Stubbifier: Debloating Dynamic JavaScript Applications

ANONYMOUS AUTHOR(S)

JavaScript's package ecosystem eases development by allowing programmers to quickly find and include modules in their projects so as to not reimplement functionality that already exists. That said, modules install with *all* of their functionality even if only a subset is used, which can cause an undue increase in code size as projects mature and more modules are sought out. There exist tools known as *bundlers* which attempt to minimize production distribution of packages by performing dead-code elimination, but these tools are insufficient, as in order to eliminate code these tools must be *sure* that the code is not used elsewhere in the program, a fact that is difficult to establish in a highly dynamic language such as JavaScript.

We propose a technique wherein instead of eliminating potentially dead code, we replace the code with a stub capable of fetching and executing the code on-demand; unreachable code is found via static or dynamic call graphs constructed from an application's tests, and this code is replaced by either file- or function-level stubs. Our technique is fully automatic, and supports the latest version of JavaScript (ES2019), and we implement this technique in a tool called *Stubbifier*. We find that *Stubbifier* significantly reduces the initial size of applications, that little code needs to be fetched and executed dynamically, and that the overhead introduced by this dynamic code loading is minimal. *Stubbifier* also supports a *guarded execution mode*, which guards applications against injection vulnerabilities in dynamically loaded code. Finally, we compare *Stubbifier* to an existing JavaScript bundler, and show that *Stubbifier* out-debloats bundlers, and further that it can synergize with bundlers to create even smaller bundled applications.

Additional Key Words and Phrases: debloating, program analysis, JavaScript, Node.js

ACM Reference Format:
Anonymous Author(s). 2018. Stubbifier: Debloating Dynamic JavaScript Applications. 1, 1 (April 2018), 23 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

JavaScript has become one of the most popular programming languages, and has been the lingua franca of client-side web development for some years [GitHub 2020; Stack Overflow 2020]. More recently, platforms such as Node.js [OpenJS Foundation 2021] have made it possible to use JavaScript for writing code that runs outside of the browser. Node.js provides a light-weight, fast, and scalable platform for writing network-based applications, enabling web developers to use the same language for both front- and back-end development. As such, server-side JavaScript development has experienced an exponential growth in the past few years.

This has given rise to a flourishing ecosystem of libraries, known as Node modules, that are freely available and widely used. The `npm` [npm 2021a] package-management system in particular has fostered higher developer productivity and increased code reuse by unburdening the programmers from many routine development tasks. As such, a typical Node module *m* can directly and indirectly rely on a myriad of other modules. While this is an essential attribute of this ecosystem, in practice, *m* typically uses only a small fraction of the functionality of its dependencies, while it still encompasses all of their code. In turn, clients of *m* inherit the unused functionality of *m* and its dependencies, as

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
XXXX-XXXX/2018/4-ART $15.00
https://doi.org/10.1145/1122445.1122456

well as that of its other dependencies. The problem of accumulating code that in practice is never invoked is known as code "bloat."

While eliminating code bloat is desirable, "debloating" Node.js applications is challenging since it is nearly impossible to perform sound static analysis on JavaScript due to the high dynamism of the language. Despite the popularity of Node.js development and severity of this issue, there is currently no technique available that can significantly debloat a modern Node.js application and still preserve its complete original behavior. JavaScript *bundlers* are applications whose primary focus is to create self-contained application distributions, and these bundlers do perform some debloating by, e.g., performing tree-shaking on imported external modules. Unfortunately, size reduction achieved by this approach is limited by the all-or-nothing nature of their code minimization: code that the bundler removes must *never* be called, else the bundled application will crash.

Previous work in the context of other languages has focused on the use of static analysis to determine unreachable code [Agesen and Ungar 1994; IBM Corporation 1997; ParcPlace-DigiTalk 1995; Tip et al. 2002]. In many existing techniques, the application stops executing when trying to invoke code that has been removed by the debloating algorithm and deviates from the intended behavior of the original application. Despite more recent advances for analyzing client-side web applications [Andreasen and Møller 2014; Jensen et al. 2011; Li et al. 2018a,b; Livshits and Kiciman 2008; Sridharan et al. 2012], the development of a static analysis for Node.js that is simultaneously sound, precise, and scalable is beyond the current state of the art.

In this paper, we present a novel technique for reducing the size of Node.js applications while preserving the original behavior, and implement this technique in a tool called *Stubbifier*. Our approach is inspired by the Doloto [Livshits and Kiciman 2008] tool, and comprises a combination of unsound static and dynamic analysis with a technique known as *code splitting* pioneered by Doloto. For a given application, *Stubbifier* infers a call graph from its tests, determines which functions and files are unreachable, and instead of *removing* this code outright, *replaces* it with stub versions that fetch and execute the original code dynamically. Our technique expands on Doloto in its support for modern JavaScript [International 2021], in its addition of file-level stubs, and importantly in its fully-automatic nature; whereas Doloto required traces of users interacting with the subject application, our approach debloats projects automatically, by determining what code is reachable from their tests. Now, as stubbed code represents untested code, *Stubbifier* also allows users to opt-in to a *guarded execution mode*, where stubbified code is modified to intercept calls to functions such as `eval` and `exec` that can execute arbitrary user code. The reduced Node.js application is then ready to be deployed, exercised, and reused by other applications.

Our work makes the following contributions.

- We propose an automated technique for reducing the size of Node.js applications. Through a combination of dynamic and static analysis along with code splitting, our approach allows the reduced version to dynamically load and execute *debloated* code, and ensures that the original behavior of a reduced application is preserved.
- We implement our approach in a tool called *Stubbifier* that supports the newest features of the JavaScript language for Node.js development. *Stubbifier* is automated and will be publicly available as an open-source tool. [1]
- We empirically evaluate our approach through a set of experiments conducted on 15 open source Node.js applications and 75 clients of these subject applications (five clients per subject, respectively). The results of our experiments show that *Stubbifier* is effective in reducing the size of Node.js applications by 56% on average while incurring only minor performance overhead.

[1]Tool URL is not provided at this time due to double-blind review.

2 BACKGROUND AND MOTIVATION

Using `npm` significantly facilitates the means of publishing, discovering, reusing, and developing Node.js modules, allowing programmers to create more concise programs that reuse existing functionality. However, this ease is not without its price: importing modules can cause projects to become excessively large due to transitive importing and downloading of dependencies. In practice, it is likely that the importing module will only use a small subset of the transitive closure of dependencies, and the rest is all unnecessary bloat. To illustrate this problem, consider the example of a popular node application `css-loader` [webpack-contrib 2021], a utility package for loading, parsing, and transforming `CSS` files and further supporting applications designed to use `CSS`. It is one of the most popular modules on `npm`, with over 10 million weekly downloads. It is imported by over 12,000 modules.

`css-loader` has 13 third-party production dependencies along with core-module dependencies. The stand-alone `css-loader` module contains only 16 files comprising 110KB. Now, installing `css-loader` with direct and indirect dependencies creates a package with 1299 files and total code size of 2764KB. This is a >81x and >25x increase in number of files and code size respectively.

To determine what constitutes application bloat, we ran a preliminary analysis on `css-loader` to determine which functions and files were reachable from the application's test suite. This simple analysis traced function calls to build a list of unreachable functions and files, and we found that 209 files were marked as potentially unreachable, and found 6 unreachable functions in otherwise reachable files. As we are dealing with JavaScript, we are cognisant that sound analysis is nearly impossible, and thus some of these potentially unreachable functions and files may indeed be reachable. Nevertheless, if we *could* devise a technique to remove all of this code, we would be able to reduce the application's size by 76.43%.

