D-Bundlr: Destructing JavaScript Bundles for Effective Static Analysis

Wenyuan Xu
wenyuan.xu@cs.au.dk
Aarhus University
Aarhus, Denmark

Alexi Turcotte
alexi.turcotte@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

Static analysis for vulnerability detection in JavaScript is an extensively studied research area. However, state-of-the-art approaches ignore *bundling*, an emerging development practice, akin to compilation, which allows developers to merge code from different providers, while also applying optimizations to reduce code size. A typical bundle heavily reuses single-letter identifiers and extensively relies on dynamic JavaScript features to emulate code dependencies, thus, hindering static analysis.

In this work, we propose a reverse engineering approach that relies on domain-specific code transformations to unpack bundles and replace reidentified libraries with their source code. Our technique applies lightweight static analysis to dissect bundles into individual components, machine learning to identify libraries, and dynamic analysis to verify that libraries were correctly identified. We implement this approach in a tool called D-Bundlr, and evaluate it by comparing the output of CodeQL (a popular static analysis tool) before and after debundling.

For a JavaScript code benchmark with known vulnerabilities, our approach allows CodeQL to recover 89% of the vulnerabilities and 83% of all alerts that were also detected in the source code, but were dormant in bundles. Similarly, for real-world bundles where we can retrieve the source code, D-Bundlr recovered 33% of the original alerts. When applied to bundles extracted from the 100,000 most popular websites, D-Bundlr identifies 34,445 instances corresponding to 63 unique libraries, and causes CodeQL to produce around 3.2K more security alerts than on packed bundles. We additionally illustrate how attackers can exploit some of our zero-day findings, causing unwanted security effects such as advertisement space hijacking.

CCS Concepts

• **Security and privacy → Web application security**; • **Theory of computation → Program analysis**; • **Software and its engineering** → *Scripting languages*.

Keywords

JavaScript, bundles, static analysis, vulnerabilities

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3764564

ACM Reference Format:
Wenyuan Xu, Alexi Turcotte, and Cristian-Alexandru Staicu. 2026. D-Bundlr: Destructing JavaScript Bundles for Effective Static Analysis. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3764564

1 Introduction

Static analysis is notoriously difficult for JavaScript [5], mainly due to the multitude of hard-to-analyze dynamic language features, e.g., `eval`, prototype inheritance, monkey patching. Nonetheless, since JavaScript consistently ranks among the most popular programming languages, offering millions of reusable open-source packages, there is a pressing need for program analyses tools for bug and vulnerability detection. Prior work proposes a plethora of sophisticated tools like TAJS [24], SAFE [33], CodeQL [1], ODGen [34], or Jelly [44], which perform complex, multi-file static analyses.

However, the scale of JavaScript projects has increased significantly over the last decade, mainly due to the reliance on bloated library code [27, 62, 68], which poses significant challenges for static analysis. While prior work proposes ways to statically analyze library-heavy JavaScript code [6, 46], they often assume that such libraries are self-contained and relatively rare [31, 45]. On the contrary, modern client-side JavaScript code include multiple libraries at once via bundling [53]. *Bundlers* are tools that package JavaScript applications alongside their library code into a single (or few) file, and also translate the code of the application into a form that is supported by browsers. Bundlers also optimize the code for size by removing unnecessary characters and sometimes exclude code that is unused by the application in a process called *tree-shaking*. Rack and Staicu [53] have recently showed that bundling is prevalent on the web and that a typical bundle mixes multiple libraries together with first party code into large mixed-origin, self-contained JavaScript files that appear first-party. From anecdotal evidence, we suspect that bundles are also used beyond client-side: in server-side[2], mobile[3], or cross-platform[4] contexts.

In this paper, we argue that *bundled JavaScript code is significantly harder to statically analyze than the original, manually-written code.* For instance, as opposed to bundled code, manually-written one often rely on semantics-rich identifiers, which are often referenced in static analysis policies to define security-relevant concepts such as sinks or sanitizers [15, 50]. Moreover, the original code does not contain any polyfilled code for browser compatibility (e.g., the

[1] https://codeql.github.com/
[2] https://webpack.js.org/api/node/
[3] https://webpack.js.org/guides/progressive-web-application/
[4] https://www.electronjs.org/docs/latest/tutorial/application-distribution

polyfill code for module import/export mechanism, Promise, and native functions). Bundled code often contains dynamic features that can lead to inaccuracy in static analysis. Besides, the original source code is better organized, which is useful when applying certain heuristics. For example, static analyses often rely on models for libraries and frameworks [59] that are applied when a library is encountered during analysis. As discussed in this paper, it is often difficult to say where one library ends and the next one begins in a bundle. Considering these difficulties with analyzing bundled code, one solution it to replace it with its manually-written counterpart before analysis. Unfortunately, the original code is hard to get in most scenarios, such as when developers want to test third-party scripts integrated into their own pages or when security engineers conduct black-box testing of websites or applications. Even if they have the source code, it is hard to map lines in the bundled code back to the source code, unless source mapping files are explicitly included [53].

Inspired by van Tonder and Le Goues' work [63] on tailoring code for program analysis, we propose the first debundling approach in the literature, with the goal of improving the recall of downstream static analysis. Our approach uses domain-specific code transformation to (1) unpack JavaScript bundles into composing files, (2) detect known libraries for which static analysis poses models, and (3) replace them with their original, manually-written version. Concretely, D-Bundlr first identifies the compartments of a bundle and split it into individual files, while preserving and making explicit the import relations. Next, D-Bundlr uses a fine-tuned machine learning model to predict if a given imported file corresponds to a known library and if so, it replaces it with its source code version. Debundling has similar goals to deobfuscation [13] or decompilation [9] and it provides multiple benefits to the analyzer:

(1) Through our decomposition, we aim for the static analysis of the decomposed code to be as effective as analyzing the original source code directly.
(2) For black-box and gray-box testing tools operating on bundles, we provide a mapping between the bundled code and the deconstructed code, enabling them to perform more comprehensive testing and security analysis.
(3) After we classify which bundled code belongs to user code and which belongs to which library, the analyzer can apply optimizations such as function summarization, mocking, and modeling that were originally used in source code analysis on the library code so that they can focus on user code.

We implement our approach in a tool called D-Bundlr and evaluate it on a set of packages with known vulnerabilities which we bundle ourselves and on 61,389 real-world bundles collected from the most popular 100,000 websites. We show that D-Bundlr only introduces a few new alerts that were not already in the original source code while enabling static analysis to produce 3.2K more promising alerts that were dormant in bundles. We discuss concrete zero-day vulnerabilities discovered by CodeQL when augmented with D-Bundlr.

In summary, in this paper, we make the following contributions:

- We propose the first debundling approach for improving the effectiveness of static analysis in the presence of bundles.
- We present extensive empirical evidence to show that state-of-the-art approaches struggle with analyzing bundles, and that debundling helps uncover unknown vulnerabilities.
- We provide a reusable prototype (https://anonymous.4open.science/r/D-Bundlr-5E80/) that can be integrated into existing static analysis pipelines.

The rest of this paper is organized as follows: Section 2 describes bundling and its detrimental effect on static analysis, Section 3 introduces our debundling approach, Section 4 presents empirical results to show the utility of our approach, Section 5 lists threats to validity, Section 6 discusses related work, and Section 7 concludes.

2 Background

In this section, we discuss how bundling works and why it is detrimental to static analysis. We present concrete examples using webpack, which according to Rack and Staicu [53] is both representative for existing bundlers and the most prevalent bundler on the web.

2.1 Bundling process

At a high level, bundlers merge multi-file applications into single-or-few file distributions. They also perform optimizations such as dead-code elimination via tree-shaking and code minification to reduce the size of bundles. A typical bundling process follows the following steps:

Step 1: *Load.* Given an entry file, the bundler will parse the content, supporting various input formats like JSX and TypeScript.

