Increasing the Responsiveness of Web Applications by Introducing Lazy Loading

Anonymous Author(s)

***Abstract*—Front-end developers want their applications to contain no more code than is needed in order to minimize the amount of time that elapses between visiting a web page and the page becoming responsive. However, front-end code is typically written in JavaScript, the ubiquitous "language of the web", and tends to rely heavily on third-party packages. While the reuse of packages improves developer productivity, it is notorious for resulting in very large "bloated" applications, resulting in a degraded end-user experience. One way to combat such bloat is to *lazily load* external packages on an as-needed basis, for which support was added to JavaScript in 2020 when *asynchronous, dynamic* imports were added to the language standard. Unfortunately, migrating existing projects to take advantage of this feature is nontrivial, as the code changes required to introduce asynchrony may involve complex, non-local transformations.**

In this work, we propose an approach for automatically introducing lazy loading of third-party packages in JavaScript applications. Our approach relies on static analysis to identify external packages that can be loaded lazily and generates the code transformations required to lazily load those packages. Since the static analysis is unsound, these transformations are presented as suggestions that programmers should review and test carefully. We implement this approach in a tool called *Lazifier*, and evaluate *Lazifier* on 10 open-source front-end JavaScript applications, showing that each application was successfully refactored, reducing initial application size and load times in all cases. On average, for these applications, *Lazifier* reduces initial application size by 36.2%, initial load time by 29.7%, and unsoundness did not arise in any of these applications.

***Index Terms*—JavaScript, client-side, refactoring, static analysis, lazy loading, dynamic loading**

I. Introduction

In web application development, it is highly desirable to minimize the time it takes for an application to load and become responsive [1]–[4]. Therefore, developers generally aim to keep the size of their distribution as small as possible and rely on tools such as bundlers, minifiers, and tree-shakers [5]–[8] to minimize code size. Unfortunately, such tools are of limited use in scenarios where an application contains functionality that is (potentially) required, but not immediately on application startup. In such cases, responsiveness can be improved by loading the code associated with such functionality asynchronously, if or when its first use occurs.

In this work, we propose an approach for automatically refactoring applications to introduce lazy loading. We are targeting a specific scenario where the functionality to be loaded lazily is isolated in a third-party library that is imported by the application under consideration. Our approach relies on static analysis to identify packages that are only used in the context of event-handling code, as they are likely only needed conditionally (or at least not needed on startup). Then, for each of these packages, another static analysis establishes the extent of the code that needs to be modified to accommodate asynchronous, lazy loading of the package. Finally, a set of declarative rewrite rules specifies the code changes required to transform the application.

We implemented this approach in a tool called *Lazifier* that targets the JavaScript programming language (ECMAScript 2021). Similar to recent other refactoring tools [9]–[11], *Lazifier* employs unsound static analysis, so the proposed code transformations are presented as *suggestions* that programmers should review and test carefully before applying. In an experimental evaluation on 10 open-source client-side JavaScript applications, the code transformations proposed by *Lazifier* resulted in an average initial application size reduction of 36.2%, which caused applications to speed up initial load time by 29.7% on average. Furthermore, we found that the actual lazy loading of packages affected by the transformations incurs little overhead. Finally, despite the potential for unsoundness in the static analysis, we found that none of the transformations proposed by *Lazifier* for the 10 subject applications caused unwanted behavioral differences.

In sum, the contributions of this paper are as follows:

- A fully automated approach for identifying packages that can be loaded lazily, and a set of rewrite rules specifying how to refactor an application to load those packages lazily;
- An implementation of this approach in a tool called *Lazifier*, targeting the JavaScript programming language;
- An evaluation of *Lazifier* on 10 applications that suggests that *Lazifier* reduces initial application size (36.2%, on average) and load time significantly (29.7%, on average) with little overhead associated with dynamic loading.

A code artifact including *Lazifier* and a reproduction of the evaluation will be submitted for evaluation and made available as an artifact should this paper be accepted.

The remainder of this paper is organized as follows. First, the relevant background is covered in Section II, the problem is further motivated in Section III, the approach is described in-depth in Section IV (in which the implementation of our tool, *Lazifier*, is overviewed in subsections IV-D), followed by the evaluation in Section V, threats to validity in Section VI, and finally related literature is overviewed in Section VII before concluding in Section VIII.

II. Background

This section reviews JavaScript's mechanisms for asynchrony and importing modules.

A. Asynchronous JavaScript

JavaScript applications rely heavily on I/O operations, e.g., interaction with servers and user input handling. JavaScript does not support concurrency at the language level, and instead relies on a run-time model based on an *event loop* that enables it to perform operations asynchronously despite being single-threaded. Essentially, the event loop is a queue of function calls (i.e., callbacks) to be executed, which follow run-to-completion semantics; calling functions asynchronously has the effect of loading them onto the event loop. Once on the event loop, a callback is executed similarly to any other synchronous code. There are three major ways to build asynchronous JavaScript applications, reviewed in turn.

1) Event-Based Programming: This style of asynchronous programming relies on functions being registered as *listener callbacks* for specific events, which are called when the associated event is emitted. As an example, consider the following code snippet, which declares a function `onClick` that is then registered as a listener callback handling the `"click"` event:

```
function onClick(event) { /* handler logic */ }
document.addEventListener("click", onClick);
```

The call `document.addEventListener("click", onClick)` registers `onClick` as the callback to handle the `"click"` event on the `document` component of the web page. Later, when a user clicks on the page, the `"click"` event fires and a call to `onClick` is placed on the event loop.

2) Promises: ECMAScript 2016 introduced *promises* to the JavaScript language standard, which is a convenient abstraction for asynchronous programming.

a) **Creating promises:** Promises are created by invoking the `Promise` constructor, which takes as argument an *executor* function, itself taking 2 arguments:

```
const p = new Promise((resolve, reject) => {
  if (someCondition)
    resolve('success');
  else
    reject(404);
});
```

Initially, the promise is in *pending* state while the asynchronous operation is in progress. This promise can transition to the *settled* state one of two ways: it is *fulfilled* by invoking the `resolve` function, or *rejected* by invoking the `reject` function as shown above.

b) **Promise-Based Control Flow:** Callbacks can be registered as *reactions* on promises. For example, using the same promise `p` as above:

```
p.then(v => {
  console.log('Promise␣fulfilled␣with␣value:␣', v);
}).catch(e => {
  console.log('Error␣code:␣', e);
});
```

This code snippet first registers a reaction using the `then` method on `p`, which will be invoked if `p` is fulfilled; e.g., if `resolve('success')` was called in the body of `p`, the value `'success'` would be passed as an argument to the callback registered with `then`. The `catch` method registers a reaction that is invoked in the event that the promise was rejected; e.g., if `reject(404)` was called in the body of `p`, the value `404` would flow into the callback registered with `catch`.