To get a sense for what kind of functionality is trimmed, consider `semver`, a dependency of `css-loader`. `semver` is the "semantic versioner for `npm`" [npm 2021b]. As it happens, only *two* functions from `semver` are ever used in `css-loader`. The `satisfies` function is imported specifically as part of the primary `css-loader` functionality, and the `inc` function is used once as a helper in the `css-loader` tests. Given that, it seems wasteful to include the *entire* `semver` code in `css-loader`, and indeed our analysis finds 27 of `semver`'s files and six `semver` functions inside of `css-loader` to be potentially unreachable. The six unreachable functions are in the file that exports the `inc` function. After debloating, the code in the `semver` package is reduced from 57KB to 35KB, a 38% size reduction.

2.1 JavaScript Bundlers

JavaScript is used for both back-end and front-end development. Many JavaScript projects use *bundlers*, tools which are designed to "bundle" applications into single-file distributions, in order to neatly package all required functionality. The idea with these bundles is that they can simply be included in other applications using **`require`** or **`import`**, but crucially without having to go through `npm` or other package managers.

Bundlers work by performing a simple static analysis of the project to determine what functionality to include in a bundle. If the project relies on dynamic **`require`** statements to import external functionality, the required files are simply included in the bundle in their entirety; if the project relies on the ECMAScript module system and its static imports and exports, tree-shaking can be performed on the imported modules, which may trim unused functionality. Thus, bundlers are in effect code debloating tools, though they cannot achieve the size reductions discussed in the `css-loader` example since they need to be *absolutely sure* that the code they remove is never invoked, otherwise the bundled application could crash. This is difficult in JavaScript, as the language's dynamism makes sound program analyses elusive.

In our evaluation, we will compare the effectiveness of bundlers against the effectiveness of *Stubbifier*, but we note that these tools are not irreconcilable: We will also show that *Stubbifier* can work *alongside* bundlers to produce minimal application bundles.

2.2 Doloto

Besides bundlers, there exist other tools to reduce JavaScript code size. Doloto [Livshits and Kiciman 2008] is one such application: it utilizes "access profiles" obtained from a user interacting with an instrumented application. These access profiles define clusters of functions that should be loaded together, and functionality that should be part of the distribution of an application. Applications processed by Doloto ship with enough functionality for initialization, and inessential functions are replaced with small stubs that are either replaced once their original code is loaded lazily, or on-demand if ever a stubbed function is invoked.

Our approach is indeed inspired by Doloto, though we differ and expand on Doloto in a few key ways, expored next.

3 OVERVIEW

Given an application, the first step of the approach is to build a call graph. Then, the application code is parsed, including any dependencies, and the call graph is used to identify unreachable code which is replaced with stubs. If the stub is ever invoked, the stub code dynamically fetches and executes the original code. Dynamic code loading is only ever performed once so as to minimize run time overhead in the event of stub expansion. The approach also allows users to opt-in to a guarded execution mode, which intercepts function calls in expanded code to ensure that they do not execute arbitrary user code.

Our technique expands on Doloto's in that it supports a much larger JavaScript language with more features (ES2019), and additionally supports file stubs where Doloto only supports function stubs. Further, our approach is fully automatic: whereas Doloto requires access profiles of users interacting with an application, our approach automatically determines which code to debloat. We encapsulate this approach in a tool called *Stubbifier*, and detail each of these steps next.

4 INPUT CALL GRAPH

Our technique provides the user with two options for a call graph: one derived from a dynamic analysis, and one from a static analysis. For both cases, *Stubbifier* uses the test suite of the input application as the entry point for the analysis, so the call graph represents the *tested* code, and anything not in the call graph is unreachable and untested. We configured both analyses to consider project dependencies found in the `node_modules` directory, which are otherwise ignored. Note that we explicitly exclude development dependencies, as they are excluded from production distributions.

It is worth mentioning that our approach is call graph-agnostic. We implemented two call graphs in our implementation of *Stubbifier*, but in principle any call graph will suffice, though soundness and precision of the call graph will obviously impact the size of the initial distribution and the amount of code that needs to be loaded dynamically.

4.1 Dynamic Call Graph

To compute the *dynamic* call graph, we use Istanbul[2], a coverage tool for Node.js that computes statement, line, branch, and function coverage for JavaScript. By default, Istanbul ignores the

[2]Link to the Istanbul project.

dependencies of the project, and we automatically generate a configuration file that specifies that coverage of *non-development*, production dependencies should be computed. We then run Istanbul on the application's tests, telling us which functions and files were invoked during testing.

4.2 Static Call Graph

To compute the *static* callgraph, we use QL [Avgustinov et al. 2016], GitHub's declarative language for semantic code analysis, which has extensive libraries for writing static analyzers [GitHub 2021]. The QL dataflow library contains functionality for tracking calls through local module imports—we implemented an extension to recognize modules in a project's `node_modules` directory, and extended QL's libraries to track through these modules. Then, we built our call graph analysis using this module tracking extension. Like with Istanbul, we use the application's tests as an entry point for the analysis.

5 GENERATING STUBS

We process the generated call graph to obtain a list of unreachable functions and files. *Unreachable files* are those in which *none* of the functions are reachable, and *unreachable functions* are those functions that are not reachable but that are in a file where at least one other function *is* reachable.

Then, using Babel [BabelJS 2021], we parse the application code, including any dependencies, and replace unreachable functions and files with *stubs* via transformations on the program's Abstract Syntax Tree (AST). We do not stub functions and files that are shorter than the stubs that would replace them.

5.1 File Stubs

We replace each unreachable file with a stub. The code in this generated stub implements Algorithm 1, which depicts the general logic for file stub expansion.

Algorithm 1: ExpandFileStub

```
1 perform all imports;
2 let $file_o$ := fetchOriginalFileCode();
3 let $file_e$ := eval($file_o$);
4 replace this file with $file_o$;
5 perform all exports;
```

At a high level, file stub expansion amounts to first performing all imports that were in the original code (line 1), then fetching the original code and evaluating it (lines 2-3), replacing the contents of the stubbed file with the original file (line 4) and finally performing necessary exports (line 5). More specifically, in files that use `require` and `module.exports` to import/export, simply storing the original code elsewhere and `eval`-ing it as needed suffices, as these mechanisms can operate dynamically. However, the ECMAScript Module System (ESM) [ECMA 2015]'s static `import`/`export` constructs cannot be executed in an `eval` (see section 15.2 of [International 2021]), so we hoist all `import` and `export` statements out of the original code and into the stub. The original code is then transformed to properly produce the values of the exports. To illustrate, consider the example in Figure 1.

In Figure 1, we see `import` and `export` statements interspersed through the file before stubbification. In the lower part of the figure, we see that the file stub generated by *Stubbifier* contains all `import` statements as-is, and `export` statements are modified (lines 13 and 14) to get their values from the dynamically executed code (i.e., from `exportObj`, line 10).

```
1  // file.js before stubbification
2  export function foo() { /* ... */ }
3  import { A };
4  function bar() { /* ... */ }
5  export default bar;
6
7
8  // file.js after stubbification
9  import { A };
10 exportObj = eval(stubs.getCodeForFile("file.js"));
11
12 let foo_UID = exportObj["foo"];
13 export {foo_UID as foo};
14 export default exportObj["default"]
```

Fig. 1. File before and after stubbification.