Step 2: *Build Dependencies Graph.* The bundler checks if the file from the previous step imports other files by checking the `require` or `import` expressions through syntactic analysis. If there are dependencies, the bundler will put them in a queue and return to Step 1; thus, all files (we will call these *modules* from now on) will be loaded. Through this process, the bundler builds a module dependency graph and collects modules into **chunks** so that the bundle can be loaded on demand or in parallel[5].

Step 3: *Compile.* In many cases, developers program in a *superset* of JavaScript, e.g., using JSX or TSX (JavaScript + HTML-like syntax) or TypeScript, so, these are transpiled in this step into standard JavaScript. This step naturally causes a loss of information, e.g., the type hints present in the TypeScript code are disposed after compilation and variables are renamed to single-letter identifiers.

Step 4: *Seal.* The bundler organizes all modules in each chunk into one big JavaScript file. This step looks like the "link" step in traditional compilation of C/C++. The bundler replaces native modules with *polyfills*, i.e., JavaScript implementing the native functionality, and perform additional optimizations[6], which make reverse engineering a challenge. For example, given a module dependencies graph, the bundler merges multiple modules into an abstract module, in case the two modules have a dependency edge and they both belong to ESM[7]. Additional optimizations are carried out at the end of this stage, like control-flow flattening. Once again, this step

[5] E.g., for webpack: https://webpack.js.org/guides/code-splitting/
[6] For details on how webpack does this, see https://github.com/webpack/webpack/blob/v5.95.0/lib/optimize/ModuleConcatenationPlugin.js.
[7] https://webpack.js.org/guides/ecma-script-modules/

alters significantly the structure of the original code in modules, making perfect reverse engineering impossible.

Step 5: *Output.* The bundler writes the code of each chunk into different files, we will describe the structure in Section 2.2.

```
1  import {useEffect} from 'react';
2  import {useLocation, useNavigate} from 'react-router-dom';
3  import queryString from 'query-string';
4  const Home = () => {
5      const location = useLocation();
6      const navigate = useNavigate();
7      useEffect(() => {
8          const url = queryString.parse(location.search).u;
9          if (url) {
10             window.location.href = url;
11         }
12     }, [location, navigate]);
13 };
14 export default Home;
```

(a) Original source code.

```
15 (() => {
16     var e = {
17         43: (e, t, n) => { t.render =... /*react code*/ }
18     },
19     /*webpack runtime functions*/
20     var l = n(43);
21     function te() { /* ... */ }
22     function re() { /* ... */ }
23     var Ke = {}
24     var Be = (s) => { /* ... */ };
25     n.d(Ke, { parse: ()=>Be });
26     const Xe = () => {
27         const e = te(),
28         t = re();
29         return (0, l.useEffect)((() => {
30             const t = Ke.parse(e.search).url;
31             t && (window.location.href = t)
32         }), [e, t])
33     };
34 })();
```

(b) Compiled code.

Figure 1: Original and bundled code from a simple React application with an open redirect vulnerability. The webpack runtime functions omitted in Figure 1b are listed in Figure 2.

To illustrate this process, consider the code in Figure 1a and its bundled version in Figure 1b. The example shows a small web application written in JavaScript, using React. The code declares a function `Home` (a React component), which will render a web page when running in the browser. In the body of `Home`, function `useLocation` from library `react-router-dom` and `queryString.parse` from library `query-string` get the parameter `u` from the URL, which will be ultimately assigned to `window.location.href`.

To bundle this application, webpack first loads the file in Figure 1a (Step 1), and identifies modules `'react'`, `'query-string'`, and `'react-router-dom'` as dependencies, also loading and analyzing them and their transitive dependencies (Step 2). In this example there is no special syntax so nothing needs to be converted (Step 3),

```
35 t={};
36 function n(r) {
37   var a = t[r];
38   if (void 0 !== a) return a.exports;
39   var l = t[r] = {exports: {}};
40   return e[r](l, l.exports, n), l.exports.
41 }
42 n.d = (e, t) => {
43   for (var r in t)
44     Object.defineProperty(e,r,{ enumerable: !0, get: t[r] });
45 }
```

Figure 2: Webpack runtime functions.

but if there was JSX in the `Home` component it would be converted now. Also, the bundler only creates a single chunk, because there are no dynamic imports. Next, since the project module and `'query-string'` use ESM modules (see the `import` statements), webpack will perform module concatenation (Step 4). All three of the aforementioned modules are merged into this chunk: the entire `'react'` library is placed in the *module map* on line 17, the `queryString` library is placed together with the application code on lines 23-25, and `useLocation` and `useNavigate` functions from `'react-router-dom'` are placed on line 21 and 22, respectively. Interestingly, since `queryString` is an ESM module, the bundler first creates an empty object (line 23), translates the `parse` function from `queryString` (line 24), and uses the special `n.d` method to add the translated `parse` method to the object, which emulates the export keyword in ESM.

2.2 Structure of a bundle

Most modules in the main chunk are bundled into a map object (*moduleObj*) (line 16) indexed by a unique module ID. The map contains a wrapper function for each module, and each wrapper function takes three arguments: the first represents the module, the second is the `exports` object, and the third is the `require` function. If the bundle contains chunks that are loaded asynchronously, the code consists of a list of push statements: the pushed element is a number array that represents the module's identifier, mapped to a wrapper function that encompasses the module's code.

The bundler inserts its own *require* function, which can be found in Figure 2. To load a module, it first looks at a cache (t); on a hit, it returns its exports field directly, otherwise it prepares the module and exports (line 39) and stores that in the cache. Then it gets the wrapper function from *moduleObj*, and calls that function (line 40). Bundlers also offer helper functions[8] to emulate other Node.js module require mechanisms. One such function is `n.d` (line 42) which simulates the `export` keyword. Here, given an object e and object t, `n.d` merges all properties in t onto e.

2.3 Challenges for static analysis

An important question we pose is: *what makes bundles so difficult to analyze?* Consider again the code in Figure 1a. With sufficient modeling of React and other libraries, static analysis can make sense of the application. Importantly, these models can also define

[8] https://github.com/webpack/webpack/blob/main/lib/RuntimeGlobals.js

taint sources and *sinks*, and static taint analysis can be used to find vulnerabilities in the application. In point of fact, there is an *open redirect vulnerability* in this code, since user-supplied data is assigned directly to `window.location.href`, which static analyses like CodeQL can detect thanks to library models. The high-level problem is that identifiers lose their semantics in the bundling process. They often get renamed to single-letter identifiers that are frequently reused across different scopes. This might lead to potential confusion for static analysis; in particular, name-based analyses [16, 17, 51] might become very imprecise in this context.

Now, let's look at this piece of code after it has been processed with webpack, in Figure 1b. As discussed in the previous section, webpack places most of the module code in the module map (line 16), and hoists some library functions, e.g., `useLocation` and `useNavigate` on line 21 and 22, respectively. Already, this process clearly obfuscates which libraries are in use, making the application of library models extremely difficult. Moreover, the way webpack emulates module loading as seen in Figure 2 makes extensive use of highly dynamic JavaScript features like dynamic read and write operations, and so context-sensitive analysis of functions like `n` (lines 36-41 in Fig. 2) is prone to path explosion as `n` will typically be called in several places. Even if we take the most advanced program analysis techniques, such as hybrid dynamic-static analysis [32], the analysis still faces challenges. The dynamic analysis can not cover all the libraries, and to the best of our knowledge, there is no general-purpose, usable demand-driven analyzer for JavaScript. And indeed, CodeQL, one of the most powerful industrial static analysis finds no vulnerabilities in scanning the bundled application of Figure 1b.

All this said, there is a straight-forward bundling process outlined in Section 2 which we aim to partially reverse. The general idea is to take bundles like the one in Figure 1b and dissect them into composing parts to reduce the detrimental effect of bundling, and at the same time preserve the semantics of the transformed code.

3 Approach

Our novel debundling technique consists of three parts:

(1) **Predictor**: given a function, predict, if possible, if it belongs to library code or to the bundler runtime;
(2) **Analyzer**: outputs information about every function, e.g., location or free variables, lightweight pointer analysis results, and the structure of each file;
(3) **Transformer**: based on the previous information, separates the bundle into individual files, potentially replacing some transformed code with its source code equivalent.

