineering use cases for different bundling software. Their analysis shows that Webpack sets itself apart through its popularity and flexibility, while Rollup is the most lightweight and Parcel the easiest to configure. A very common feature that all popular bundlers provide is solving scope conflicts between the identifiers introduced by different JavaScript libraries. Patra et al. [33] studied this problem and implemented a solution to solve it. Nonetheless, none of this work aims to identify known vulnerabilities in real-world bundles.

Transformed code. Bundlers often apply code transformations during the bundling process thus, the code they produce can be considered transformed. Skolka et al. [40] and Moog et al. [29] study two types of transformed code: minification and obfuscation. Minification is natively supported by most bundlers, thus, a very common transformation applied to bundled code. Moog et al. discover that 90% of the websites in the Alexa Top10K contain at least one transformed script. They further state that applying code transformations does not imply malicious intent and that benign code transformation techniques exist. These findings are in line with the results of Skolka et al., although they find that a much smaller percentage of scripts (38.5%) are transformed. Skolka et al. also investigate the hidden behavior of obfuscated code and show that such code tends to access privacy-sensitive APIs. This effort is related to our intention of finding hidden third-party libraries in bundles and evaluating their impact on security and privacy.

Reidentifying vulnerable dependencies. The closest related work to ours is by Lauinger et al. [23] who investigate if real-world websites include vulnerable libraries. They find that approximately 37% of websites include at least one vulnerable JavaScript library. In the case of bundles, the results appear less alarming, with only a fraction of bundles containing known vulnerabilities. As discussed in the introduction, in contrast to our methodology, the library detection techniques proposed by Lauinger et al. cannot handle bundled code. Beyond the web domain, Backes et al. [5] build library profiles using class hierarchies to identify vulnerable libraries in popular Android apps. Fischer et al. [16] measure that more than a million mobile apps use insecure code snippets copied from Stack Overflow. Furthermore, Derr et al. [12] show that outdated libraries in Android applications can be easily updated, without having to deal with breaking changes. Wang et al. [49] present a technique for identifying vulnerable code inside binaries by matching fine-grained memory layouts between the vulnerable code and target binaries. Duan et al. [14] advocates for automatically removing vulnerable code from binaries. Recent work by Azad et al. [4] and Koishybayev et al. [21] propose software debloating as a mean to reduce the attack surface of web applications by removing unused, vulnerable code. While related, none of this work deals with vulnerable code inside bundles. Nonetheless, while advocating for code-centric approaches for analyzing and updating problematic dependencies, Ponta et al. [35] identify and tackle the widely-used practice of re-bundling library code in Java applications. While related, this development practice is significantly different than JavaScript bundling, where specialized tools are used to recombine multiple files from different parties into a single bundle.

Researchers also studied how vulnerable dependencies evolve and how they are perceived by developers. Decan et al. [11] find that on average, it takes more than a month until newly-introduced vulnerabilities are discovered, but more than a year to be removed from dependant projects. By interviewing 25 developers, Pashchenko et al. [32] find that developers often rely on popularity in detriment of security posture when selecting potential dependencies and they are reluctant to update to newer versions of their dependencies due to breaking changes. These results may affect the composition and evolution of bundles, which future work should further study.

Npm security. A plethora of related work aims to identify security risks in npm packages. Zimmermann et al. [52] show that due to the tightly-interconnected nature of this ecosystem, vulnerabilities and risks can easily propagate to tens of dependant packages. Abdalkareem et al. [1] find that developers often rely on trivial packages, which leads to an over-fragmentation of the ecosystem and an amplification of the security risks. Staicu et al. [43] show that injection vulnerabilities are prevalent in npm packages and propose a hybrid program analysis technique for preventing exploitation. Moreover, Brown et al. [7] and Staicu et al. [44] show that low-level code also poses a significant risk to the ecosystem. Davis et al. [10] and Staicu et al. [42] study ReDoS vulnerabilities, an availability problem caused by the slow matching algorithms implemented in the JavaScript engines. Moreover, they show that ReDoS vulnerabilities in packages can be exploited against popular web sites. Xiao et al. [50] introduce hidden property abuse, a serialization-related problem in which attacker-controlled values are carelessly copied inside properties of internal objects. They show that this technique can be used to mount powerful injection attacks or authentication bypass. Similarly, Shcherbakov et al. [38] show that prototype pollution vulnerabilities can be leveraged for remote code execution against open-source web applications. Li et al. [26] propose a sophisticated static program analysis for detecting prototype pollutions, which they then generalize as a general-purpose vulnerability detection framework for npm packages [27]. Chinthanet et al. [9] discuss the challenges encountered when trying to build a general-purpose vulnerability detection tool for Node.js. Finally, Bhuiyan et al. [6] propose a data set of vulnerabilities with exploits to foster tool development in this domain.