To allow this exporting, the original code from Figure 1 is modified to construct an object containing all of the original exports. This constructed object is the last statement that will be executed when the code is passed to `eval`, and is therefore the return value of `eval`. This is illustrated in Figure 2.

```
15 function foo() { /* ... */ }
16 function bar() { /* ... */ }
17
18 {  foo: foo,
19    default: bar };
```

Fig. 2. Modified original code with ES6 imports and exports (this is what would be eval'd).

Here, we see that the **export** was removed from the definition of `foo`, and that `foo` was added to an object on line 18, which also includes an entry for `bar`, the default export. The last statement in an `eval`-ed code block is implicitly returned—here, that is an object containing the exports—allowing the stub to retrieve the exported values (as in line 10 of Figure 1).

5.2 Function Stubs

Functions deemed unreachable are also replaced with stubs. These stubs implement Algorithm 2, which depicts the general logic for dynamically loading and executing code upon stub expansion. When a stub is expanded, it first fetches the code, either by getting it from a cache, fetching it from a server, or otherwise retrieving it from storage. Either way, the code is evaluated into a function, and function properties are copied from the stub version to the newly created function object. If possible, *Stubbifier* will replace the stub with the freshly evaluated original function. If not, the code is cached, and then the function is executed. A concrete example of a function stub can be found in Figure 3, where we show the stub for `getValidHeaders` from the `node-blend` [nod 2021] project.

First, note that *Stubbifier* outfits each file with a global `stubs` object which contains the code cache and functionality to fetch code. On line 21 we see a call to `stub.getCode("UID_for_LOC")` which fetches the *original function definition* (found through `"UID_for_LOC"`, a unique ID for the

Algorithm 2: ExpandFunctionStub

```
    Data: args: function arguments
    Data: uid: unique ID for this function stub
 1  if uid cached then
 2      let $fun_{str}$ := code cached at uid;
 3  else
 4      let $fun_{str}$ := fetch original function code;
 5  let $fun_e$ := eval($fun_{str}$);
 6  copy function properties to $fun_e$;
 7  if can replace function definition then
 8      replace stub with $fun_e$;
 9  else
10      cache $fun_{str}$;
11  call $fun_e$ with args;
12  return result;
```

```
20  function getValidHeaders(headers) {
21    let toExec = eval(stubs.getCode("UID_for_LOC"));
22    stubs.cpFunProps(getValidHeaders, toExec);
23    getValidHeaders = toExec;
24    return toExec.apply(this, arguments);
25  }
```

Fig. 3. Example of a stubbed function.

function that *Stubbifier* generates from the code location when the stub is created). That code is then passed to `eval`, which will return a function object containing the original code. Line 22 copies any function properties from `getValidHeaders` to the fresh function[3]. Finally, line 23 redefines the `getValidHeaders` with the expanded stub, and line 24 calls the function with its original arguments[4]. Since `getValidHeaders` has reassigned itself on line 23, any subsequent calls to this function will call the expanded stub, with no need to re-`eval` the code.

This is the general approach for function stubs, however some types of functions required special treatment.

Anonymous Functions. In JavaScript it is possible to create a function with no identifier, commonly seen when creating an anonymous function passed as a callback argument. In these cases, the function cannot reassign itself in the style of line 23 (since it has no name to refer to itself by), so the loaded code is cached, and future stub expansions `eval` the cached code. For example, Figure 4 displays the `getValidHeaders` stub if this function did not have an ID. Here we see that rather than immediately passing the code loaded with `stubs.getCode("UID_for_LOC")` to `eval`, the `stubs` cache is accessed on line 27. Code is only loaded on a cache miss, in which case the loaded code is immediately cached.

[3]Recall that in JavaScript functions are objects, and can have properties assigned dynamically.
[4]`apply` calls its receiver as a function, binding its first argument to **`this`** inside the function, and passing the other arguments as function arguments. `arguments` is a metavariable available inside functions that refers to its arguments.

```
function(headers) {
    let toExecString = stubs.getStub("UID_for_LOC");
    if (! toExecString) {
        toExecString = stubs.getCode("UID_for_LOC");
        stubs.setStub("UID_for_LOC", toExecString);
    }
    let toExec = eval(toExecString);
    toExec = stubs.cpFunProps(this, toExec);
    return toExec.apply(this, arguments);
}
```

Fig. 4. Example of stubbed anonymous function.

One might wonder why the function stub expansion caches the loaded code, evaluating it every time the stub is invoked, rather than caching the expanded function object. Functions in JavaScript are *closures* which *close* variables from surrounding scopes directly into the object. If we are generating a stub for a function inside of another, *function arguments* are included in the closure. If we were to cache this object, any subsequent call to the function would refer to the values of function arguments when the stub was first expanded, which may lead to incorrect program behavior. Thus, we have to `eval` every time. Note that this problem is not present for functions with ID, as the function reassigning itself does not store a closure.

Class and Object Methods. When stubbifying object or class methods, an issue arises with references to **`this`**. In functions outside a class or object, **`this`** refers to the function object itself, while in a class/object, **`this`** refers to the *object instance* on which the function was invoked. These class methods need to be referenced in a different way to allow for function property copying and reassignment.

Fortunately, class and object methods can be accessed as *properties* of **`this`**. If `getValidHeaders` was a method in some **`class`** `A`, the following replacements would be made:

```
// outside a class/object
stubs.cpFunProps(getValidHeaders, toExec);
getValidHeaders = toExec;

// inside a class/object
stubs.cpFunProps(this.getValidHeaders, toExec);
this.getValidHeaders = toExec;
```

For class/object methods with no ID, we generate a dynamic property access on **`this`** to reassign the function object as at code generation time, we know the key corresponding to nameless object properties.

Classes/objects can also have *getter* and *setter* methods:

```
class A {
    get propName() { console.log("getter"); }
    set propName() { console.log("setter"); }
}
let x = new A();
x.a; // prints "getter"
x.a = 5; // prints "setter"
```

Getter and setter stubs are generated with special reassignment code. Dynamically accessing and defining a getter for some property `"p"` is done using `this.__lookupGetter__("p")` and `this.__defineGetter__("p")` respectively (and similarly for setters); these calls are used in place of direct accesses as properties of `this` in the stub.

Arrow Functions. Arrow functions were introduced in ECMAScript 2015, and provide a more concise syntax for functions. When creating stubs for arrow functions, we run into an issue as the metavariable `arguments` cannot be used to reference the function arguments. To get around this, we make use of the *rest parameter* [Mozilla 2021], also introduced with ES6. By replacing the original function parameters with a rest parameter, we have essentially recreated the functionality of `arguments`. For example, if `getValidHeaders` was an arrow function, it would be written as:

```
let getValidHeaders = (headers) => { ... }
```

and its stub would resemble:

```
let getValidHeaders = (...args_UID) => {
    // only change the last line of the stub
    getValidHeaders.apply(this, args_UID);
}
```

Unstubbable Functions. *Stubbifier* does not transform generators, as `yield` cannot be present inside of an `eval`, nor does it transform constructors. Constructors necessitate that `super` be called before any use of the `this` keyword. Generating constructor stubs would require a more sophisticated analysis of constructor code, and as sound static analysis of JavaScript is still very challenging, we decided against stubbifying them altogether. We do not consider this to be a big issue, as these types of functions are rarely stubbed.