3.1 Predictor

First, we predict the provenance of each exported or wrapper function in the bundle. The predictor $Predict(f)$ takes a function f and outputs a function prototype *proto* with extra information if the function belongs to a known package, or *Unknown*. *proto* takes the form $\langle Package, Module, Function \rangle$, where *Package* is the package name that the function belongs to, *Module* is the relative path of the module where the function is being exported, and *Function* is the exported name of this function. If the *Function* is a wrapper for a module, the *Function* part of *proto* is *WRAPPER*; if the module is the entry file of a package (defined in its `package.json` under "main"[9] or "export"[10] fields), the *Module* is simply *main*.

Our approach uses three different prediction schemes:

(1) AST-pattern matching, $AstPredict(f)$
(2) Code similarity with testing, $SimPredict(f)$
(3) Machine learning with testing. $MLPredict(f)$

We try each of these prediction schemes in turn until one produces a result; otherwise, the prediction is *Unknown*.

3.1.1 AST-Based Pattern Matching. The AST-based approach will predict $\langle proto, p \rangle$ for f if there exists $(pattern, proto) \in Patterns$ such that $\textbf{match}(pattern, f) = (true, p)$. Here, $pattern \in Patterns$ is a tuple $(proto, astPattern)$, where $astPattern \in AstPattern$ is an abstraction over AST nodes with placeholders for each type of node, like ID for identifiers or ANY matching anything. For example, an *astPattern* called p_m could look like $p_m = function\ ID_1(ID_2)\{\ldots\}$. Then, $\textbf{match}(pattern, f)$ is a function that returns a tuple (b, p), with $b \in \{true, false\}$ indicating if f matches *pattern*, and p is a tuple of the pattern and a map of all concrete values associated with any placeholders in the pattern. For example, say we had a concrete AST node $f = function\ foo(x)\{\}$, then $\textbf{match}(p_m, f) = (true, (p_m, \{ID_1 \mapsto foo, ID_2 \mapsto x\}))$.

This lightweight scheme is mainly used for finding bundler require functions. We will also use $match(pattern, f)$ in Section 3.2 to find bundler boilerplate functions.

3.1.2 Code Similarity (with Validation). The code similarity-based prediction consists of two steps, a *search* step followed by a *validation* step. For a function f, we perform a similarity search over a predefined mapping $simmap : Code \rightarrow Proto$, which stores known code-to-prototype associations. We compute the similarity score $Sim(f, c)$ for each entry c in *simmap* and select the code snippet c^* that has the highest similarity. The corresponding prototype $proto^*$ and similarity score s^* are retrieved. If the best similarity score is below a predefined threshold, the search step returns *Unknown*. $Sim(code1, code2)$ first tokenizes the code: variables are tokenized as `Var:value`, strings and numbers as `const[value]`, all of other tokens are preserved as they are.Then, we compute the LCS similarity of two tokenized strings as the code's similarity. Next, we validate whether f and $proto^*$ are semantically consistent using the function $Valid(f, proto^*)$. (We will illustrate the implementation details of *Valid* in Section 3.1.4.) If the validation succeeds, we return $\langle proto^*, trace \rangle$, where *trace* is the function running trace we get from *Valid*; otherwise, we return *Unknown*. This two-stage process ensures that the final predicted prototype is not only the most similar one but also satisfies necessary validity constraints.

3.1.3 Machine learning approach (with validation). Mirroring code similarity, this approach also has a *search* and *validation* step. First, it uses a pretrained model to get the top-k candidate function prototypes predicted by the model. The model not only returns k candidates but also returns the corresponding probabilities $p_0, ..., p_k$. Only probabilities above a predefined threshold are considered, and then candidates are sorted in descending order based on their probability score p_i. Then, the approach iterates through the candidates and finds the first prototype that passes the validity check

[9] https://nodejs.org/api/packages.html#main
[10] https://nodejs.org/api/packages.html#exports

$valid(f, proto_i)$ (discussed shortly in Section 3.1.4). If no valid prototype is found, the function returns *Unknown*.

3.1.4 Validation. The approach aims to validate that a predicted prototype *proto* for a function f is consistent with the dynamic behavior by ensuring that the interface and/or behavior of the predicted prototype matches what we can observe in the bundle. Assuming $proto = \langle p, m, WRAPPER \rangle$, i.e., a module wrapper function, we first executed the compiled function with dummy arguments and check if the properties of the `exports` object(the third dummy argument we given) are a subset of the actual properties exported by the module m of package p. Otherwise, for high-frequency function prototypes, we manually specify more specific arguments, compile the prototype function, and run it in a special environment where we record its return value, as well as crucial reads ($\langle 'R', base, field \rangle$), writes ($\langle 'W', base, field, value \rangle$), and function calls ($\langle 'C', caller(args) \rangle$, where *args* is a list of *arg*) made. Then, we use the same argument to call the function f and compare if the function in bundles matches the similar return value and has similar crucial read/write/call operations that *proto* has.

In validation, the behavior of functions needs to closely approximate their behavior in a real runtime environment. However, due to factors such as the use of dummy values and differences between the execution environment (we run all the code in Node.js runtime, not in browser), directly running the program may lead to issues such as raising errors, infinite loops, or infinite recursion. To get around this, we run the code in a sandbox that intercepts calls to the bundler's require function, and also mocks the document, window, and DOM. We also use code transformations as follows.

- For any *Loop* expression, change it to *If* expression to prevent the infinite loop, for example, `for(init;test;update){body}` becomes`{init; if(test){body; update;}}`. Similar transformations are also applied to for *While* and *Do-while*.
- *Break* and *Continue* expression are removed.
- *Call* expressions are translated to prevent infinite recursion.

Please notice that these transformations are only used for validation, and we apply the same transformation both on the function in bundle and *proto*. Although this way, our function execution will not have 100% of the original function's semantics, considering that the functions to be identified are very simple (usually source or sink functions), such execution and comparison are already sufficient.

3.2 Analyzer

Once we have predictions for each exported or module wrapper function, we can begin analyzing the bundle to find invocations of the bundler's require function and build the module map. Given a JavaScript bundled file, the approach first uses pattern matching to distinguish whether the file is the *main chunk* or a *sub-chunk*. We classify the main chunk using signatures from the bundler's inserted "runtime" functions, e.g., boilerplate functions used for emulating import relationships. For sub-chunks, modules are often in a global module map or array [53] which we use for identification.

Then, we performed a lightweight pointer analysis. The bootstrap rules are the same as the traditional one. When pointer analysis reaches a fixpoint, we consult the predictions made by the previous step in order to find the bundler require function and through this we can find the module map. There are two cases: if r was predicted using *AstPredict*, then the prediction contains enough information to find the module map directly in the AST. If r was predicted by *SimPredict* or *MLPredict*, we call it in a sandbox with a dummy value v and trace any calls made by the function; one of those traces will take the form $\langle 'C', mapVar[v](*) \rangle$, and $mapVar$ is the variable pointing to the *moduleObj* object. We then build the modules map *Modules* as follows: $\forall id \in Props(moduleObj) : Modules[id] = pt_1(moduleObj.id)$. Here, $pt_1(v)$ is a function that first asserts the $sizeOf(pts(v)) = 1$ and returns that single token (if assertion failed then we say D-Bundlr doesn't support this bundle).

For each call site, if the callee's point-to set contains the bundler's require function or a dummy *require*, and the argument is string or integer c, we create a special variable with the constant to resolve exactly what in the module map was required because the bundler will use these to index the module map. E.g., on line 20 in Fig. 1b, we see the bundler require function uses an integer literal to access the module map `n(43)` .

Then, for each module in the module map *Modules*, we create three dummy objects *module*, *exports*, and *require*, place them into the points-to set of their respective parameters according to the structure we discussed in Section 2.2, and continue pointer analysis to reach a new fix point for later use. The handling of sub-chunks is simpler than for the main chunk, as we only need to use AST pattern matching to get *moduleObj* and *Modules*.