c) **Synchronizing Promises:** As promises reflect asynchronous computations, in general, there is no guarantee on the order in which they will be settled. The promise library provides the `Promise.all` method to synchronize a list of promises: it accepts an array of promises and returns a single promise that is resolved with the array of values corresponding to the fulfilment of each promise, and the i^{th} value corresponds to the i^{th} promise. If any promise in the input array is rejected, `Promise.all` is rejected as well. For example, the following code snippet shows `Promise.all` synchronizing the fulfilment of three promises, and in the `then` reaction, `v[0]` corresponds to the value `p0` resolved with, `v[1]` with `p1`, and `v[2]` with `p2`.

```
Promise.all([p0, p1, p2])
    .then(v => { let total = v[0] + v[1] + v[2]; })
    .catch(e => { ... });
```

3) Async/Await: ECMAScript 2017 expanded JavaScript by introducing the `async` and `await` keywords to the language, which provide syntactic sugar on top of promises. First and foremost, `await` expressions are only allowed inside of `async` functions. The expression `await p` halts execution within the scope of an `async` function until the promise `p` is settled, at which point `await p` will return the value that `p` was resolved with. If `p` is rejected, `await p` will `throw` the value `p` was rejected with, which can be handled in a `try/catch`. This greatly simplifies asynchronous control flow, e.g., in the following snippet, a promise `p` is `await`-ed; if `p` resolves, the value it resolved with flows into the local variable `v`, and the function returns `v.toUpperCase()`; if `p` was rejected, the value it was rejected with would flow into `e` in `catch(e)`, at which point the error could be handled.

```
async function louder(p) {
    try {
      let v = await p; // => 'success'
      return v.toUpperCase();
    } catch (e) {
      // ...
    }
}
```

Importantly, a function that is declared as `async` *always* returns a promise that resolves with the value the function returns: if an `async` function f contains an expression `return e`, where `e` is a value of type T, then f returns an object of type $Promise\langle T\rangle$ that is resolved with the value `e`. To use the return value, one can `await` calls to the function; for example, consider this snippet using `louder` and `p` defined previously:

```
async function bar() {
    const a = await louder(p); // => 'SUCCESS'
}
```

Note: as of ECMAScript 2022, `await` expressions are also allowed at the top level (i.e., outside a function body), although it is a new language feature and is subject to unexpected behavior: e.g., if the `await`-ed promise is rejected outside of the context of a `try/catch`, the application crashes, and top-level awaits in the context of circular dependencies can cause a deadlock [12].

B. Importing Packages in JavaScript

As this work is concerned with lazily loading packages, we overview JavaScript's mechanisms for importing packages.

a) **require:** The traditional method of including external code in JavaScript is to use `require`, a function that dynamically *and synchronously* loads and executes the package matching the supplied name. Consider:

```
const xlsx = require("xlsx");
function importXLSXData(data) {
  const contents = xlsx.read(data, {...});
  // do stuff with the contents.
}
```

First, the `"xlsx"` package is imported at runtime and saved in the `xlsx` global variable. `"xlsx"` exports a `read` function to convert raw spreadsheet data, and so inside `importXLSXData` the exported function is referenced as a property on the `xlsx` object (`xlsx.read`). Notably, `xlsx` contains *the entire* package code.

b) **static import:** ECMAScript 6 introduced the static `import` declaration as an alternative to the dynamic `require`. These `import` statements must be at the top level, all bindings must be identifiers, and the package name must be a string literal (this makes them easier to analyze statically); e.g., the statement `import * as xlsx from "xlsx"` imports the entire `"xlsx"` package. A major advantage of static `import` statements is that a developer can specify which parts of a package they want to import; e.g., in the following snippet, the `read` function exported by `"xlsx"` is imported directly:

```
import { read } from "xlsx";
function importXLSXData(data) {
  const contents = read(data, {...});
  // do stuff with the contents.
}
```

The strict nature of these static import statements allows static analyzers to more effectively determine the extent to which an application exercises the code it imports, which can sometimes lead to smaller distributions—this is called *tree-shaking* [7], [8]. Unfortunately, JavaScript's high degree of dynamism limits the power of these static analyses [13]–[15], preventing tree-shaking from removing much code.

c) **dynamic import:** Static imports are syntactically rigid by design, and so ECMAScript 2020 introduced a dynamic, *asynchronous* `import` function. The `import` function accepts a string containing the name or path of a package as an argument and returns a promise. That promise can either resolve with an object containing all the exported functions and objects, or be rejected if the package cannot be found. This syntax is especially useful for importing large or rarely used external packages, since they will not be bundled with the rest of the application. This can often result in smaller initial application sizes and potentially faster load times. The following code snippet illustrates how to dynamically import `"xlsx"` only in the context of `importXLSXData`:

```
async function importXLSXData() {
    const xlsx = await import("xlsx");
    const data = xlsx.read(...);
}
```

Note that if a dynamic import for a particular package is encountered more than once, the package is loaded only once, and all subsequent invocations resolve to the same cached instance. Thus, even if `import("xlsx")` or `importXLSXData` is invoked multiple times, the `"xlsx"` package will be loaded only once and served to all subsequent invocations.

III. Lazy Loading

To illustrate our approach, consider an open-source JavaScript application that displays a list of recent movies to users, complete with information about them (**Movies-web-ui** [16]). Users can filter the list of movies and, optionally, export their filtered selection. The code snippet in Fig 1**(a)** is taken directly from **Movies-web-ui**, showing how they implement an "export" button and associated functionality. Note that this application uses a few external packages: `React`, an extremely popular UI framework for JavaScript, `file-saver` [17] for saving files, and `xlsx` [18] for dealing with spreadsheet-like data. The file exports a function `exportCSV` that creates a JSX [1] button component (lines 59-63). The `"click"` event handler associated with this button (lines 60-61) eventually calls the `exportToCSV` function (lines 49-57), which leverages the `xlsx` package to convert a JSON file representing the user's selection to a sheet (line 52), and `file-saver` to save the selection to a file (line 56).

Crucially, in this example, the `xlsx` and `file-saver` packages are only needed to implement the export functionality and are not useful to users that simply want to browse the list of movies. It should also be noted that the references to these packages on lines 52, 54, and 56 are the only references to these packages in the entire application.

In such cases, it is desirable to load packages lazily, so that users who do not use the associated functionality do not incur the overhead of loading code that they will not use. The code snippet in Fig 1**(b)** depicts how this can be achieved, and code changes are highlighted. First, note the lack of static imports to `xlsx` and `file-saver`, and the inclusion of dynamic imports to the packages instead (lines 74-75).