5.3 Bundler Integration

Recall that bundlers are JavaScript projects with the goal of creating application *bundles*, which are self-contained versions of projects that are comparatively easier to distribute. Many of these bundlers perform some form of limited code debloating, and we will compare this with Stubbifier's debloating capabilities in the evaluation. Before that, we describe how Stubbifier can *synergize* with bundlers to create even smaller application bundles; there are a few changes to the transformation pipeline that are required in order to support integration with bundlers.

First, bundling must always happen *before* the stubbifying, as bundlers perform their own code transformation: when merging all the application code into one file, they often refactor the code so as to avoid variable name conflicts and repeated imports. If *Stubbifier* was run on the application before bundling, the bundler would only perform its analysis on the stubbed code, since the stubs are just stored as strings. Then, when a stub is expanded in the bundle, it has not been transformed to match the bundle and is therefore likely to have naming conflicts and result in an error.

We base our call graph construction on the tests of an application, and tests are nearly always based on the original project source code, and not bundles. To circumvent this, *Stubbifier* determines a mapping of the functions to be stubbed from their positions in the original code to their positions in the bundle. This way, we can still construct call graphs from tests without having to reconfigure the test suite to invoke the application bundle. We thus implement a two-stage transformation: we build the call graph as normal, but instead of directly stubbing the identified dead code, we transform this code and add a flag call as the first statement in each of these functions (in particular, `eval("STUB_THIS_FUNCTION")`). Then, we bundle the application. After bundling, we parse the bundle and stub all flagged functions, ensuring that we remove the flag after stubbing.

Note that we do not perform any file stubbing when integrating with bundles: since the bundler generates one big file with all the code, this wouldn't make sense. Instead, in files that originally would have been fully stubbed, we just flag every function to be stubbed.

In summary, the bundler integration process is as follows:

(1) Build the call graph (same process as for un-bundled code);
(2) Transform all functions that should be stubbed by inserting the flag `eval("STUB_THIS_FUNCTION")`;
(3) Automatically generate a bundler configuration file, and run the bundler;
(4) Transform the bundle: parse and stub functions starting with `eval("STUB_THIS_FUNCTION")`, and remove the flag calls.

In Section 7, we discuss an experiment on the effect of *Stubbifier* on bundled code.

6 GUARDED EXECUTION MODE

Since *Stubbifier* builds the input call graph based on the application's tests, the stubbed code is also the *untested* code. Thus, executing this code could pose a security risk.

Stubbifier includes an option to detect calls to a prespecified list of "dangerous" functions in expanded code. This is achieved by intercepting all function calls with a check to see whether or not the function is (perhaps an alias of) one of these dangerous functions. In our current implementation, the list of these functions consists of: `eval`, `process.exec`, `child_process.fork`, and `child_process.spawn`, common functions that execute arbitrary code. It is trivial to include other functions to this list, so users can customize what functions they want guard against. We include an example of the code with guards in the supplementary material.

These checks can be configured to generate a warning, or exit the application if a dangerous function is about to be called. This transformation is run on the original code, so that stubs load the code with guards included.

Since these functions could be aliased, we must wrap *every* function call with these checks. As such, the size of the loaded code is increased dramatically. The guards also incur more runtime overhead, which we discuss in the following section.

7 EVALUATION AND DISCUSSION

This section presents an evaluation of *Stubbifier*, in which we pose the following research questions:

- **RQ1.** How much does *Stubbifier* reduce application size?
- **RQ2.** How much code is dynamically loaded when the stubbified applications are used?
- **RQ3.** What is the overhead of stub expansion?
- **RQ4.** What is the run time of *Stubbifier*?
- **RQ5.** What are the effects of enabling guards?
- **RQ6.** How do the size reductions achieved by *Stubbifier* relate to those achieved by bundlers?

7.1 Experimental Setup and Methodology

To find projects to evaluate *Stubbifier*, we picked from a list of the most popular projects published by `npm`: we downloaded and installed the project, tried running its test suite, and excluded projects that were untested or had failing tests, as *Stubbifier* uses the test suite to generate the call graphs.

Table 1a lists the projects we use for our evaluation, as well as some relevant metrics. The first row reads: the project `memfs` has 18k lines of code (LOC) in the analyzed files[5], and there are 133 files analyzed (Num files). The `memfs` test suite has 284 tests (Tests), one production dependency (Production deps)[6], and its analyzed code comprises 146 KB (Size).

[5]This includes the project's source code, and all the production dependencies in `node_modules`.
[6]This includes transitive production dependencies, and does not include `devDependencies`.

Project	LOC	Num files	Tests	Production deps	Size (KB)
memfs	18k	133	284	1	146
fs-nextra	11k	184	138	0	52
body-parser	20k	210	231	21	364
commander	13k	177	351	0	70
memory-fs	14k	167	44	11	120
glob	13k	175	1706	10	86
redux	105k	4491	82	2	267
css-loader	71k	1299	430	36	2764
q	16k	135	243	0	281
send	14k	157	152	17	97
serve-favicon	10k	121	30	5	20
morgan	14k	159	81	8	55
serve-static	13k	160	90	19	106
prop-types	15k	152	287	4	106
compression	13k	149	38	11	66

(a) Summary of projects used for evaluation

Project	Size (KB)	Reduction %	Expanded (KB)	Red after exp (%)
memfs	19	87%	[19, 138]	[87%, 5%]
fs-nextra	31	39%	[31, 45]	[39%, 14%]
body-parser	65	82%	[211, 297]	[42%, 18%]
commander	68	2%	[68, 68]	[2%, 2%]
memory-fs	41	66%	[41, 87]	[66%, 27%]
glob	61	28%	[70, 80]	[18%, 7%]
redux	201	25%	[221, 221]	[17%, 17%]
css-loader	559	80%	[559, 895]	[80%, 68%]
q	37	87%	[37, 100]	[87%, 64%]
send	59	39%	[59, 92]	[39%, 5%]
serve-favicon	15	24%	[15, 18]	[24%, 8%]
morgan	25	55%	[41, 45]	[25%, 20%]
serve-static	38	64%	[38, 83]	[64%, 21%]
prop-types	18	83%	[56, 56]	[48%, 48%]
compression	24	63%	[24, 24]	[63%, 63%]

(b) Size of projects stubbified with static CG

Project	Size (KB)	Reduction %	Expanded (KB)	Red after exp (%)
memfs	17	89%	[17, 136]	[89%, 7%]
fs-nextra	47	10%	[47, 47]	[10%, 10%]
body-parser	173	53%	[180, 253]	[51%, 31%]
commander	59	16%	[59, 59]	[16%, 16%]
memory-fs	100	17%	[100, 117]	[17%, 3%]
glob	84	4%	[91, 91]	[-6%, -6%]
redux	189	29%	[209, 209]	[22%, 22%]
css-loader	584	79%	[584, 1372]	[79%, 50%]
q	206	27%	[206, 209]	[27%, 26%]
send	89	8%	[89, 93]	[8%, 5%]
serve-favicon	19	3%	[19,19]	[3%, 3%]
morgan	49	13%	[52, 52]	[7%, 7%]
serve-static	98	7%	[98, 102]	[7%, 4%]
prop-types	16	85%	[53, 53]	[50%, 50%]
compression	46	29%	[46, 46]	[29%, 29%]

(c) Size of projects stubbified with dynamic CG

Table 1

Experimental Design and Procedure. For each project, we generated static and dynamic call graphs. For each call graph, we replace unreachable functions and files with stubs. To address **RQ4**, we time the process from call graph generation to stubbification. To address **RQ1**, we compare the size of the application before and after stubbification. We compute only the size of source code (excluding tests), *including* production dependencies and *excluding* development dependencies.