3.3 Transformer

Once we have the *Modules* map, we can begin transforming the code. For each entry $\langle id, func \rangle$ in the *Modules* map, we move the $func$ code into a separate file from the main chunk. We applied serval transformations to the code of these files and discuss a select few in this section (the rest are described in the artifact). In the end, our approach outputs module files that contain the transformed module code and a transformed main chunk.

The anatomy of these transformation rules is $\frac{cond}{ori_l \rightarrow debundled}$, which denotes that if *cond* holds, the original code *ori* at line l is transformed into *debundled*. If one *ori* has multiple transforms, like $\frac{c_1}{ori_l \rightarrow d_1}$ and $\frac{c_2}{ori_l \rightarrow d_2}$ and c_1 and c_2 both hold, we will transform ori_l to $d_1; d_2$. The order of d_x in the debundled code does not matter. Throughout, $pts(v)$ denotes the points-to set of variable v, $ConstantVar[a]$ is a variable that always has value a, and $Props(o)$ contains all the properties of objects o.

3.3.1 Transform bundler runtime functions. There are many bundler-specific runtime functions to transform, and we show two rules here for illustrative purposes, replacing the `require` function:

$$\frac{\langle r, [i], l \rangle \in Callsites \quad require \in pts(r) \quad i = ConstantVar[id]}{\texttt{r(i)}_l \longrightarrow \texttt{require('./id.js')}} \text{(require)}$$

We also must transform bundler export functions. For instance, webpack has two implementations, and we discuss one: `require.export( export, "exportField", ()=>{ return fieldFunction})`,

which is by the following rule:

$$\frac{\langle r.d, [e, field, func], l\rangle \in Callsites \quad require \in pts(r) \\ export \in pts(e) \quad field = ConstantVar[id] \\ get \in pts(func) \quad exportFunc \in pts(Return[get])}{\texttt{r.d(e, field, func)}_l \longrightarrow \texttt{exports.f=exportFunc}}$$

(export)

3.3.2 Transform predicted function or module. If module m is recognized as the index file of package p , i.e., $Predict[m] = \langle p, main, wrapper\rangle$, we replace the require:

$$\frac{m \in valuesOf(Modules) \quad Predict[m] = \langle p, main, wrapper\rangle}{\texttt{require('./id.js')} \longrightarrow \texttt{require('p')}}$$

(replace-wrapper-index)

If the function f is recognized, we replace the call site of that function to `require('p').func` .

3.3.3 Transform react `jsx` *and* `createElement` *function.* For React, a developer can write JavaScript combined with HTML syntax like `const e = <h1>hello</h1>` . Bundlers will translate this to `const e = React.jsx('h1', children: 'hello')` or `React.createElement ('h1', null, 'hello')` . Recovering JSX literals is important as many encode sinks, e.g., `<a href=`sink'␣/>` , which would not be identified in the bundle.

For the method call *createElement*, we use AST pattern matching to get and replace that call site (similar for a call to *jsx*):

$$\frac{\texttt{ANY.createElement(string, \{p1:v1, p2: v2, ...\},}c_{lst})_l}{\texttt{ANY.createElement(...}c_{lst})_l \longrightarrow \texttt{<string p1=v1 p2=v2 ...>}c_{lst}\texttt{</string>}}$$

(replace-createElement)

3.4 Training

The purpose of the model used by *MLPredict* is to recover the original function or module from bundled functions, so we prepare a dataset of (source, bundled) code pairs. Concretely, for a specified list of packages we prepare a dataset $DS = \{\langle f, proto\rangle\}$ as follows:

(1) For each package, we make a project whose entry file requires it and try to output all its exported fields (to make sure no functions are removed by bundler tree-shaking). Then we bundle that project with source mapping enabled.
(2) We apply the lightweight static analysis introduced on Section 3.2, and get the *Modules* map.
(3) For all $wrapfunc \in valuesOf(Modules)$, we use the source mapping file to get the original module code and the module location, including the package name *package* and relative module path *module*. So we have:

$$\langle wrapfunc, \langle package, module, WRAPPER\rangle\rangle \in DS,$$
$$\langle modulecode, \langle package, module, WRAPPER\rangle\rangle \in DS.$$

(4) For all hoisted functions f_h, we get the original function f_o via source map as well as the *package* and *module* of that function. Then we have:

$$\langle f_h, \langle package, module, nameof(f_o)\rangle\rangle \in DS,$$
$$\langle f_o, \langle package, module, nameof(f_o)\rangle\rangle \in DS.$$

where $nameof(f)$ return function's exported name, or definition name. We ignore anonymous functions.

For all labels (the RHS of each tuple) in *DS*, we perform one-hot encoding when training, so that when it is used for prediction, it also returns a vector which can be decoded as a map, $proto \rightarrow p$.

3.5 Semantics of Debundled Code

Even if pointer analysis is unsound, dividing module functions into different files and using the require function to link them together does not disrupt semantics. If we miss some pointer information, we will just not transform the code. Recovering JSX from bundles is also semantics-preserving if we assume that no other functions are named "jsx" or "createElement". We do risk changing semantics via library *prediction*, which is why we introduce the verification step to minimize the number of incorrect predictions.

3.6 Implementation

We implemented our approach in a tool called D-Bundlr, available at https://github.com/Anemone95/D-Bundlr.

Predictor. We chose GraphCodeBERT with a classification header as our prediction model, as it is lightweight and works well for the dataset size we consider [20]. While the training process can handle any library, our prototype currently supports all frontend packages for which CodeQL has models (we list all 250 such packages in the artifact). These packages are often security relevant, e.g., they contain sources required for taint analysis, as in `query-string`[11]. For the verification step, we manually modeled 25 functions by providing specific inputs, the expected operations in the trace and the return value (in the artifact). We provide a semi-automated script to do the extraction, and in total, it took one of the authors 10 hours to complete the modeling.

Analyzer. For the AST patterns used to detect main and sub chunks we reuse and extend the patterns proposed by Rack and Staicu [53], focusing on the webpack bundler in this prototype. We implement our static pointer analysis based on Jelly (https://github.com/cs-au-dk/jelly), a state-of-the-art static analysis tool for JavaScript. To balance scalability and precision, we use the Madhurima et al. [10] approach and limit the maximum wave to 10 and the maximum indirection round for each wave to 1.

Transformer. For the transformations recovering JSX, we adapted transformations from the webcrack project[12]. The other transformations were implemented in Babel[13].

4 Evaluation

D-Bundlr pre-processes and transforms bundles to be more amenable to static analysis, so we pose the following research questions:

(RQ1) How many alerts/vulnerabilities detected by static analysis are lost when analyzing bundles instead of source code?
(RQ2) How many such alerts/vulnerabilities are recovered when analyzing applications debundled by D-Bundlr?
(RQ3) How many false positives are introduced by D-Bundlr's debundling process?

[11] https://www.npmjs.com/package/query-string#api
[12] https://webcrack.netlify.app/
[13] https://www.npmjs.com/package/@babel/core

(RQ4) How many additional vulnerabilities are detected in bundles processed by D-Bundlr "in the wild"?
(RQ5) What is the contribution of the various components of D-Bundlr in its success?

Subject Application Selection and Basic Settings

In general, the source code corresponding to bundles is not available, and few bundles have source maps, which complicates answering **RQ1-3** as we ideally would have a set of "ground truth" confirmed vulnerabilities, or else the source code of the bundle on which we can run CodeQL. To this end, we use the **218** packages from the work of Brito et al. [8], which perform a study of the effectiveness of static vulnerability detection tools on NPM modules as the first part of our dataset; we refer to this as "ground truth dataset". We prepared a special webpack config for these to bundle them ourselves. We also use a dataset of **251** bundle scripts from the Tranco [49] top 100,000 websites[14] which contain source maps so that we can recover the original code (we check if a source map exists by checking for the following comment on the last line: `//sourcemap=...`); we refer to this as "source map dataset". This dataset contains more client-side code, but does not have a ground truth.