The call `import('file-saver')` on line 74 creates a promise that is resolved with an object representing the `file-saver` package. Once the loading of the package has been completed, the `await` on the same line ensures that this object can be assigned to the local variable `fileSaver`. Recall that `await` expressions are only allowed in the context of `async` functions, so the `exportToCSV` function must gain the `async` keyword (line 73). This changes the return type of `exportToCSV` to $Promise\langle JSX\rangle$, so all call sites to this function should be `await`-ed to ensure that application behavior remains unchanged. In particular, an `await` is added at the call to `exportToCSV` on line 85. This new `await` requires the surrounding function to be made `async` as well (line 84), at which point we have reached a context that implicitly handles asynchrony: callbacks that serve as event handlers are not expected to return anything, so no further transformations are required once they are made `async`.

[1] JSX is a type provided by React that closely matches HTML, allowing programmers to easily construct HTML-like objects in their JavaScript code.

```javascript
import React from 'react';
import * as fileSaver from 'file-saver';
import * as xlsx from 'xlsx';

export const exportCSV = ({csvData, fileName}) => {
  const fileType = '...';
  const fileExtension = '.xlsx';
  const exportToCSV = (csvData, fileName) => {


    const ws = xlsx.utils.json_to_sheet(csvData);
    const wb = {Sheets: {...}, SheetNames: [...]};
    const buffer = xlsx.write(wb, {...});
    const data = new Blob([buffer], {type: fileType});
    fileSaver.saveAs(data, fileName + fileExtension);
  }
  return (
    <button className="export"
      onClick={(e) =>
        exportToCSV(csvData,fileName)}>
      Export
    </button>
  )
}
```

(a)

```javascript
import React from 'react';
// this import was removed
// this import was removed

export const exportCSV = ({csvData, fileName}) => {
  const fileType = '...';
  const fileExtension = '.xlsx';
  const exportToCSV = async (csvData, fileName) => {
    const fileSaver = await import('file-saver');
    const xlsx = await import('xlsx');
    const ws = xlsx.utils.json_to_sheet(csvData);
    const wb = {Sheets: {...}, SheetNames: [...]};
    const buffer = xlsx.write(wb, {...});
    const data = new Blob([buffer], {type: fileType});
    fileSaver.saveAs(data, fileName + fileExtension);
  }
  return (
    <button className="export"
      onClick={async (e) =>
        await exportToCSV(csvData,fileName)}>
      Export
    </button>
  )
}
```

(b)

Fig. 1. Excerpt of a client-side application which uses xlsx: (a) version with static import (b) version with dynamic import

This simple refactoring reduces the amount of code that is loaded by over 30% (from 1.4mb to 0.96mb), and improves the initial load time of the application by just under 50% (from 517ms to 286ms, averaged over 10 runs). If the user *does* want to export their selection, the packages are loaded rather quickly (0.11s), and the total amount of code loaded by the application is 1.4mb, i.e., the same as the original size.

There are certain additional complexities that the above example only hinted at. For instance, when making a function `async`, *all* call sites to the function must be `await`-ed, no matter where they are. This can cause a cascade of transformations that may not be localized to a single file. Further, certain code patterns need to be modified to accommodate `async` functions (e.g., the expression `someArray.forEach(f)` is blocking if the callback `f` is synchronous, but non-blocking if `f` is `async`). In the next section, we describe these complexities and present our approach to automatically detecting packages that can be loaded lazily, and specify the code transformations required.

IV. Approach

Our approach for automatically refactoring applications to introduce lazy loading consists of the following three steps:

1) *Determine packages that are only used in the context of event handlers;*
2) *Confirm which of these can be loaded lazily, and identify the code transformations required;*
3) *Enact the transformation.*

For (1), we propose a fully automated static analysis to detect which packages are *only* used in the context of event handling code and that therefore are not initially needed by the application. For (2), another static analysis determines all of the functions containing references to a given lazy loading candidate. Each of those functions will require a dynamic, *asynchronous* import of the package, which will require several other code transformations to support the now asynchronous import. If any of these transformations are not possible, the lazy loading candidate is discarded. Finally, for (3) we propose a set of declarative rewrite rules describing the code changes required to refactor the application to lazily load the package. Each of these phases is described in turn.

Soundness. We assume that the static analyses used in steps 1) and 2) are potentially *unsound*, because static analysis for JavaScript that is simultaneously sound, precise, and scalable is well beyond the state-of-the-art due to the dynamism inherent to the language [13]–[15]. This means that the transformations proposed by the approach may not preserve behavior, and should be carefully reviewed by a programmer, similar to the approach taken by other refactoring tools for JavaScript [9]–[11]. In Section V, we investigate the degree to which this unsoundness causes behavioral differences.

A. Identify Candidate packages for Lazy Loading

To identify packages that should be loaded lazily, we provide a fully-automated analysis that detects packages that are only used in the context of event-handling code. Given a call graph for an application, this analysis identifies functions that are supplied to event-handling mechanisms (e.g., registered as "on-click" attributes of HTML elements, or registered as event listeners), and determines all of the functions that are (transitively) called from those handlers. If *all* references to a package are in this list of functions, then it is flagged as being a candidate for lazy loading. This list of event handlers is:

- functions passed to `onClick` or other `on` or `click` events on JSX and HTML components, including functions identified using string representations of their name;
- any code snippets included in an event handler attribute (e.g., code in the `onClick` event of an HTML element);

- functions passed as callback arguments to event handlers (e.g., `reader.on('load', callback)`);
- functions assigned to properties of the `window` object that represents the Document Object Model (DOM).

B. Validate and Determine Transformations Required

To successfully load a package p lazily, all static imports to p must be removed, and functions containing references to p must be refactored to load the package dynamically. This involves removing static `import ... from 'p'` statements and inserting dynamic `import('p')` expressions where appropriate. The expression `import('p')` yields a promise that eventually resolves with the content of the package `'p'`. While that promise is pending, the current context that depends on the package should not proceed, and `await`-ing that call will suspend execution until the promise is resolved. Then, if assigning the `await`-ed import to a variable (e.g., `let x = await import('p')`), the package itself will be stored in `x` and execution can resume.

Now, `await` expressions are only allowed inside of functions marked as `async`, but making a function `async` changes its return type to $Promise\langle T\rangle$, where T is the function's original return type. To preserve existing application behavior, all call sites to this function will need to be `await`-ed, which itself requires more functions to be made `async` and more call sites to be `await`-ed, and so on. It is imperative that *all* call sites to newly `async` functions be `await`-ed, else program behavior will be affected; this means that the transformation is *all or nothing proposition*, and if any call sites cannot be `await`-ed, we must abandon the entire transformation, and discard p as a lazy loading candidate.