To address **RQ2** and **RQ3**, we identified five clients of each of our original packages by looking at their list of dependents on `npm`, excluding clients without tests or with failing tests. We also made sure that the dependency is actually used in the client: there are some projects that list a package as a dependency but it is no longer used in the source code, and we excluded these. Finally, we exclude clients that rely on an old version of Node.js. We run all our experiments on the same version of Node.js (14.3.0), to avoid any updates to the runtime environment that could affect run times and thus skew our results.

To determine the performance overhead caused by executing stubbed code, we compared the runtime of each of these clients' tests when using the stubbed and original dependency. When running the test suite with the stubbed dependency, we also tracked the total size and number of stub expansions to determine *how much* code is loaded dynamically. For the purpose of our evaluation, the stubs were kept on the machine running the evaluation.

We ran the test suites 10 times to increase our confidence in the results, and report the average. To minimize bias caused by caching, we ran and discarded results for two test runs before running our timed experiments. Since some of the tests generate files in `/tmp`, we also clear this directory between every test suite run. Finally, to mitigate versioning errors, we run our experiments on a client using the same version of the dependency as the one that we transform. Specifically, we do the following when testing a client:

- `npm` or `yarn install` in the root of the client project
- Replace the dependency in question in the client's `node_modules` with a *symbolic link* to the source code of the dependency that we will transform.
- Run the client's tests.
- Transform the dependency. The symbolic link means the client needs no change.
- Rerun the client's tests with the transformed dependency.

For **RQ5**, we rerun our experiments with guards enabled, tracking the client runtimes and size of expanded code to determine the increase in overhead due to these extra checks. We also conduct a case study on `depd`[dep 2021e], an application with a known vulnerability. We invoke *Stubbifier* on *clients* of `depd`, finding that the vulnerability is stubbed, and run the client's tests (similarly for transitive clients). Our guards report the vulnerability, and this case study is detailed below.

For **RQ6**, we configured each of our subject applications to use the `rollup` bundler. To do so, we automatically generate a bundler configuration file which bundles based on the listed entry point of the application, and is configured to create a single bundle that also includes all of the project dependencies[7]. After bundling the application, we applied *Stubbifier* to the bundle. We measure and report the sizes of this bundle, both with and without having used *Stubbifier*, to determine what additional size reduction *Stubbifier* has over that achieved by the bundler.

Overview of Results. Tables 1b and 1c display the results of running *Stubbifier* on our selected projects. We show the size of the original source code, the size of the application distribution, and then the resulting size of the distribution after we run our transformation on it, with both the static and dynamic callgraphs.

[7]The default behaviour of `rollup` is to ignore anything in `node_modules`, but we wanted the bundle to include everything over which *Stubbifier* is applied, which does include the production dependencies.

Note that the size immediately after transformation is only representative of the stubbed application size if none of the stubs are expanded. To gain a realistic estimate of the size reduction in a standard use-case of the application, we also found five clients for each application and tracked how many stubs were expanded during the client's test execution. Then, we consider the size of the application to be its base stubbed size *plus* the total size of the stubs that were expanded during the client tests. This is reported as a range of the lower and upper bounds of application size over the five clients. The full data is included in the supplementary material.

The first row of Table 1b reads: after running *Stubbifier* with the static call graph, the `memfs` source code is reduced the size to 19KB, which is a reduction of 87% of the original application size. This expanded to a minimum of 19KB (i.e., nothing was expanded) and a maximum of 138KB over the five clients tested; the expanded code is a reduction of 87% (with minimum expansion) and 5% (with maximum expansion) of the original application size. The first row of Table 1c can be read the same way, but for results after running *Stubbifier* with the dynamic call graph on `memfs` and testing with the same five clients.

RQ1: How much does *Stubbifier* reduce application size?

To answer this question, we refer the reader to Tables 1b and 1c. In nearly every case, we see size reduction of in excess of 24% for the static call graph, the notable exception being `commander` with only a 2% size reduction (however, `commander` has no dependencies, so it is not surprising that there is less size reduction). The static call graph has a greater size reduction in 11/15 cases, and otherwise the size reduction is comparable between input call graphs.

Many of these packages have millions of weekly downloads, and so the size savings add up quickly: for example, `css-loader` is 2.764MB, and with 10 million weekly downloads we have nearly 28TB of data transferred to users every week. *Stubbifier* reduces `css-loader`'s initial size by 80% with both call graphs, which would contribute to 22 fewer TB being transferred weekly.

RQ1 Takeaway: On average, *Stubbifier* reduces initial application size by 56% using the static call graph, and by 31% with the dynamic call graph.

Now, simply looking at initial size reduction does not tell the full story, as *Stubbifier* is designed to support dynamic code loading. The next research question addresses this.

RQ2: How much code is dynamically loaded when the stubbified applications are used?

Again referring to Table 1b and 1c, this time to the **Expanded KB** range columns, we see that the top end of the expanded ranges using the static call graph are smaller (or equal) than the expanded ranges using the dynamic call graph in 11/15 cases. This aligns with our findings in **RQ1**. In all but one case, the minimal expanded size is close to the reduced application size, and the max size increase is only > 2x in two cases.

To break down the results further, we consider the results for all clients of a few packages. Tables 2a and 2b display all the metrics tracked for all clients of `redux`, `q`, and `body-parser`. These metrics are the test suite runtimes, the percentage slowdown due running the stubbed code, and number and size of stubs dynamically expanded during the tests. We chose these applications to display as we felt they are a representative sample of our results; the full data for all clients of all projects is included in the supplementary material.

The first row of Table 2a reads: for `redux`, its client application `Choices` has an average test suite runtime of 5.06 seconds. When the `Choices` test suite is rerun with stubbed `redux` (via the static call graph), it has an average runtime of 5.16 seconds, which is a slowdown of 2%; 1 file stub and no function stubs were expanded, and the total size of stubs expanded was 20.06KB. The first row of

	Client		Stubbed code: effect of expansions				
Proj	**Client Proj**	**Time (s)**	**Time (s)**	**Slowdown (%)**	**Files**	**Fcts**	**Expanded (KB)**
redux	Choices	5.06	5.16	2%	1	0	20.06
	found	30.61	31.83	4%	1	0	20.06
	Griddle	8.93	8.91	0%	1	0	20.06
	react-beautiful-dnd	61.70	63.49	3%	2	0	20.06
	redux-ignore	0.57	0.58	2%	1	0	20.06
q	decompress-zip	0.70	0.74	6%	1	0	63.25
	downshift	1.43	1.44	1%	1	0	63.25
	node-ping	3.80	4.20	10%	1	0	63.25
	passport-saml	0.41	0.44	6%	0	0	0.00
	requestify	2.92	2.99	2%	1	0	63.25
body - parser	appium-base-driver	8.66	10.04	14%	39	0	146.10
	express	1.05	1.89	45%	48	0	231.69
	karma	2.08	2.12	2%	40	0	199.57
	moleculer-web	5.80	6.46	10%	48	0	231.69
	typescript-rest	13.17	14.89	12%	48	0	231.69