To answer **RQ4** and **RQ5**, we use all bundles from the Tranco top 100,000 as our dataset ("tranco top 100,000 dataset"). Each website includes multiple scripts with different origins; we assume that all scripts from the same domain belong to the same bundle and input them together into D-Bundlr, and we consider such scripts as a single script when counting scripts in Table 3.

We ran our experiment on four Dell R6525 2x AMD Epyc 7773x servers. Model training is done on one NVIDIA DGX A100 server. We gave D-Bundlr 1 hour, 10 GB of memory for analyzing each bundle. It takes ~3 days to collect all results in Table 3.

We use CodeQL (v2.20.6) to discover vulnerabilities, and configured it to also analyze library code. We used all error level severities, path-problem kinds, and security-related rules (we exclude "Hard-coded data interpreted as code" for the website datasets because this rules assume a library-level threat model, and we are not aware of how attackers can exploit this in websites). To match alerts between source code and bundle, we use the source map to link the alerts' sink locations. If two alerts' vulnerability types and sink locations match, we say they are the same. Similarly, to compare alerts between source and debundled code, we first use source map to link the alerts' sink locations from the source code to the bundle, and then test if the alert's vulnerability type is the same and if the code similarity of the sink in the bundle and in the debundled code is greater than 70%; if both are true, we say these are the same alert.

RQ1 How many alerts are lost when analyzing bundles instead of source code?

For this question, we run CodeQL on both source and bundled code in the ground truth dataset and the source map dataset. Columns **Source Code**, **Bundle**, and $\overline{\textbf{Bundle}}$ in Tables 1 and 2 show the results, where **Source Code** shows the number of vulnerabilities/alerts[15] that CodeQL detects in source code, **Bundle** is the number that CodeQL can get from the compiled bundle, and $\overline{\textbf{Bundle}}$ shows how many are missing when only analyzing bundles.

Overall, of **218** packages and **251** websites, CodeQL failed to detect all vulnerabilities and only preserved 5 alerts in ground truth dataset, while in the source map dataset, CodeQL finds 4 alerts. The main reason is that CodeQL cannot precisely analyze the webpack runtime function, which is crucial to establish cross-module dataflow to detect vulnerabilities. All of the 9 preserved alerts follow a similar pattern, namely, all of the taint sources and sinks relevant to the alerts are *globally defined*, rather than defined in some library model, and moreover all of the flows relevant to the alert are confined to a single module. E.g., taint sources include `process.env`, and sinks include regular expression constructors.

Takeaway. CodeQL detects few alerts in bundled applications.

RQ2: How many vulnerabilities are recovered when analyzing debundled code?

We ran D-Bundlr to debundle all the bundles analyzed in **RQ1**, and then ran CodeQL to check if the vulnerability reported in analyzing the source code was also present in the debundled code. In Tables 1 and 2 consider columns **D-Bundlr**, which shows all vulnerabilities and alerts that CodeQL detected from debundled code, and **Src ∩ D-Bundlr**, which shows the overlap of alerts from the source code and from debundled code. In all, CodeQL recovered 89% vulnerabilities and 83% alerts in the ground truth dataset and 33% in the source map dataset. The main reason is that D-Bundlr reverts the module dependence relationship to use native require and export mechanism, which is easier to analyze.

We manually checked the alerts that CodeQL still failed to detect when used with D-Bundlr. Many are due to missed library predictions: e.g., if the source or sink is located in unmodeled libraries, the alert will be missed. This particularly affects the ground truth dataset because D-Bundlr does not attempt to predict backend libraries like "express" or "koa". The others are due to aggressive optimizations applied by bundlers at the end of their sealing process (Step 4 in Section 2); these can be extremely challenging to reverse, but future work should explore the use of deobfuscation techniques [13]. E.g., in Figure 3, webpack significantly changes the code by flattening the control flow in the try-catch block.

Takeaway. By using D-Bundlr before analysis, CodeQL recovers 89% of vulnerabilities and 83% of alerts present in the ground truth data set, and 33% of alerts in the source map dataset.

RQ3: How many false positives are introduced by D-Bundlr's debundling process?

For this research question, we again examine the same dataset as in **RQ1** and **RQ2**, and here we look at **D-Bundlr \ src** columns which reports the cases where there are more alerts detected in the debundled application as compared with the source application.

We can see that D-Bundlr doesn't generate many "new" alerts. On source map dataset, D-Bundlr include 4 new alerts that do not

[14] we use their list made in 2025/01/23 and we made a snapshot for all websites at the same date

[15] In this paper, alerts refer to any alerts reported by CodeQL, while vulnerabilities refer to alerts that are confirmed to be exploitable.

Table 1: CodeQL results on source code, bundle, and debundled code from ground truth dataset

VulnType	Source Code		Bundle		$\overline{\text{Bundle}}$		D-Bundlr		Src ∩ D-Bundlr		D-Bundlr \ Src	
	Vulns	Alerts	Vulns	Alerts	Vulns	Alerts	Vulns	Alerts	Vulns	Alerts	Vulns	Alerts
Stored cross-site scripting	0	5	0	0	0	5	0	4	0	4	0	0
Reflected cross-site scripting	0	11	0	1	0	10	0	8	0	8	0	0
Regular expression injection	0	40	0	4	0	36	0	20	0	20	0	0
Uncontrolled command line	1	1	0	0	1	1	0	0	0	0	0	0
Uncontrolled data used in path expression	86	219	0	0	86	219	78	198	78	196	0	2
Server-side request forgery	0	1	0	0	0	1	0	1	0	1	0	0
Download of sensitive file through insecure connection	0	1	0	0	0	1	0	1	0	1	0	0
User-controlled bypass of security check	0	3	0	0	0	3	0	0	0	0	0	0
Clear-text logging of sensitive information	0	1	0	0	0	1	0	0	0	0	0	0
Log injection	0	92	0	0	0	92	0	79	0	79	0	0
Total	87	374	0	5	87	369	78	311	78	309	0	2

Table 2: CodeQL results on source code, bundle, and debundled code from source map dataset

VulnType	Source Code	Bundle	$\overline{\text{Bundle}}$	D-Bundlr	Src ∩ D-Bundlr	D-Bundlr \ Src
Unsafe HTML constructed from library input	13	0	13	5	5	0
Cross-window communication with unrestricted target origin	15	1	14	2	1	1
Client-side cross-site scripting	69	1	68	36	36	0
Clear text storage of sensitive information	1	0	1	0	0	0
Regular expression injection	31	0	31	0	0	0
User-controlled bypass of security check	3	0	3	0	0	0
Improper code sanitization	2	0	2	2	0	2
Log injection	1	0	1	0	0	0
Code injection	1	0	1	1	1	0
Client-side request forgery	5	0	5	3	2	1
Client-side URL redirect	16	2	14	7	7	0
Total	157	4	153	56	52	4

belong to the source code, while in ground truth dataset, it has two which we manually investigate. We found that in both cases the debundled code looks much different from the original source code because the control flow looks *similar to the bundled code* because D-Bundlr could not reverse some bundler optimizations.

Takeaway. By using D-Bundlr before analysis, CodeQL only reports 6 false positives across both studied datasets.

RQ4: How many additional vulnerabilities are detected "in the wild" due to D-Bundlr?

Now we report on a large-scale experiment where we use D-Bundlr to debundle all 61,389 bundle scripts we get from tranco top 100,000 dataset, 64,262 of which have bundles; this is summarized in Table 3.

First consider columns **Bundle**, **Prediction Approach** and $\textbf{New}_{\textbf{Prediction}}$: For each column, we calculate the number of alerts, as well as the number of scripts and sites with alerts. When we compute the $\textbf{New}_{\textbf{X}}$ columns, we count the number of *unique* bundles and sites that contain new alerts, and for alerts we count and sum the number of new alerts *for each bundle individually*.

In general, with D-Bundlr, CodeQL can detect 3,214 new alerts, which are related to 1,529 bundles and 7445 websites. The benefits of D-Bundlr are two-fold: first, all of the transformations from Section 3.3 remove a lot of dynamic indirection introduced by the bundling process (e.g., wrappers, optimizations) which makes the code much easier to analyze. Additionally, D-Bundlr understands the bundler's require mechanism and reverses it to make inter-module relationships more clear for static analysis. Also note that the number of websites is 4.8X greater than scripts, which supports our claim that there are many third-party scripts commonly reused by different websites, and these scripts are not secure.