Algorithm 1: Validating p and building S_{async}

Data: p: a package being imported dynamically
Data: CG: the call graph of the program

1 let S_{async} := {};
2 let F := [functions referencing p];
3 **while** F *not empty* **do**
4 let f := select and remove a function from F;
5 **if** f *not visited* **then**
6   **if** f *is a reaction* **or** f *is argument to promise constructor* **or** f *registered as event handler* **then**
7    $S_{async} := S_{async} \cup \{f\}$;
8   continue;
9   let C_f := callers of f in CG;
10   **if** f *is constructor* **or** $c \in C_f$ *is top level* **or** f *returns promise* **then**
11    $S_{async} := \{\}$;
12   break;
13   $S_{async} := S_{async} \cup \{f\}$;
14   $F := F \cup C_f$;
15   mark f as visited;
16 **return** S_{async};

Algorithm 1 describes the process of creating the set S_{async} of functions needing to be made `async` while validating the transformation. As inputs to the algorithm, the package p is supplied along with the call graph CG of the program. First, S_{async} is initialized as the empty set (line 1), and the list F of functions yet to be processed is initialized with all functions containing references to the package p (line 2). The main loop (lines 3-15) iterates through functions $f \in F$ that have not yet been visited. First, lines 6-8 describes a special case where a function to be made asynchronous is already in a context that handles asynchrony, in which case no further transformations are required. Then, all callers of the function f are obtained from the call graph (line 9). Lines 10-12 *validates* the transformation by identifying situations that cannot support asynchrony. First, constructors cannot be `async`. Second, if f is called at the top level of the application, there is no sense in lazily loading p as the dynamic import would be executed on application startup anyway. (Also, top-level `await` expressions are only supported as of ECMAScript 2022.) Third, if f already returns a promise, the programmer is likely using it accordingly and may not want calls to it to be `await`-ed, and so it should not be transformed. In such cases, the transformation is rejected and p is not loaded lazily. If f passes this check, then f is added to S_{async}, all of f's callers are added to the list F of functions left to process, and f is marked as visited; analysis continues until F is exhausted.

C. Code Transformations

The application can be refactored to lazily load package p once the set S_{async} of functions that need to be made `async` is known. Several transformations are required to handle the transition to asynchronous imports, specified as declarative rewrite rules in Figure 2. The figure depicts simplified, idealized JavaScript to illustrate the salient details of the transformation. We will describe them one by one next.

ASYNC-FUNCTION: This transformation is simple: if a function `f` is in the set S_{async} of functions that need to be made `async`, the function definition gains the `async` keyword.

ASYNC-CALL: All potential calls to a function $\texttt{f} \in S_{async}$ need to have `await` expressions inserted before the call.

FOREACH-FOROF: The expression `arr.forEach(f)` calls the callback `f` on each element of `arr`, and importantly *returns nothing*, i.e., `forEach` is type *void*. If `f` were made asynchronous, the call to `forEach` would not wait for all of the asynchronous calls to resolve, and execution would simply continue past the call. In the event that `f` contains no `return` statements, the body `B` of `f` is made into the body of a `for ... of` loop that iterates over the elements of the array (the loop iterator `a` is chosen to match the argument name of `f`).

FOREACH-MAP: In the event that `f` *does* contain a `return` statement, conversion to a `for ... of` loop is not possible. Instead, the `forEach` is transformed into a `map`, and the call to `map` is surrounded in an `await`-ed `Promise.all` to ensure

$$\frac{\texttt{f} \in S_{async}}{\texttt{fun f(A) \{B\}} \longrightarrow \texttt{async fun f(A) \{B\}}} \quad \text{(ASYNC-FUNCTION)}$$

$$\frac{\texttt{f} \in S_{async} \qquad \texttt{g} \text{ can resolve to } \texttt{f}}{\texttt{g(args)} \longrightarrow \texttt{await g(args)}} \quad \text{(ASYNC-CALL)}$$

$$\frac{\texttt{f} \in S_{async} \qquad B \text{ body of } \texttt{f} \\ \text{no returns in } B \qquad a = \text{the single argument of } \texttt{f}}{\texttt{arr.forEach(f)} \longrightarrow \texttt{for}([\texttt{i}, a] \texttt{ of arr.entries())} \{B\}} \quad \text{(FOREACH-FOROF)}$$

$$\frac{\texttt{f} \in S_{async} \qquad B \text{ body of } \texttt{f} \\ \text{returns in } B}{\texttt{arr.forEach(f)} \longrightarrow \texttt{await Promise.all(arr.map(f))}} \quad \text{(FOREACH-MAP)}$$

$$\frac{\texttt{f} \in S_{async}}{\texttt{arr.map(f)} \longrightarrow \texttt{await Promise.all(arr.map(f))}} \quad \text{(AWAIT-MAP)}$$

$$\frac{p \in P_D \qquad v_0, ..., v_n \textbf{ ref } p \in \texttt{B} \\ \texttt{dynImp} := \texttt{const } p_{name} = \texttt{await import}(p) \\ \texttt{decl}_k := \texttt{const } v_k = p.v_k^{name} \quad \forall k \in 0, ..., n}{\texttt{fun f(A) \{B\}} \longrightarrow \texttt{fun f(A) \{dynImp; decl}_\texttt{0}\texttt{; ... decl}_\texttt{n}\texttt{; B\}}} \quad \text{(INSERT-DYNAMIC-IMPORT)}$$

$$\frac{\texttt{x} \in S_{async} \qquad \texttt{f}_\texttt{B} := \texttt{async ()} \Rightarrow \texttt{\{B\}}}{\texttt{get x() \{B\}} \longrightarrow \texttt{get x() \{return f}_\texttt{B}\texttt{();\}}} \quad \text{(GETTER)}$$

Fig. 2. Transformation rules for introducing lazy loading and necessary code changes to support newly introduced asynchrony.

that all of the asynchronous callbacks fully execute before continuing.

AWAIT-MAP: Similar to the previous rule, if a callback passed to `map` is to be made asynchronous, the `map` is surrounded in an `await`-ed `Promise.all`.

INSERT-DYNAMIC-IMPORT: If a function `f` contains references $(v_0, ..., v_n)$ to a package p that is to be made dynamic $(p \in P_D)$, a dynamic import to the package p is created (`const` p_{name} = `await import`(p)), where p_{name} will serve as a reference to the package in this scope. Then, declarations are created for each $v_k \in v_0, ..., v_n$ extracting the relevant component v_k^{name} from the import p_{name}. The dynamic import and associated declarations are then inserted at the beginning of the function body.

GETTER: Getters present a special case as they cannot be made asynchronous. A new asynchronous function $\texttt{f}_\texttt{B}$ is created with the body `B` of the getter `x`. The body of `x` is then replaced with a return to the call to $\texttt{f}_\texttt{B}$—callers of `x` will `await` calls to it, and so the promise returned by $\texttt{f}_\texttt{B}$ can be `await`-ed then.