(a) Stubbed with static call graph

	Client		Stubbed code: effect of expansions				
Proj	**Client Proj**	**Time (s)**	**Time (s)**	**Slowdown (%)**	**Files**	**Fcts**	**Expanded (KB)**
redux	Choices	5.06	5.05	0%	1	0	20.06
	found	30.61	31.34	2%	1	0	20.06
	Griddle	8.93	9.03	1%	1	0	20.06
	react-beautiful-dnd	61.70	62.12	1%	2	0	20.06
	redux-ignore	0.57	0.59	3%	1	0	20.06
q	decompress-zip	0.70	0.78	10%	0	5	2.98
	downshift	1.43	1.44	1%	0	1	0.88
	node-ping	3.80	4.08	7%	0	6	2.86
	passport-saml	0.41	0.42	2%	0	0	0.00
	requestify	2.92	3.05	4%	0	2	0.86
body - parser	appium-base-driver	8.66	9.26	6%	8	0	6.69
	express	1.05	1.21	14%	14	0	79.70
	karma	2.08	2.09	1%	12	0	79.03
	moleculer-web	5.80	6.38	9%	14	0	79.70
	typescript-rest	13.17	14.48	9%	14	0	79.70

(b) Stubbed with dynamic call graph

Table 2. Results for Clients of Select Projects

Table 2b shows the results of rerunning again with stubbed redux via the dynamic call graph: now, Choices' test suite has an average runtime of 5.05 seconds, which is a slowdown of 0%; 1 file stub and no function stubs were expanded, and the total size of stubs expanded was 20.06KB.

Digging into the client-specific data reveals some interesting trends. There appears to be a correlation between the number of stubs expanded for the static and dynamic call graphs. For example, consider the clients of body-parser: even though there are more stub expansions using the static call graph vs. using the dynamic call graph, it appears that there are "sets" of functionality that are commonly expanded together (seen here as whenever 48 file stubs are expanded in the static case, 14 file stubs are expanded in the dynamic case). The range of expansions among clients suggest that some of the clients use more of an application's untested functionality than others.

We also noted consistency in *which* stubs are expanded. For example, in the "sets" of expanded functionality described earlier, these are the *same* 48 and 14 files every time. As an additional example, all the clients of `redux` expand one file stub (one client expands two)—this is always the *same* stub that is expanded. In the other applications, there is always significant overlap in which stubs are expanded with different clients. This suggests that some of these applications have commonly used functionalities that are untested, so developers could use this information to shore up their test suites.

Finally, we observe that the dynamic call graph typically produces far fewer file stub expansions than the static call graph. There are a few dimensions to this. On one hand, as JavaScript is a dynamic language, the static call graph is likely to be incomplete—functions in JavaScript are often called in highly dynamic ways, and these kinds of calls are more easily detected using dynamic analyses. On the other hand, the dynamic call graph is more susceptible to lower-quality tests: if the application is poorly tested, the dynamic call graph will report many unreachable functions and files. It is not immediately clear which call graph yields "better" results, as fewer stubs mean less size reduction, but also less overhead—we ultimately leave the decision up to the developer.

RQ2 Takeaway: Most package clients load very little code dynamically. Many applications have commonly loaded "sets" of code, representing broadly used, untested functionality.

RQ3: What is the overhead of stub expansion?

To establish the performance overhead of running applications debloated with *Stubbifier*, we collected run times of test suites for clients of stubbed applications.

We do not aggregate runtime information over all clients of a package as the overhead depends on many factors outside of our control: the number of tests, the structure of the tests, the raw running time of the test, etc. We instead perform a case study on the effect of the dynamic code loading on the individual clients for the three projects presented in Tables 2a and 2b. The results for all of our test applications are included in the supplementary material, but the trends are upheld across the full data. We now refer to the time columns of Tables 2a and 2b.

We can draw a few observations from these results. First, we observe a correlation between the slowdown and the number of stub expansions: as more code is dynamically loaded, the performance overhead increases. This aligns with our expectations, as more expansions mean more compute time than the original expansion-free test suite. That said, the runtime overhead is never extreme, and the slowdowns still leave the runtimes of the test suites well within the same order of magnitude. As a percentage, some runtime overhead is high (e.g. `body-parser`'s `express` dependency), but the magnitude of the change is not (only 0.84 seconds). We do not see high percentage slowdowns for long-running tests, for instance `redux`'s `found` and `react-beautiful-ignore` clients have 4% and 3% slowdowns respectively. We conjecture that the amount of overhead mostly has to do with the I/O required to load the dynamic code.

RQ3 Takeaway: By and large, the magnitude and percentage overhead introduced by dynamic loading is small.

RQ4: What is the runtime of *Stubbifier*?

We report the run time of *Stubbifier* on each of the original 15 projects in Table 3. We separate the times to build the call graphs from the times to apply our transformation.

	Static CG		Dynamic CG	
Project	**CG generation (s)**	**Transf. (s)**	**CG generation (s)**	**Transf. (s)**
memfs	740.18	2.46	15.97	2.72
fs-nextra	380.97	1.11	13.94	1.10
body-parser	295.38	3.43	10.53	3.79
commander	554.93	1.94	24.56	1.61
memory-fs	324.06	1.73	5.33	1.75
glob	300.80	2.09	18.66	1.46
redux	1349.09	3.18	182.02	4.02
css-loader	1137.77	14.85	48.61	15.52
q	336.31	4.41	10.98	4.85
send	279.16	1.75	7.57	1.67
serve-favicon	259.06	0.76	3.91	0.79
morgan	313.76	1.20	8.65	1.16
serve-static	276.89	1.67	7.51	1.60
prop-types	752.94	2.12	12.79	1.89
compression	279.18	1.28	6.78	1.37

Table 3. Callgraph generation and transformation timing

The first row of the table reads: for `memfs`, generating the static call graph takes 740.18 seconds and running the transformation using it takes 2.46 seconds; generating the dynamic call graph takes 15.97 seconds and running the transformation using it takes 2.72 seconds.

The runtime of the transformation itself is negligible. The longest runtime is 15 seconds on the `css-loader` project, unsurprising given that `css-loader` is the largest application in our test suite with a size of 2.76MB. There is no difference between the transformation times using the static vs dynamic call graphs. This is also unsurprising, as the same process is used to run the transformation in either case, and generally there are a similar number of stubs created. In cases such as `q`, where the dynamic call graph produces a larger stubbed application and yet it takes longer to run, this is because there are more *function* stubs being generated (compared to a single file stub being generated with the static callgraph).

The runtime of the call graph construction is more interesting. Overall, we see that constructing a static call graph takes one to two orders of magnitude more time than constructing the dynamic call graph. We also observe a correlation between the times to construct the static and dynamic call graphs. To construct the dynamic call graph, *Stubbifier* simply computes a coverage report from running an application's tests (including `node_modules`), which amounts to the time to run the tests plus some small overhead. The slower runtime of static call graph construction is due to our inclusion of the generation of the QL database in the overall runtime, which is directly proportional to the amount of code in the project (in order to run any static analysis queries, QL must build a database of the application's code—this is a one-time cost as long as the code does not change).

We envision the use-case of *Stubbifier* to be a final stage in the creation of a production release, and so we do not believe a build-time of 5-15 minutes to be prohibitive. If time is an issue or if a user wanted to apply *Stubbifier* more frequently, they can opt for the dynamic call graph.

RQ4 Takeaway: The average runtime of *Stubbifier* with the static call graph is not prohibitive (at roughly 8.3 minutes), and is much lower (28 seconds) with the dynamic call graph.