We see that there are some alerts missed after debundling, e.g., for **VulnTypes Improper code sanitization** and **Code injection**. We investigated these, and found that a major source of missed alerts are some issues in CodeQL that it failed to track some kinds of dataflow between modules[16]. Checking these missed alerts is an arduous task since bundles are not designed to be human-readable, but we manually checked 10 missed alerts and did not find any cases that were not caused by CodeQL bugs. To avoid missing any true alerts, one could scan both the bundle and unbundled code.

Takeaway. By using D-Bundlr before analysis, CodeQL detects 3,214 new alerts in 1,529 bundles across 7445 websites.

RQ5: What is the contribution of the various components of D-Bundlr in its success?

We run D-Bundlr with prediction and without prediction for each bundle in the tranco top 100,000 dataset and columns **Basic Approach** (without prediction) and **Prediction Approach** (with prediction) show the results. Note that the **Basic Approach** can still use *SimPredict* to recognize the *require* function.

In most of cases, prediction performs better since we recognized crucial libraries and functions that CodeQL needs. Figure 4 shows an example, where `React.useMemo(component, dependencies)` is

[16]We raised the issue-18979 with the CodeQL team, and they confirmed.

Table 3: Results for the tranco top 100,000 websites. Bundle indicates alerts found in the bundles, Basic Approach alerts found by using D-Bundlr without prediction, and New_{Basic} indicates the new alerts relative to the bundle. Similarly, Prediction Approach indicates alerts found by D-Bundlr with prediction, and $New_{Prediction}$ indicates new alerts found with D-Bundlr with prediction relative to the bundle.

Vuln Type	Bundle			Basic Approach			New_{Basic}			Prediction Approach			$New_{Prediction}$		
	Alerts	Scripts	Sites	Alerts	Scripts	Sites	Alerts	Scripts	Sites	Alerts	Scripts	Sites	Alerts	Scripts	Sites
User-controlled bypass of security check	1050	470	660	1278	664	870	278	233	255	1322	690	897	278	233	255
Client-side request forgery	556	439	746	644	512	5997	96	91	5355	651	519	6004	101	96	5360
Clear-text logging of sensitive information	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
Clear text storage of sensitive information	644	344	1047	692	382	1094	76	55	65	722	398	1110	91	66	76
Log injection	11	4	4	12	5	5	1	1	1	12	5	5	1	1	1
Client-side URL redirect	1333	920	1905	2058	1464	2595	745	576	750	2087	1480	2612	762	585	760
Unsafe dynamic method access	29	28	51	28	27	46	0	0	0	28	27	46	0	0	0
Cross-window communication with unrestricted target origin	808	513	1568	1024	728	2401	237	234	983	1025	729	2401	238	235	983
Regular expression injection	283	241	796	276	237	792	1	1	1	282	242	797	1	1	1
Code injection	246	187	406	234	171	386	22	17	22	244	181	397	22	17	22
Database query built from user-controlled sources	4	1	1	4	1	1	0	0	0	4	1	1	0	0	0
Download of sensitive file through insecure connection	2	2	2	2	2	2	0	0	0	2	2	2	0	0	0
Client-side cross-site scripting	2384	1605	2595	3828	2608	3805	1482	1063	1305	3862	2627	3824	1504	1076	1319
Unsafe HTML constructed from library input	6219	1087	1161	6202	1071	1144	194	21	22	6232	1074	1148	194	21	22
Improper code sanitization	1060	470	674	930	455	652	21	12	12	983	479	683	21	12	12
Total	14629	4974	8953	17213	6171	14733	3154	1500	7416	17457	6272	14828	3214	1529	7445

```
async change(method, url, as, options,
        forcedScroll) { // `as` is tainted
    try {
        [pages, { __rewrites: rewrites }] = await
              Promise.all([...]);
    } catch (err) {
        window.location.href = as; // sink
        return false;
    }
}
```

(a) A relative source code about an alert report in https://cux.io/_next/static/chunks/*.js.

```
return o.default.wrap(function(o) {
    for (;;) switch (o.prev = o.next) {
        case 36:
        //...
        o.prev = 39; o.t0 = f; o.next = 44;
        return Promise.all([...]);
        case 51:
        o.prev = 51;
        o.t2 = o.catch(39);
        window.location.href = n;
        return o.abrupt("return", !1);
        //...
    }
}
```

(b) The corresponding debundled code.

Figure 3: A missed alert due to flattened control flow.

a React function that re-runs `component` only if `dependencies` changes[17]. In this code, the sensitive data `t.userId` will flow to `g` and be send out by `postMessage`. Recognizing React is crucial for this alert because the alert relies on a model of `React.useMemo`[18].

This being said, the difference between the basic approach and the approach with prediction is not too big. To shed light on this, let's look at the 5 most often predicted packages or functions in Table 4. Four of those packages (across > 6.1scripts) are related to React (*scheduler* is one of the libraries of React). These packages are

[17]https://react.dev/reference/react/useMemo

[18]https://github.com/github/codeql/blob/d1876251ee2b58d4c35c97e5f78817dbf44b4769/javascript/ql/lib/semmle/javascript/frameworks/React.qll

```
var Me = require('react');
const g = Me.useMemo(() => {
    let e = { /* ... */ };
    u && (e = {
        ...e,
        mirId: t.userId, // sensitive data
        hashSum: h.value
    });
    return e;
}, [u, c, h.value, l, t]); // if [u,c,h.value,l,t] change
const jr = e => {
  if (window.parent) {
    window.parent.postMessage(JSON.stringify(e), "*"); //sink
  }
};
Me.useEffect(() => {
    return jr(g)
}
```

Figure 4: The debundled code in https://privetmir.ru.

Table 4: Top 5 Most Frequently Predicted Packages/Functions

Packages/Functions	Number of Scripts
react	13402
react-dom	9018
scheduler	7476
crypto-js	640
react-is	497

designed with security in mind, so using their APIs as intended is generally safe, e.g., it is more difficult to write XSS-vulnerable code using React. Guo et al. [21] report that their state-of-the-art tool only find 96 alerts among 5753 scanned repositories, and CodeQL reports even less than their scanner. In other words, in the wild, around 1.6% of repositories using React have alerts, so it is not too shocking that library predictions does not find many more alerts.

We found two major reasons for missed alerts using predictions: (1) some alerts *are in the recognized packages*, and once we recognize them with prediction we replace the corresponding require with

```
f = i(57835)
m = i(25617)
h = i(87756)
r = (0,m.I0)()
g = (0,f.NE)()
// Set event handler
let e = e => {
    e.data.bannerEmail && (r((0, h.NC)()),
    c(e.data, g || ""))
}
return window.addEventListener("message", e)
// Post processing of the event data
var c = (e, t) => {
    var g = document.createElement("a"
    var v = document.createElement("img");
    v.setAttribute("src", r.brandingPlachta.img)
    g.appendChild(v)
}
```

Figure 5: Complex flow in the bundle _app-bfe264cc03dbbb8d from www1.pluska.sk, detected by D-Bundlr.

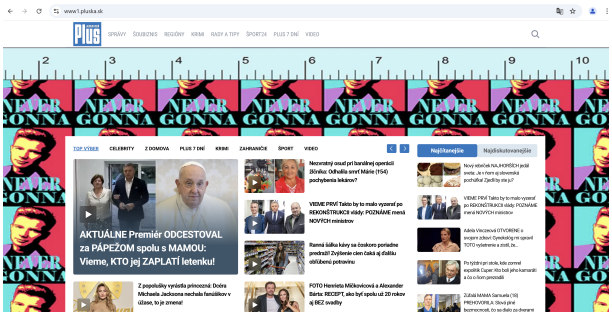

Figure 6: Hijacked advertisement space.

the predicted module (as in Section 3.3.2), and (2) the others are again related to the aforementioned CodeQL bugs.