The code transformation in the motivating example was determined automatically using this approach, and involved applications of rules ASYNC-FUNCTION, ASYNC-CALL, and INSERT-DYNAMIC-IMPORT. Fig. 3 shows small code examples depicting the transformations associated with the other rules: Fig. 3**(a)** and **(b)** shows rule FOREACH-FOROF, Fig. 3**(c)** and **(d)** shows rule FOREACH-MAP, Fig. 3**(e)** and **(f)** shows rule AWAIT-MAP, and finally Fig. 3**(g)** and **(h)** shows rule GETTER.

D. Implementation

This approach is implemented in a tool called *Lazifier*. All static analyses are built in CodeQL [19], including data flow analyses required to detect uses of imported packages and call graph construction. All call graphs were obtained through CodeQL's own static call graph construction algorithm for JavaScript [20], which is unsound. The code transformation is built in JavaScript using Babel [21] to parse code, manipulate ASTs, and emit transformed code.

V. EVALUATION

We pose the following research questions in order to evaluate the approach proposed in this paper:

RQ1) How does lazy loading affect the size and initial load time of applications?

RQ2) How often does the transformation introduce unwanted behavioral changes?

RQ3) How much code is loaded lazily, and how quickly is it loaded?

RQ4) How many code changes are required to support lazy loading?

RQ5) What is the running time of *Lazifier*?

```
arr.forEach((e) => {
  if (e)
    foo();
  else
    bar();
});
```

(a)

```
for([i, e] of arr.entries()) {
  if (e)
    await foo();
  else
    bar();
}
```

(b)

```
arr.forEach((e) => {
  if (e)
    return foo();
  else
    return bar();
});
```

(c)

```
await Promise.all(arr.map(async (e) => {
  if (e)
    return await foo();
  else
    return await bar();
}));
```

(d)

```
arr.map((e) => {
  if (e)
    foo();
  else
    bar();
});
```

(e)

```
await Promise.all(arr.map(async (e) => {
  if (e)
    await foo();
  else
    await bar();
}));
```

(f)

```
const o = {
  x : 1,
  get y() {
    return foo(x);
  }
}

o.y;
```

(g)

```
const o = {
  x : 1,
  get y() {
    return (async () => {
      return await foo(x);
    })();
  }
}

await o.y;
```

(h)

Fig. 3. Code showing the before and after of applying select rewrite rules: **(a)-(b)** shows FOREACH-FOROF, **(c)-(d)** shows FOREACH-MAP, **(e)-(f)** shows AWAIT-MAP, and **(g)-(h)** shows GETTER.

Experimental Methodology

To answer these research questions, we first compiled a list of 10,000 open-source client-side JavaScript applications by scraping GitHub for repositories that had JavaScript UI frameworks stated as dependencies. Then, we ran the `npm-filter` [31] tool to identify projects for which *Lazifier* identified at least one package as a candidate for lazy loading (yielding 998 projects). We manually inspected projects in this list until we found 10 that could be successfully installed, started, and interacted with. The vast majority of JavaScript projects on GitHub suffer from installation errors (e.g., developer-specified dependencies no longer work), build errors (e.g., build configurations that are only valid for certain operating systems/environments), or environment errors (e.g., many client-side applications rely on external servers that are inaccessible). Since we wanted to have a high degree of confidence in our understanding of our subject applications, we expended considerable effort finding applications that suffered from none of these aforementioned issues.

To answer **RQ1**, we first determine the original application's initial size using the "bytes transferred" metric from Chrome DevTools' [32] "Network" tab on a hard refresh of the application page, and then apply the transformation and similarly determine the initial size of the transformed application. To time the initial application load, we again leverage the Chrome DevTools' "Network" tab, and note the "Load" time field upon performing a hard refresh—we note this time pre- and post-transformation, and collect and average 10 load times.

To answer **RQ2**, we manually interacted with each application to determine how to make it execute code from packages that were flagged to be loaded lazily, then applied the transformation and repeated the interaction, manually ensuring that the application behavior was unchanged.

To answer **RQ3**, we identify how to trigger each of the dynamic imports (in the same manner as in **RQ2**), and note the size of the code chunk transferred when doing so through the Chrome DevTools' "Network" tab (again consulting the "bytes transferred" metric), and note the time taken to transfer that chunk through the "Load" time field.

To answer **RQ4**, we configured *Lazifier* to: display which packages were flagged to be loaded lazily, display the dynamic import statements that were added to the program, and log the code transformations it was applying.

And finally, to answer **RQ5**, we used the Unix `time` utility to time the execution of *Lazifier* on each application. To run *Lazifier*'s analyses, a CodeQL database must be built for the project, and so we used the time utility to time the CodeQL database build for each project.

All measurements were taken on a 2016 MacBook Pro running Catalina 10.15.7, with a 2.6GHz Quad-Code Intel

TABLE I

INFORMATION ABOUT SUBJECT APPLICATIONS. THE FIRST ROW READS: *the first application is called **upoint-query-builder** from* **Harinathlee**, *and commit hash* `f9aa0f1` *was used for the evaluation;* ***upoint-query-builder*** *has 10,341 lines of code. The initial size of the application is 0.84mb, reduced to 0.61mb after loading modules lazily, corresponding to a 27.4% size reduction. The size of the application once modules are loaded dynamically is 0.84mb. It took 201s to run Lazifier on this project, which required an additional 28s to build the CodeQL database.*

Project Name	Commit Hash	LOC	Initial Size (mb) Before	Initial Size (mb) After	Size Reduction	Expanded Size (mb)	Tool Run Time (s)	QLDB Time (s)
Harinathlee/upoint-query-builder [22]	f9aa0f1	10,341	0.84	0.61	27.4%	0.84	201	28
sadupawan1990/excelreader [23]	4a5f9cb	9,733	4.8	3.4	29.2%	4.8	187	44
fahimahammed/task [24]	b641bc0	9,747	0.94	0.48	48.9%	0.94	180	36
hongtaodai/react-excel [25]	2d59e85	9,685	1.9	1.5	21.1%	1.9	178	33
Abhishek312s/Movies-web-ui [16]	58904a3	9,789	1.4	0.96	31.4%	1.4	180	35
vishumane/ExcelSheet_Validation_Reactjs [26]	f38cb9e	9,942	0.90	0.40	55.6%	0.90	181	35
thewca/scrambles-matcher [27]	1de93f7	11,304	1.1	0.83	24.5%	1.1	188	37
hoverGecko/timetable [28]	0fa8527	9,932	0.60	0.38	36.7%	0.60	314	80
Akalay27/workday-schedule-exporter [29]	97ca596	9,718	0.90	0.44	51.1%	0.90	186	35
ultimateakash/react-excel-csv [30]	18c6d97	9,779	0.85	0.62	27.1%	0.85	206	34
				Average Size Reduction:	36.2%		**Average Run Time:**	240

Code i7 processor and 16GB RAM. We used Google Chrome version 112.0.5615.137 (Official Build) (x86_64) in incognito mode. Next, we respond to each of the **RQ**s in turn.