Client Proj	With guards Time (s)	Slowdown (%)	Exp. KB	Exp. KB no guards
decompress-zip	1.22	43%	240.9	63.3
downshift	1.47	3%	240.9	63.3
node-ping	4.89	22%	240.9	63.3
passport-saml	0.48	13%	0.0	0.0
requestify	3.30	12%	240.9	63.3

(a) Stubbed with static call graph

Client Proj	With guards Time (s)	Slowdown (%)	Exp. KB	Exp. KB no guards
decompress-zip	0.78	10%	16.6	3.0
downshift	1.63	13%	3.2	0.9
node-ping	4.69	19%	14.9	2.9
passport-saml	0.51	19%	0.0	0.0
requestify	3.43	15%	4.3	0.9

(b) Stubbed with dynamic call graph

Table 4. Results for Clients of q with guards enabled

RQ5: What are the effects of enabling guards?

Enabling guarded execution mode allows *Stubbifier* to report calls to functions that can execute arbitrary user code that execute in dynamically loaded code. The checks inserted by this mode do have an impact on code size and execution times, and the goal of this research question is to establish the magnitude of that effect, as well as the general usefulness of the mode.

We first consider the performance and code expansion angles: **how much does guarded execution mode affect performance and expanded code size?** To answer this, we re-ran our evaluation with guards enabled. The initial distribution sizes for the 15 applications is the same, but we noted an increase in expanded code sizes. In many cases, we found the expanded code sizes to exceed the size of the original application. This is unsurprising, as the code size overhead of the guards is significant. The full data is included in the supplemental material.

Consider Tables 4a and 4b, which report experimental data for the q package's five clients. The first row of Table 4a reads: for the decompress-zip client of q, the test suite runs in 1.22s which is a slowdown of 43% over running the test suite with an unstubbed q dependency; during these tests 240.9KB of code is unstubbed, as compared to 63.25KB of code unstubbed with unguarded stubbed q (this last column is also included in Table 2a). We can see that the expanded code size with the static call graph is just shy of 4x larger with guards enabled. The performance of the code also degrades, though the raw numbers are again fairly low—we again suspect the increased slowdown to be (mostly) due to the increased code size of dynamically loaded applications, where the program needs to load more code. That said, we did observe some significant overhead in longer-running applications, for instance slowdowns of 19% and 3% in the longer running test suites of redux's found and react-beautiful-dnd clients resp. (with the static call graph), as compared with 4% and 3% respectively without guards enabled.

Overall, we interpret these results to suggest that enabling guards leads to the same reduction in initial application size, but it does increase the expanded size with added overhead. Guards could be enabled during a testing phase where package designers want to find security vulnerabilities in

	Bundle		Stubbed Bundle Dynamic CG		Stubbed Bundle Static CG	
Package	**Size (KB)**	**Red %**	**Size (KB)**	**Red %**	**Size (KB)**	**Red %**
memfs	128	53%	10	92%	10	92%
fs-nextra	52	0%	21	60%	21	60%
body-parser	626	36%	534	15%	534	15%
commander	72	17%	1	98%	1	98%
memory-fs	100	17%	62	38%	62	38%
glob	84	2%	42	50%	42	50%
redux	22	92%	7	67%	7	67%
css-loader	962	59%	393	59%	393	59%
q	66	77%	1	99%	1	99%
send	130	43%	89	31%	89	31%
serve-favicon	18	21%	12	31%	12	31%
morgan	54	3%	30	44%	30	45%
serve-static	107	22%	95	11%	95	11%
prop-types	**NA**	**NA**	**NA**	**NA**	**NA**	**NA**
compression	23	66%	21	7%	21	7%

Table 5. Effect of stubbifying bundled projects

their applications, and to establish whether or not this is realistic we report on a case study of the depd [dep 2021e] application.

With guards enabled, *Stubbifier* reported calls to `eval` while running the test suites of `body-parser`, `send`, and `serve-static`. Upon investigation, we found that the dangerous calls were not in the code of these packages themselves, but hidden in their dependencies. Specifically, all these packages have a dependency on an *old version* of depd (`body-parser` and `send` have a direct dependency, and `serve-static` has a transitive dependency as it depends on `send`).

We confirmed that this is indeed a problem by looking at the depd project repository on Github—the current version of depd had the problematic `eval` removed. Specifically, it was removed on January 12, 2018 with commit [dep 2021a], which fixed three issues, [dep 2021b], [dep 2021c], and [dep 2021d]. These issues were filed because `eval` is not only bad practice, but its use is disallowed in Chrome apps and Electron apps, both of which are very popular platforms (we are paraphrasing points brought up in the linked GitHub issues).

To fix this issue, we tried an experiment where we removed the lock on the depd version (i.e. set it to `*`), and all client tests still work as expected.

RQ5 Takeaway: Enabling guards increases the size and performance overhead, but does catch vulnerabilities in dynamically loaded code. In particular, guarded mode allows developers to detect dangerous functions in imported modules of which developers may be unaware, and we found several examples of this in our experiments.

RQ6: How does *Stubbifier* compare to bundlers, and what is the effect of *Stubbifier* on bundled code?

Our approach is able to synergize with bundlers to create even smaller distributions. To support this, we pose and answer our sixth research question with an experiment wherein we configured each of our subject applications to use the `rollup` bundler, and applied *Stubbifier* to the bundle.

Table 5 displays the results of this experiment. The first row of this table can be read as follows: for the `memfs` project, the size of the rollup bundle is 128KB, which is a reduction of 53% from the original size of the project. When we stubbify that bundle using the dynamic callgraph as input, the result is a bundle of 10KB, which is a further reduction of 92% from the original bundle. When we stubbify the bundle instead with the static callgraph as input, the result is also a bundle of 10KB, with the same reduction of 92% from the original bundle.

In every case, we see that *Stubbifier*'s size reduction far exceeds the size reduction afforded by bundlers. Indeed, the purpose of bundlers is not to reduce application size, and that is merely a secondary benefit: the main goal of a bundler is to produce a single file that can be distributed for ease-of-use. *Stubbifier* can debloat these bundles, and in every case we find that *Stubbifier* helps bundlers create even smaller bundles. To ensure that the stubbified bundles actually work, we reconfigured the test suites of `commander`, `body-parser`, and `node-glob` to use the stubbified bundle, and found that the project tests executed as expected.

It is clear from Table 5 that there is a wide range of reduction achieved by applying *Stubbifier* to the bundles. Of particular note are `commander` and `q`, which show a reduction of 98% and 99% respectively. We investigated these bundles to determine what was causing this enormous size reduction, and in both of these cases we observed the same thing: `rollup` had put the entire module into one function, and this was a function that was not recognized as being called by our call-graph algorithms. Effectively, this meant that the entire module was stubbed, which explains the extreme size reduction. However it is not a very useful stub, since of course it needs to be expanded when the module is used.

The `prop-types` row in Table 5 has **NA** for all its entries. This is indicative of the fact that `rollup` does not work on this project. The reason `rollup` does not work here is that `prop-types` has a dependency on some BabelJS preset libraries, which throw an error when their format is changed to one compatible with the bundle.

RQ6 Takeaway: *Stubbifier* achieves significant further code size reduction over the bundled applications.

---

8 THREATS TO VALIDITY

Our approach uses an application's test suite as the entry point for call graph construction. This entwines the performance of our tool with the quality of the tests. An application with a low-quality test suite will poorly exercise application functions, thus leading to more stubs and likely more stub expansion. To mitigate this, we did not care for the *quality* of an application's tests when selecting projects for our evaluation, only that the application had tests at all. Concretely, Table 1a shows that applications have differing numbers of tests, as many as 1706 and as few as 30, with every application having over 10K LOC, suggesting that the quality of the test suites of the projects in our evaluation varies.