Takeaway. D-Bundlr's library prediction detects many packages and allows CodeQL to detect more alerts.

Case Study. Let us consider `https://www1.pluska.sk/`, a popular Slovakian news website that was flagged by our approach. The bundle contains 326 assignments to the identifier `c` and more than 52 re-declarations, which can confuse CodeQL's function call resolution, but by pre-processing with D-Bundlr CodeQL could identify a flow from a post message to the `img.src` sink. This allows adversaries to steal advertisement space in the vulnerable page by sending well-crafted POST messages, and ads already included on the page might attempt such attacks to maximize their product exposure. Concretely, by sending the post message below:

```
postMessage({ bannerEmail: "bundle", plachta: "
    attackerImgLink", brandingPlachta: { img: "
    attackerImgLink" }, bannerClickUrl: "
    attackerControlledURL"}, "*")
```

Attackers can inject their own advertisements into the affected website. To do so, they can frame the site in an `iframe` to allow POST messages, due to a lax content security policy. An attacker, can hijack their ad space as shown in Fig. 6.

5 Threats to Validity

It is tricky to accurately map alerts from source code to debundled code since the source first needs to be bundled before it can be debundled and both bundling and debundling complicate the mapping. In the evaluation, we compare alert kind and code similarity of the sinks, and if both match we report the alert as the same. To mitigate potential bias, we manually checked 10 of the matches in the ground truth dataset and 10 in the source map dataset and found this approach to be accurate.

It is possible that the datasets we used for **RQ1-3** are not representative of all JavaScript. To mitigate this and balance our need to validate D-Bundlr, we selected a mix of datasets. As discussed in the introduction, bundlers are also used on server-side, and ground truth dataset contains mostly vulnerable server-side packages from npm. We augmented this with source map dataset to also validate D-Bundlr on client-side code, and finally we believe that in the larger-scale evaluation on the tranco top 100,000 dataset contains a wide variety of representative client-side JavaScript.

Finally, we present our approach as bundler-agnostic but our prototype is focused on webpack. This decision was made for practicality, as webpack is by far the most widely used bundler and is also representative of typical bundlers according to Rack and Staicu [53]. There is always an exception: the rollup bundler links modules by hoisting *every module* into the same level and resolves the require relationships during bundle preparation. We believe that static analysis should be more successful in analyzing these bundles since the bundles do not need a dynamic require mechanism, but would still benefit from the prediction phase of our approach.

6 Related Work

Transforming code to facilitate static analysis has garnered the attention of researchers. Some studies have examined the impact of transformations and compilation on static analysis results [39, 43], there is work exploring testing static analyses using semantics-preserving transformations [67], and others have explored transforming programs to make them more amenable to analysis [63]. Some work in this area perform evaluations similar to us [16, 59].

Bundles are understudied, but recently there has been an empirical study of bundles and their security implications [53], and other recent work proposed an approach to automatically rewrite privacy-harming portions of bundles at runtime [1]. Also related is debloating, e.g., by replacing unused code with "stubs" that can dynamically load the code if needed [38, 62], classify and remove JavaScript from the front-end [11], emulating usage while a page is running and debloating accordingly [29], learning-based approaches [22], configuration-based approaches [28], and using custom runtime environments [27]. Debloating makes applications more secure, and these benefits have been studied in the web domain as well [7]. The prevalence of minification and obfuscation has been studied [58], and there are approaches which attempt to de-obfuscate minified code [13], detect the applied obfuscation technique statically [42], and learning-based approaches to detect obfuscated malicious code [52]. Ren et al. [54] study the impact of obfuscation on learning-based approaches to detect malicious JavaScript, and detecting such malicious code is well-studied [23, 25, 26, 37].

Library detection has also been studied extensively. On the front-end, Liu and Ziarek [35] propose a novel data structure which helps in the detection of libraries in bundled applications, and further extend this work to accurately detect library versions [36]. Inferring libraries correctly can enable analysis to take advantage of previously computed/known *summaries* of analysis results and make analysis more compositional, e.g., function summaries [48, 55, 66], and importantly library summaries [40, 56, 60]. Chow et al. [16] leverage library API names to suppress taint analysis false positives, Chibotaru et al. [14] infer taint specifications from large software repositories, and Tileria et al. [61] extract taint specification from software documentation. Detecting React is also quite valuable, React itself has garnered the attention of the research community [18, 41], and specifically on the effect of client-side third-party applications on application vulnerability [4].

Studying how applications interact with their dependencies in a *software supply chain* has gained attention in recent years. Lauinger et al. [31] study library usage patterns and Nikiforakis et al. [45] the trust relationship in front-end JavaScript applications. Shen et al. [57] study how sources of taint are propagated along software supply chains. Williams et al. [65] present a survey of software supply chain vulnerability literature, and Ladisa et al. [30] a taxonomy of attacks. Vasilakis et al. [64] propose a technique to circumvent the need for third-party libraries by learning library behavior and replacing libraries with models. Within the supply chain, Patra et al. [47] studied conflicts *between* libraries, and Ferreira et al. [19] propose an approach that distills library execution to state evolution of objects in the library to detect vulnerabilities.

Given the security concerns alluded to in this paper, some users prefer to block JavaScript from running at all (e.g., through a learning-based approach [2] or runtime behavior [12]), but this has negative consequences, e.g., Amjad et al. [3] study if such blocking can be achieved without sacrificing legitimate website functionality.

7 Conclusion

In this paper, we propose D-Bundlr, the first debundling technique for improving the recall of static analysis when analyzing bundles. D-Bundlr combines static, dynamic, and machine learning to unpack bundles, identify known libraries, and replace them with their original source code. We show that our debundling approach helps state-of-the-art static analysis produce more security warnings, and discuss how attackers can exploit some of these previously-unknown findings. We believe that both academics and practitioners should better support emerging trends in JavaScript development, e.g., bundling, by tailoring program analyses to accommodate these realities. Moreover, future work on debundling should explore the similarities to decompilation and reuse the vast body of knowledge in this adjacent field.

Acknowledgments

This research was supported by the Danish Research Council for Technology and Production, grant ID 10.46540/3105-00037B. This research was partially funded by the European Union (ERC "Semantics of Software System", S3, 101093186). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

We gratefully acknowledge Markus Kötter's support in utilizing CISPA's computational resources. We would also like to thank Anders Møller and Andreas Zeller for their advice, support, and enthusiasm for our work.

References

[1] Mir Masood Ali, Peter Snyder, Chris Kanich, and Hamed Haddadi. 2024. Unbundle-Rewrite-Rebundle: Runtime Detection and Rewriting of Privacy-Harming Code in JavaScript Bundles. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, (CCS)*.

[2] Abdul Haddi Amjad, Shaoor Munir, Zubair Shafiq, and Muhammad Ali Gulzar. 2024. Blocking Tracking JavaScript at the Function Granularity. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*. ACM, 2177–2191.

[3] Abdul Haddi Amjad, Zubair Shafiq, and Muhammad Ali Gulzar. 2023. Blocking JavaScript Without Breaking the Web: An Empirical Investigation. *Proc. Priv. Enhancing Technol.* 2023, 3 (2023), 391–404.

[4] Terzi Anastasia and Bibi Stamatia. 2024. Managing Security Vulnerabilities Introduced by Dependencies in React. JS JavaScript Framework. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*. IEEE, 126–133.

[5] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.* 50, 5 (2017), 66:1–66:36.

[6] Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. ACM, 17–31.

[7] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*. 1697–1714.

[8] Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, and Nuno Santos. 2023. Study of javascript static analysis tools for vulnerability detection in Node.js packages. *IEEE Transactions on Reliability* (2023).

[9] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. 2022. Decomperson: How Humans Decompile and What We Can Learn From It. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 2765–2782.