RQ1: *How does lazy loading affect the size and initial load time of applications?*

Lazifier's transformation leverages ECMAScript 2020's ability to load packages on demand: If all static imports to a package are replaced with dynamic imports, the JavaScript runtime dynamically fetches the package when a dynamic import is executed, and the package is not included in the application at start time. The initial application size is reported in columns **Initial Size (mb) Before** and **After** in Table I, corresponding to the size of the applications pre- and post-refactoring. We note significant size reduction across all applications (36.2% on average), as high as 51.6%.

While smaller applications are desirable in and of themselves, we investigate the degree to which this size reduction hastens the initial load time of refactored applications. Averages of 10 load times are reported in Fig. 4, with three columns for each subject application, the first two of which are relevant here: the first column corresponds to the load time pre-refactoring, and the middle column to the load time post-refactoring. We find statistically significant (T-test, two-tailed, 95% confidence) reductions in initial load time in all cases, with an average speedup of 29.7%, as high as 47.5%.

The size of refactored applications is smaller in all cases, which translates to a statistically significant reduction in application start times.

RQ2: *How often does the transformation introduce unwanted behavioral changes?*

Since the approach presented in this paper relies on unsound static analysis, the transformations suggested by *Lazifier* are not guaranteed to preserve application behavior. In our subject applications, *Lazifier*'s refactorings caused 15 packages to be loaded lazily, introducing 21 dynamic imports to those packages, requiring 47 other transformations (i.e., applications of a rewrite rule). We manually interacted with the applications and ensured that all transformed code was exercised, and found no behavioral differences introduced by the transformation.

For the 10 subject applications under consideration in this evaluation, there was no evidence of behavioral differences due to unsoundness in the static analysis.

RQ3: *How much code is loaded lazily, and how quickly is it loaded?*

When a package is loaded dynamically, the application asynchronously fetches package code and executes it, making the package available. Dynamically loading packages may result in a larger total application size, since dynamic imports load the entire package code (so no tree-shaking can be done as in the case of static imports). The total expanded size of each application is reported in column **Expanded Size (mb)** in Table I. Interestingly, we note that the total size of applications after dynamic loading is always the same as the initial size without refactoring, suggesting that tree-shaking is not an effective technique at reducing the size of imported packages.

We also noted the time taken to perform this transfer, reported in Fig. 4, specifically the third column ("dynamic") in each set of three. The transfer is small relative to initial load times in all cases (85.8ms on average), though note that we do not simulate latency in this test, and assuredly transferring data over a network would incur overhead related to latency.

The total size of the code loaded by the refactored applications (including lazily loaded packages) is comparable to the total size of the original applications, and dynamically loading packages is generally not noticeable.

RQ4: *How many code changes are required to support lazy loading?*

Since *Lazifier* suggests code changes that should be vetted carefully by programmers, it would be helpful if the extent

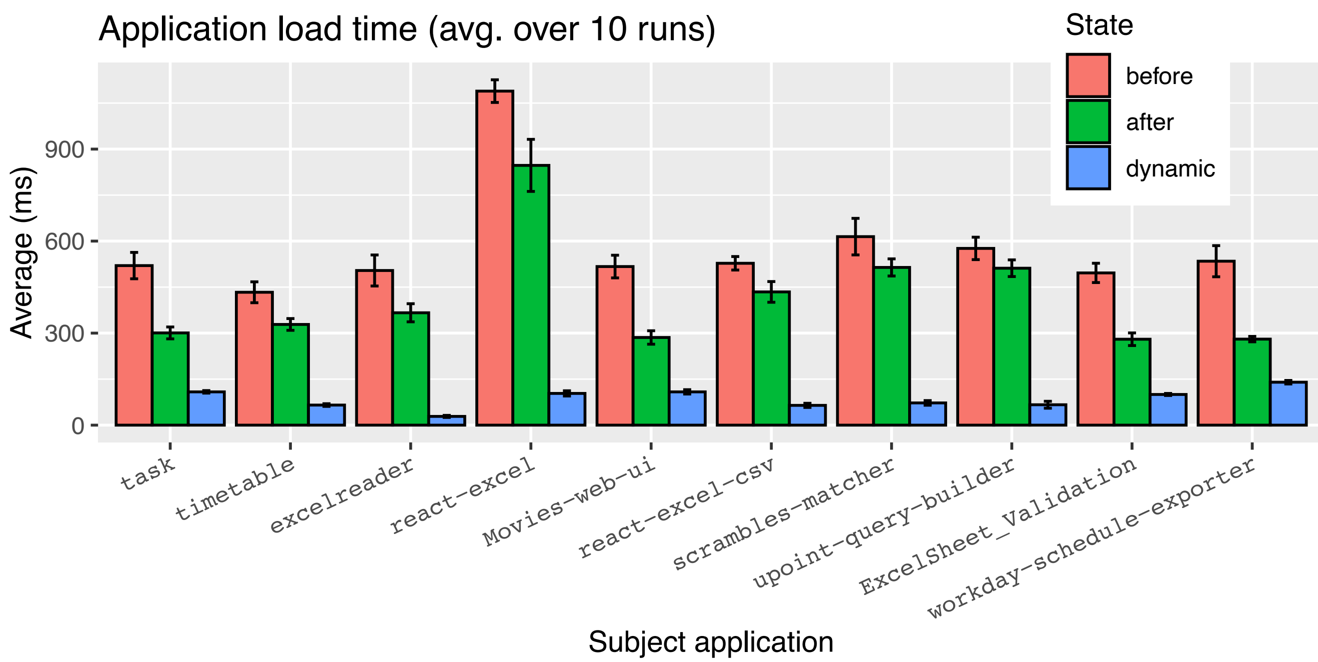

Fig. 4. Load times for each subject application are depicted in this plot, with a set of three columns for each application. In each set, three times are presented: first, the time taken pre-refactoring (before), then after refactoring (after), and finally the time taken to dynamically load all packages (dynamic). These are averages over 10 runs, and error bars indicate +/- one standard deviation.

of the transformations required was small and manageable. Table II lists information about the code transformations suggested by *Lazifier* in each subject application, namely how many packages could be loaded lazily (column **# Imps. Removed**), how many dynamic import statements were required to lazily load the packages (column **# Dyn. Imps.**), and finally how many applications of other rewrite rules were necessary to support lazily loading the packages (column **# Trans. Changes**). All cases required few code transformations, at most 15 for **upoint-query-builder** (the number of changes including added dynamic imports), with a median of 6 changes (again including added dynamic imports) per application, which should be manageable for a developer to review.

The number of code changes suggested by *Lazifier* is small, so the effort needed by programmers to review these changes is manageable.