In response to a series of supply chain attacks against npm, Duan et al. [13] rigorously study the indicators for possibly-malicious packages, and propose a hybrid program analysis technique for finding zero-day attacks. They uncover more than 300 unknown supply chain attacks for which CVE numbers were later assigned. Typosquatting is a type of supply-chain attack that tries to trick users into downloading a malicious package. To this end, the attacker uploads a malicious package that is spelled almost identically as a popular benign package. Taylor et al. [47] proposes using lexical similarity for detecting typosquating. In contrast, Garrett et al. [17] detect malicious packages by inspecting package releases for newly introduced security-related functions or packages. Finally, Ferreira et al. [15] and Vasilakis et al. [48] advocate for mitigating supply chain attacks by enforcing a permission system for packages. The permission system only grants a package the minimal required permissions that it needs to function. Nonetheless, AlHamdan and Staicu [3] show that language-based sandboxes are easily bypassable in JavaScript and should not be used to confine untrusted code. As we show in this work, supply chain attacks may affect bundled code as well, thus, future work should investigate the feasibility of mitigating such attacks in production websites.

Measurement studies of web security and privacy. Empirical studies of the web aimed at understanding the prevalence of security problems or the adoption of security mechanisms is a well-studied research area. There are large-scale measurements studies of the web that aim to understand: the prevalence of XSS vulnerabilities [25] or cross-site leaks [36], the inconsistent deployment of security mechanisms [37], the insecure usage of post messages [41], the likelihood of cookie hijacking [39], the prevalence of browser fingerprinting [2, 22] or cryptomining [20], the adoption of new technologies like WebSockets [30] and WebAssembly [18] or of security mechanisms like CSP [8, 28]. To the best of our knowledge, no prior work studies the prevalence and impact of JavaScript bundling on the web.

7 CONCLUSION

In our work, we conduct a large-scale empirical analysis of bundled code. We explore different characteristics of bundles and assess their posture with regards to software supply chain security. The results show the high prevalence of bundles in the web ecosystem, as more than 40% of all websites load at least one bundle, and the average number of bundles is more than one per website. We find that third-party bundles are more frequent, but contain a less diverse set of libraries than first-party bundles.

Analyzing the security of bundles, we detect 33 unique vulnerable `npm` packages, which account for a total of 1 051 vulnerabilities in bundled code. Out of these vulnerabilities, 17 are labeled as critical and 59 are of high severity. Through manual analysis of the 100 lowest-rated `npm` packages included in bundles, we discover five deprecated and ten security holding packages, which act as a placeholder for packages that were removed from the `npm` registry because they contain malicious code. These findings indicate that bundled code is vulnerable to supply-chain attacks and that bundles might have already been the targeted by such attacks.

Overall, our results show that bundles hold precious information for security analysts, which future work on web application security must take into account. Considering the plethora of risks in the npm ecosystem, we warn about the possibility of complex supply chain attacks, in which attackers compromise a single popular package to attack multiple target websites that include this package via bundles.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments, which helped us significantly improve the manuscript. We also thank Ben Stock for his early feedback on this work.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on npm. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.

[2] Gunes Acar, Marc Juárez, Nick Nikiforakis, Claudia Díaz, Seda F. Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: dusting the web for fingerprinters. In *Conference on Computer and Communications Security (CCS)*.

[3] Abdullah Alhamdan and Cristian-Alexandru Staicu. 2023. SandDriller: A Fully-Automated Approach for Testing Language-Based JavaScript Sandboxes. In *USENIX Security Symposium*.

[4] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: quantifying the security benefits of debloating web applications. In *USENIX Security Symposium*.

[5] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Conference on Computer and Communications Security (CCS)*.

[6] Masudul Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. 2023. SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript. In *International Conference on Software Engineering (ICSE)*.

[7] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *Symposium on Security and Privacy (S&P)*.

[8] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2016. Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild. In *Conference on Computer and Communications Security (CCS)*.

[9] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2020. Code-Based Vulnerability Detection in Node.js Applications: How far are we?. In *International Conference on Automated Software Engineering (ASE)*.

[10] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.

[11] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories (MSR)*.