[10] Madhurima Chakraborty, Aakash Gnanakumar, Manu Sridharan, and Anders Møller. 2024. Indirection-Bounded Call Graph Analysis. In *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria (LIPIcs, Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:22. https://doi.org/10.4230/LIPICS.ECOOP.2024.10

[11] Moumena Chaqfeh, Yasir Zaki, Jacinta Hu, and Lakshmi Subramanian. 2020. JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*. ACM / IW3C2, 763–773.

[12] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. 2021. Detecting Filter List Evasion with Event-Loop-Turn Granularity JavaScript Signatures. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1715–1729.

[13] Tianyu Chen, Ding Li, Ying Zhang, and Tao Xie. 2025. JSimpo: Structural Deobfuscation of JavaScript Programs. *ACM Transactions on Software Engineering and Methodology* (2025).

[14] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 760–774.

[15] Yiu Wai Chow, Max Schäfer, and Michael Pradel. 2023. Beware of the Unexpected: Bimodal Taint Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*. ACM, 211–222.

[16] Yiu Wai Chow, Max Schäfer, and Michael Pradel. 2023. Beware of the unexpected: Bimodal taint analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 211–222.

[17] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 752–761.

[18] Fabio Ferreira and Marco Tulio Valente. 2023. Detecting code smells in React-based Web apps. *Information and Software Technology* 155 (2023), 107111.

[19] Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2024. Efficient static vulnerability analysis for javascript with multiversion dependency graphs. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 417–441.

[20] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

[21] Zhiyong Guo, Mingqing Kang, V.N. Venkatakrishnan, Rigel Gjomemo, and Yinzhi Cao. 2024. ReactAppScan: Mining React Application Vulnerabilities via Component Graph. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 585–599. https://doi.org/10.1145/3658644.3670331

[22] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 380–394.

[23] Yiheng Huang, Ruisi Wang, Wen Zheng, Zhuotong Zhou, Susheng Wu, Shulin Ke, Bihuan Chen, Shan Gao, and Xin Peng. 2024. SpiderScan: Practical Detection of Malicious NPM Packages Based on Graph-Based Behavior Modeling and Matching. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1146–1158.

[24] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5673)*. Springer, 238–255.

[25] Maryna Kluban, Mohammad Mannan, and Amr Youssef. 2022. On measuring vulnerable javascript functions in the wild. In *Proceedings of the 2022 ACM on Asia conference on computer and communications security*. 917–930.

[26] Maryna Kluban, Mohammad Mannan, and Amr Youssef. 2024. On detecting and measuring exploitable JavaScript functions in real-world applications. *ACM Transactions on Privacy and Security* 27, 1 (2024), 1–37.

[27] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*. USENIX Association, 121–134.

[28] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*. 1–6.

[29] Jesutofunmi Kupoluyi, Moumena Chaqfeh, Matteo Varvello, Russell Coke, Waleed Hashmi, Lakshmi Subramanian, and Yasir Zaki. 2022. Muzeel: assessing the impact of JavaScript dead code elimination on mobile web performance. In *Proceedings of the 22nd ACM Internet Measurement Conference, IMC 2022, Nice, France, October 25-27, 2022*. ACM, 335–348.

[30] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1509–1526.

[31] 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 *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society.

[32] Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. 2024. Reducing Static Analysis Unsoundness with Approximate Interpretation. *Proc. ACM Program. Lang.* 8, PLDI, Article 194 (June 2024), 24 pages. https://doi.org/10.1145/3656424

[33] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*. Citeseer, 96.

[34] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 143–160.

[35] Xinyue Liu and Lukasz Ziarek. 2023. PTDETECTOR: An Automated JavaScript Front-end Library Detector. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 649–660.

[36] Xinyue Liu and Lukasz Ziarek. 2025. PTV: Better Version Detection on JavaScript Web Library Based on Unique Subtree Mining,. In *ACM International Conference on the Foundations of Software Engineering (FSE)*.

[37] Zhonglin Liu, Yong Fang, Cheng Huang, and Yijia Xu. 2023. MFXSS: An effective XSS vulnerability detection method in JavaScript based on multi-feature model. *Computers & Security* 124 (2023), 103015.

[38] Benjamin Livshits and Emre Kiciman. 2008. Doloto: Code splitting for network-bound Web 2.0 applications. In *Proceedings of the 16th ACM SIGSOFT international symposium on foundations of software engineering*. 350–360.

[39] Francesco Logozzo and Manuel Fähndrich. 2008. On the relative completeness of bytecode analysis versus source code analysis. In *International Conference on Compiler Construction*. Springer, 197–212.

[40] Jingbo Lu, Dongjie He, Wei Li, Yaoqing Gao, and Jingling Xue. 2023. Automatic Generation and Reuse of Precise Library Summaries for Object-Sensitive Pointer Analysis. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 736–747.

[41] Magnus Madsen, Ondrej Lhotak, and Frank Tip. 2020. A Semantics for the Essence of React. In *European Conference on Object-Oriented Programming*.

[42] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*. IEEE, 569–580.

[43] Kedar S Namjoshi and Zvonimir Pavlinovic. 2018. The impact of program transformations on static program analysis. In *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25*. Springer, 306–325.

[44] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node.js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 29–41. https://doi.org/10.1145/3460319.3464836

[45] 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 *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. ACM, 736–747.

[46] Changhee Park, Hyeonseung Im, and Sukyoung Ryu. 2016. Precise and scalable static analysis of jQuery using a regular expression domain. In *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*. ACM, 25–36.

[47] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. 2018. ConflictJS: finding and understanding conflicts between JavaScript libraries. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 741–751.

[48] Amir Pnueli. 1981. The temporal semantics of concurrent programs. *Theoretical computer science* 13, 1 (1981), 45–60.

[49] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/tranco-a-research-oriented-top-sites-ranking-hardened-against-manipulation/

[50] Michael Pradel and Satish Chandra. 2021. Neural software analysis. *Commun. ACM* 65, 1 (2021), 86–96.

[51] Michael Pradel and Koushik Sen. 2018. DeepBugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 147:1–147:25.

[52] Yan Qin, Weiping Wang, Zixian Chen, Hong Song, and Shigeng Zhang. 2023. TransAST: A Machine Translation-Based Approach for Obfuscated Malicious JavaScript Detection. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network, DSN 2023, Porto, Portugal, June 27-30, 2023*. IEEE, 327–338.

[53] Jeremy Rack and Cristian-Alexandru Staicu. 2023. Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[54] Kunlun Ren, Weizhong Qiang, Yueming Wu, Yi Zhou, Deqing Zou, and Hai Jin. 2023. An empirical study on the effects of obfuscation on static machine learning-based malicious javascript detectors. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1420–1432.

[55] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61.

[56] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2021. Lossless, persisted summarization of static callgraph, points-to and data-flow analysis. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[57] Yijun Shen, Xiang Gao, Hailong Sun, and Yu Guo. 2025. Understanding vulnerabilities in software supply chains. *Empirical Software Engineering* 30, 1 (2025), 1–38.

[58] 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, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 1735–1746.

[59] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting taint specifications for JavaScript libraries. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South*

Korea, 27 June - 19 July, 2020. ACM, 198–209.

[60] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 83–95.

[61] Marcos Tileria, Jorge Blasco, and Santanu Kumar Dash. 2024. DocFlow: Extracting Taint Specifications from Software Documentation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[62] Alexi Turcotte, Ellen Arteca, Ashish Mishra, Saba Alimadadi, and Frank Tip. 2022. Stubbifier: debloating dynamic server-side JavaScript applications. *Empir. Softw. Eng.* 27, 7 (2022), 161.

[63] Rijnard van Tonder and Claire Le Goues. 2020. Tailoring programs for static analysis via program transformation. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 824–834.

[64] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C Rinard. 2021. Supply-chain vulnerability elimination via active learning and regeneration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1755–1770.

[65] Laurie Williams, Giacomo Benedetti, Sivana Hamer, Ranindya Paramitha, Imranur Rahman, Mahzabin Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, et al. 2024. Research directions in software supply chain security. *ACM Transactions on Software Engineering and Methodology* (2024).

[66] Yichen Xie and Alex Aiken. 2005. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 351–363.

[67] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 237–249.

[68] 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 *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association, 995–1010.