TABLE II
INFORMATION ABOUT CODE TRANSFORMATIONS. THE FIRST ROW READS: *in* **upoint-query-builder**, *2 packages were loaded dynamically instead of statically; 3 dynamic import statements were added, and 12 applications of other rewrite rules were required to support the transition.*

Project Name	# Imps. Removed	# Dyn. Imps.	# Trans. Changes
upoint-query-builder	2	3	12
excelreader	1	1	2
task	1	1	2
react-excel	1	1	2
Movies-web-ui	2	2	5
ExcelSheet_Validation_Reactjs	2	3	7
scrambles-matcher	1	2	4
timetable	1	2	4
workday-schedule-exporter	3	4	6
react-excel-csv	1	2	3
In total:	15	21	47

RQ5: *What is the running time of Lazifier?*

The time taken to run *Lazifier* is reported in column **Tool Run Time (s)** of Table I. This includes the time to run the static analyses and also transform the application, though the transformation itself runs extremely quickly. The time to build the CodeQL database is reported in column **QLDB Time (s)** in Table I: this is a fixed cost once per project, and can be reused by other CodeQL queries.

The run time of *Lazifier* is 240s on average, demonstrating its suitability for practical use.

VI. THREATS TO VALIDITY

The technique presented in this paper was inspired by the work of Gokhale et al. [11], and suffers similar threats to validity. Namely, the code transformations proposed by our approach are unsound and are not guaranteed to preserve program behavior. There are many reasons for losses of soundness, e.g., the static analyses that build call graphs are unsound, and our technique introduces asynchrony to applications which may cause data races. In a sense, this unsoundness is inevitable as JavaScript is a highly dynamic language not amenable to sound static analysis. Nevertheless, in our evaluation we found that *Lazifier* proposed no behavior-altering transformations in spite of this unsoundness.

Beyond this, it is possible that our set of subject applications may not be representative. To mitigate this, we selected our subject applications from a list of client-side JavaScript applications sampled essentially randomly from GitHub. We did prune this list such that we could build and run the applications to evaluate the effectiveness of our technique, but believe that our random initial selection of projects mitigates risk of bias.

VII. Related Work

Broadly, this work is concerned with refactoring web application source code to lazy load libraries that are only conditionally required. Software debloating is a related area of research focused on trimming unused functionality from applications and has many applications in security, particularly when unused code is removed from applications. Also, the refactoring proposed in this work introduces asynchrony to an application, which is another well-studied area of research.

Debloating and Lazy Loading: Software debloating is well-studied. Many applications contain far more code than is required, commonly referred to as "dead code", and the study of debloating is the study of how to determine and safely remove this dead code. Besides increasing application size, dead code is undesirable as it increases the "attack surface" of an application, i.e., more code provides more opportunities for an attacker to take advantage of a system. For example, Bhattacharya et al. [33] studies situations where functions accumulate more features than are strictly necessary, yielding poor performance when spurious functionality is not needed. Koo et al. [34] propose configuration-driven software debloating, where application configurations are linked with feature-specific libraries, and libraries are only loaded when the appropriate configuration criteria are met. This is a semi-automated process, and the code itself is not changed. Doloto [35] proposes an approach that leverages developer-supplied application traces to automatically refactor applications to load entire "routes" lazily, only when they are needed; their approach performs dynamic loading synchronously, which is disallowed in the modern web standard. Soto et al. [36] propose an approach to automatically specialize Java dependencies according to how they are used by the application's test suite, and Sharif et al. [37] propose a technique that leverages constant value configuration data to specialize applications.

Some recent work has been concerned with debloating JavaScript applications. Stubbifier [38], for example, leverages an application's test suite to determine "probably unused" code and replace this code with small *stubs* that can dynamically fetch and execute the code if it was actually needed. Stubbifier cannot debloat client-side applications (which, incidentally, rarely have test suites). Malavolta et al. [39] propose a technique to debloat client-side JavaScript applications with various levels of optimization; first, dead code is determined by consulting a call graph of the application, and one of the optimization levels proposed in the work replaces dead code with snippets to load the code lazily. Vasquez et al. [40] propose a technique that flags external library functions as being potentially dead, and removes them once a programmer confirms that they are truly unused. These pieces of work are concerned with removing *unused* functionality, and often lazily loading the dead code if they were wrong about the code being dead, whereas our approach removes *conditionally* used functionality, and none of these tools would not remove the packages identified by our approach as they are used in the application. In a sense, these approaches are complementary.

Refactoring to Introduce Asynchrony: Loading packages lazily must be done asynchronously on the web, as blocking I/O operations are prohibited in the modern web standard. Thus, the refactoring proposed in this paper also refactor the applications to be asynchronous w.r.t. the lazily loaded packages. There are numerous pieces of related work in this area. Most closely related is Desynchronizer [11], which refactors JavaScript applications to use asynchronous APIs where synchronous APIs were once used. Other research loosely in this space includes work by Khatchadourian et al [41] on automatically parallelizing Java 8 streams, by Dig et al. [42] to parallelize Java loops, by Wloka et al. [43] on refactoring applications to be reentrant, by Dig et al. [44] for leveraging concurrency APIs to transform sequential code. Essentially, making synchronous code asynchronous is a difficult problem; in our work, we introduce *just enough* asynchronous constructs to allow for packages to be lazily loaded.

There is also a related wide body of work on *understanding* asynchronous applications, e.g., work by Alimadadi et al. [45] on understanding event-based asynchrony in JavaScript applications, on understanding asynchrony on the entire application stack [46], and on understanding the effects of DOM-sensitive changes [47]. This is complementary to our work, as *Lazifier* presents refactorings (that introduce asynchrony!) as suggestions to be vetted by programmers.

VIII. Conclusion

Client-side developers want to minimize the amount of time users need to wait for a web application to load and become responsive. Existing tools such as bundlers, minifiers, and tree-shakers focus on eliminating unused functionality and reducing code size, but do not address scenarios where an application contains functionality that is (potentially) required, but not immediately when the application starts up. In such cases, responsiveness can be improved by loading such functionality lazily. We have presented an approach for detecting situations where an entire library can be loaded lazily. The approach uses static analysis to identify packages that are only used in the context of event handling and to compute the changes that must be made to the code to accommodate lazy loading. A set of declarative rewrite rules specifies the code changes required to transform the application.

This approach was implemented in a tool called *Lazifier*, and evaluated on 10 open-source client-side JavaScript applications. In all cases, *Lazifier* successfully refactored the applications, resulting in an average initial application size reduction of 36.2%, which caused applications to start up 29.7% more quickly on average.

References

[1] D. F. Galletta, R. M. Henry, S. McCoy, and P. Polak, "Web site delays: How tolerant are users?," *J. Assoc. Inf. Syst.*, vol. 5, no. 1, p. 1, 2004.