[12] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Conference on Computer and Communications Security (CCS)*.

[13] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *Network and Distributed System Security Symposium (NDSS)*.

[14] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries. In *Network and Distributed System Security Symposium (NDSS)*.

[15] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing malicious package updates in npm with a lightweight permission system. In *International Conference on Software Engineering (ICSE)*.

[16] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *Symposium on Security and Privacy (S&P)*.

[17] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting suspicious package updates. In *International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*.

[18] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *The World Wide Web Conference (TheWebConf)*.

[19] Muhammad Ikram, Rahat Masood, Gareth Tyson, Mohamed Ali Kâafar, Noha Loizon, and Roya Ensafi. 2019. The Chain of Implicit Trust: An Analysis of the Web Third-party Resources Loading. In *The World Wide Web Conference (TheWebConf)*.

[20] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. 2019. Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild. In *The World Wide Web Conference (TheWebConf)*.

[21] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the attack surface of Node.js applications. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[22] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. 2021. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *USENIX Security Symposium*.

[23] 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)*.

[24] Sonja Laurila. 2020. Comparison of JavaScript Bundlers. (2020).

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

[26] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting Node.js Prototype Pollution Vulnerabilities via Object Lookup Analysis. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[27] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *USENIX Security Symposium*.

[28] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. 2019. Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers. In *Network and Distributed System Security Symposium, (NDSS)*.

[29] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *Conference on Dependable Systems and Networks (DSN)*.

[30] Paul Murley, Zane Ma, Joshua Mason, Michael Bailey, and Amin Kharraz. 2021. WebSocket Adoption and the Landscape of the Real-Time Web. In *The World Wide Web Conference (TheWebConf)*.

[31] 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 *Conference on Computer and Communications Security (CCS)*.

[32] Ivan Pashchenko, Duc Ly Vu, and Fabio Massacci. 2020. A Qualitative Study of Dependency Management and Its Security Implications. In *Conference on Computer and Communications Security (CCS)*.

[33] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. 2018. ConflictJS: finding and understanding conflicts between JavaScript libraries. In *International Conference on Software Engineering (ICSE)*.

[34] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Network and Distributed System Security Symposium (NDSS)*.

[35] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering (ESE)* (2020).

[36] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. 2023. The Leaky Web: Automated Discovery of Cross-Site Information Leaks in Browsers and the Web. In *Symposium on Security and Privacy (S&P)*.

[37] Sebastian Roth, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. 2022. The Security Lottery: Measuring Client-Side Web Security Inconsistencies. In *USENIX Security Symposium*.

[38] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. 2023. Silent spring: Prototype pollution leads to remote code execution in Node.js. In *USENIX Security Symposium*.

[39] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. 2016. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In *IEEE Symposium on Security and Privacy (S&P)*.

[40] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to hide? studying minified and obfuscated code in the web. In *The World Wide Web Conference (TheWebConf)*.

[41] 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)*.

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

[43] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *Network and Distributed System Security Symposium (NDSS)*.

[44] Cristian-Alexandru Staicu, Sazzadur Rahaman, Ágnes Kiss, and Michael Backes. 2023. Bilingual Problems: Studying the Security Risks Incurred by Native Extensions in Scripting Languages. In *USENIX Security Symposium*.

[45] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. 2018. Didn't You Hear Me? - Towards More Successful Web Vulnerability Notifications. In *Network and Distributed System Security Symposium (NDSS)*.

[46] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. 2016. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In *USENIX Security Symposium*.

[47] Matthew Taylor, Ruturaj K Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Spellbound: Defending against package typosquatting. *arXiv preprint arXiv:2003.03471* (2020).

[48] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Mir: Automated Quantifiable Privilege Reduction Against Dynamic Library Compromise in JavaScript. In *Conference on Computer and Communications Security (CCS)*.

[49] Haijun Wang, Xiaofei Xie, Shang-Wei Lin, Yun Lin, Yuekang Li, Shengchao Qin, Yang Liu, and Ting Liu. 2019. Locating vulnerabilities in binaries via memory layout recovering. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.

[50] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. 2021. Abusing Hidden Properties to Attack the Node.js Ecosystem. In *USENIX Security Symposium*.

[51] David Zeber, Sarah Bird, Camila Oliveira, Walter Rudametkin, Ilana Segall, Fredrik Wollsén, and Martin Lopatka. 2020. The representativeness of automated web crawls as a surrogate for human browsing. In *The World Wide Web Conference (TheWebConf)*.

[52] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium*.