[2] G. Lindgaard, G. Fernandes, C. Dudek, and J. M. Brown, "Attention web designers: You have 50 milliseconds to make a good first impression!," *Behav. Inf. Technol.*, vol. 25, no. 2, pp. 115–126, 2006.

[3] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2122–2131, 2014.

[4] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar, "Klotski: Reprioritizing web content to improve user experience on mobile devices," in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pp. 439–453, USENIX Association, 2015.

[5] D. Crockford, "jsmin," 2023. See https://www.crockford.com/jsmin.html.

[6] mishoo, "uglify-js," 2023. See https://www.npmjs.com/package/uglify-js.

[7] Rollup, "Tree shaking," 2023. See https://rollupjs.org, also see https://rollupjs.org/faqs/#what-is-tree-shaking for tree-shaking.

[8] webpack, "Tree shaking," 2023. See https://webpack.js.org. Also, see https://webpack.js.org/guides/tree-shaking/#root for tree shaking.

[9] E. Arteca, F. Tip, and M. Schaefer, "Enabling additional parallelism in asynchronous javascript applications," *35th European Conference on Object-Oriented Programming (ECOOP 2021)*.

[10] A. Turcotte, M. W. Aldrich, and F. Tip, "Reformulator: Automated refactoring of the n+1 problem in database-backed applications," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, (New York, NY, USA), Association for Computing Machinery, 2023.

[11] S. Gokhale, A. Turcotte, and F. Tip, "Automatic migration from synchronous to asynchronous JavaScript APIs," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[12] ECMAScript, "Proposal for top level awaits," 2023. See https://github.com/tc39/proposal-top-level-await.

[13] J. Park, I. Lim, and S. Ryu, "Battles with false positives in static analysis of javascript web applications in the wild," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume* (L. K. Dillon, W. Visser, and L. A. Williams, eds.), pp. 61–70, ACM, 2016.

[14] H. Y. Kim, J. H. Kim, H. K. Oh, B. J. Lee, S. W. Mun, J. H. Shin, and K. Kim, "DAPP: automatic detection and analysis of prototype pollution vulnerability in node.js modules," *Int. J. Inf. Sec.*, vol. 21, no. 1, pp. 1–23, 2022.

[15] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting node.js prototype pollution vulnerabilities via object lookup analysis," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021* (D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, eds.), pp. 268–279, ACM, 2021.

[16] Abhishek312s, "Movies-web-ui," 2023. See https://github.com/Abhishek312s/Movies-web-ui/58904a3.

[17] eligrey, "file-saver," 2023. See https://www.npmjs.com/package/file-saver.

[18] SheetJS, "xlsx," 2023. See https://www.npmjs.com/package/xlsx.

[19] Microsoft, "CodeQL," 2023. See https://codeql.github.com/.

[20] Microsoft, "CodeQL JavaScript data flow library," 2023. See https://github.com/github/codeql/tree/7323d4e/javascript/ql/lib/semmle/javascript/dataflow.

[21] Babel, "Babel," 2023. See https://babeljs.io/.

[22] Harinathlee, "upoint-query-builder," 2023. See https://github.com/Harinathlee/upoint-query-builder/f9aa0f1.

[23] sadupawan1990, "excelreader," 2023. See https://github.com/sadupawan1990/excelreader/4a5f9cb.

[24] fahimahammed, "task," 2023. See https://github.com/fahimahammed/task/b641bc0.

[25] hongtaodai, "react-excel," 2023. See https://github.com/hongtaodai/react-excel/2d59e85.

[26] vishumane, "Excelsheet_validation_reactjs," 2023. See https://github.com/vishumane/ExcelSheet_Validation_Reactjs/f38cb9e.

[27] thewca, "scrambles-matcher," 2023. See https://github.com/thewca/scrambles-matcher/1de93f7.

[28] hoverGecko, "timetable," 2023. See https://github.com/hoverGecko/timetable/0fa8527.

[29] Akalay27, "workday-schedule-exporter," 2023. See https://github.com/Akalay27/workday-schedule-exporter/97ca596.

[30] ultimateakash, "react-excel-csv," 2023. See https://github.com/ultimateakash/react-excel-csv/18c6d97.

[31] E. Arteca and A. Turcotte, "Npm-filter: Automating the mining of dynamic information from npm packages," in *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, (New York, NY, USA), p. 304–308, Association for Computing Machinery, 2022.

[32] Google, "Chrome DevTools," 2023. See https://developer.chrome.com/docs/devtools/.

[33] S. Bhattacharya, K. Gopinath, and M. G. Nanda, "Combining concern input with program analysis for bloat detection," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 745–764, 2013.

[34] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating," in *Proceedings of the 12th European Workshop on Systems Security*, pp. 1–6, 2019.

[35] B. Livshits and E. Kiciman, "Doloto: Code splitting for network-bound web 2.0 applications," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, (New York, NY, USA), p. 350–360, Association for Computing Machinery, 2008.

[36] C. Soto-Valero, D. Tiwari, T. Toady, and B. Baudry, "Automatic specialization of third-party java dependencies," *arXiv preprint arXiv:2302.08370*, 2023.

[37] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: Application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, (New York, NY, USA), p. 329–339, Association for Computing Machinery, 2018.

[38] A. Turcotte, E. Arteca, A. Mishra, S. Alimadadi, and F. Tip, "Stubbifier: debloating dynamic server-side javascript applications," *Empirical Software Engineering*, vol. 27, no. 7, p. 161, 2022.

[39] I. Malavolta, K. Nirghin, G. L. Scoccia, S. Romano, S. Lombardi, G. Scanniello, and P. Lago, "Javascript dead code identification, elimination, and empirical assessment," *IEEE Transactions on Software Engineering*, pp. 1–23, 2023.

[40] H. Vázquez, A. Bergel, S. Vidal, J. Díaz Pace, and C. Marcos, "Slimming javascript applications: An approach for removing unused functions from javascript libraries," *Information and Software Technology*, vol. 107, pp. 18–29, 2019.

[41] R. Khatchadourian, Y. Tang, M. Bagherzadeh, and S. Ahmed, "Safe automated refactoring for intelligent parallelization of Java 8 streams," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019* (J. M. Atlee, T. Bultan, and J. Whittle, eds.), pp. 619–630, IEEE / ACM, 2019.

[42] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. E. Johnson, "Relooper: refactoring for loop parallelism in Java," in *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pp. 793–794, 2009.

[43] J. Wloka, M. Sridharan, and F. Tip, "Refactoring for reentrancy," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pp. 173–182, 2009.

[44] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pp. 397–407, 2009.

[45] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding javascript event-based interactions," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 367–377, 2014.

[46] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Understanding asynchronous interactions in full-stack javascript," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 1169–1180, 2016.

[47] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Hybrid dom-sensitive change impact analysis for javascript," in *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.