Also, we are cognizant that we are drawing generalized conclusions based on a subset of JavaScript projects. To mitigate potential bias in project selection, we selected projects in a systematic manner (we picked the 15 first projects that had running tests) among the most popular on `npm` as we wanted to ensure that we could find five clients for each of our subjects also with running tests.

9 RELATED WORK

Minimizing code bloat is a well-studied area of research.

Early in the paper, we described DOLOTO [Livshits and Kiciman 2008], a JavaScript debloating tool and an inspiration to *Stubbifier*. One of the main differences between our approaches is that our tool is fully automatic, debloating based on an application's tests, while DOLOTO requires a trace of a programmer interacting with the application. Further, we support a much larger JavaScript language, including all features of the 2019 JavaScript specification (ES2019), and can generate stubs at two-levels of granularity (files and functions) to their one. A notable difference is that DOLOTO caches function *objects*, while we cache *code*—as we generate function stubs for inner functions, we cannot cache function objects as they can close over outer function parameter values. Also, we do not load functions in clusters, though our addition of file stubs is similar in spirit.

Building minimal application bundles is both well-studied and prevalent in industry. Several implementations of Smalltalk developed in the 1990s (e.g., [IBM Corporation 1997; ParcPlace-DigiTalk 1995]) include features for "packaging" or "delivering" applications, and IBM's 1997 Handbook for VisualAge for Smalltalk [IBM Corporation 1997] describes a reference-following strategy to determine minimal code for a package. Compacting code is a related area, for example De Sutter et al. [2002] present Squeeze++, a link-time code compactor for low-level C/C++ code. Another facet of this area is specializing distributions: Sharif et al. [2018] present TRIMMER, which specializes LLVM bytecode applications to their deployment context using input specialization. The performance impact of using application bundles has also been studied in the context of Java, where Hovemeyer and Pugh [2001] study performance issues that arise when bundles of JVM class files for Java applications are downloaded from a server. Broadly, these approaches rely on some form of "application profile" obtained via program analysis—*Stubbifier* builds this profile via static or dynamic analysis of application tests.

In a similar vein, trimming optional functionality from applications has been studied by Bhattacharya et al. [2013], who propose an approach relying on a combination of human input, dynamic analysis, and static analysis to identify optional functionality. Koo et al. [2019] present a technique relying on manual analysis of configuration files and profiling to obtain coverage information for executions in different configurations, minimizing based on that coverage.

Much existing work is concerned with entirely removing unused code. Agesen and Ungar [1994] present an type-inference based application extractor for Self [Agesen et al. 1993] which extracts a bloat-free source file for distribution. The Jax application extractor for Java [Tip et al. 1999] relies on efficient type-based call graph construction algorithms such as RTA [Bacon and Sweeney 1996] and XTA [Tip and Palsberg 2000] to detect unreachable methods, and further relies on a specification language [Sweeney and Tip 2000] in which users specify classes and methods that are accessed reflectively, going above-and-beyond dead code elimination with, e.g., class hierarchy compaction [Tip et al. 2002]. Rayside and Kontogiannis Rayside and Kontogiannis [2002] present a tool for extracting subsets of Java libraries using Class Hierarchy Analysis [Dean et al. 1995] to identify the subset of a library that is required by a specific application, though their work does not consider unsoundness.

On the other hand, the use of code splitting techniques has been explored previously in different contexts. Besides DOLOTO [Livshits and Kiciman 2008], Krintz et al. [1999] proposed a code-splitting technique for Java that partitions classes into separate "hot" and "cold" classes to avoid transferring code that is rarely used. Wagner et al. [2011] present an optimistic compaction technique for Java applications, where minimized distributions are outfitted with a custom class loader that performs partial loading and on-demand code addition.

9.1 Control Flow Integrity

The guarded execution mode resembles works on Control Flow Integrity (CFI) verification by, e.g., Abadi et al. [2009]. A CFI policy dictates that program execution must follow a predetermined path of a control flow graph, enforced via program rewriting and runtime monitoring. Conceptually, our guarded execution mode enforces a policy where program execution cannot invoke a predefined list of functions. Zhang et al. [2013] present a CFI approach that enforces a policy preventing jumps to any but a white-list of locations, whereas our guarded mode enforces a black-list of functions. Niu and Tan [2015] develop a "per-input" CFI technique to avoid the overhead of constructing a control flow graph, and our mode avoids this altogether by pre-transforming code to intercept calls.

10 CONCLUSION

The JavaScript package ecosystem allows programmers to quickly find and import functionality into their applications, but as these applications mature they become bloated with unused, unwanted functionality from imported modules. Some applications use bundlers employing dead code elimination to try and minimize the size of production distributions, but this technique must be sure that the code it eliminates is indeed dead, a difficult fact to establish in JavaScript. We presented a technique to debloat JavaScript applications in light of the language's dynamism. Our technique replaces potentially dead code with stubs instead of removing it altogether, and these stubs are equipped to fetch and execute the original code if it was not truly dead. We implemented this technique in *Stubbifier*, and found that it significantly reduced the size of applications, and that little code is loaded dynamically with minimal overhead when an application debloated with *Stubbifier* is used in a client. We also found that *Stubbifier* out-debloats bundlers, and further *synergizes* with them to achieve significant further size reduction.

REFERENCES

2021a. Commit: remove eval. https://github.com/dougwilson/nodejs-depd/commit/887283b4. Accessed: 2021-04-16.

2021b. depd issue 20. https://github.com/dougwilson/nodejs-depd/issues/20. Accessed: 2021-04-16.

2021c. depd issue 22. https://github.com/dougwilson/nodejs-depd/issues/22. Accessed: 2021-04-16.

2021d. depd issue 24. https://github.com/dougwilson/nodejs-depd/issues/24. Accessed: 2021-04-16.

2021e. dougwilson/nodejs-depd. https://github.com/dougwilson/nodejs-depd. Accessed: 2021-04-16.

2021. mapbox/node-blend. https://github.com/mapbox/node-blend. Accessed: 2021-04-16.

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (Nov. 2009), 40 pages. https://doi.org/10.1145/1609956.1609960

Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. 1993. Type Inference of SELF. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings.* 247–267. https://doi.org/10.1007/3-540-47910-4_14

Ole Agesen and David Ungar. 1994. Sifting Out the Gold: Delivering Compact Applications from an Exploratory Object-Oriented Programming Environment. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94).* Portland, OR, 355–370. *ACM SIGPLAN Notices* 29(10).

Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In *Proc. 29th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA).*

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy.* 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

BabelJS. 2021. Babel Parser Documentation. https://babeljs.io/docs/en/babel-parser. Accessed 2021-04-16.

David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, USA, October 6-10, 1996.* 324–341. https://doi.org/10.1145/236337.236371

Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. 2013. Combining Concern Input with Program Analysis for Bloat Detection. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13).* ACM, New York, NY, USA, 745–764. https://doi.org/10.1145/2509136.2509522

Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. 2002. Sifting out the Mud: Low Level C++ Code Reuse. *SIGPLAN Not.* 37, 11 (Nov. 2002), 275–291. https://doi.org/10.1145/583854.582445

Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings.* 77–101. https://doi.org/10.1007/3-540-49538-X_5

ECMA. 2015. 262: ECMAScript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,* (2015).

GitHub. 2020. Language Trends on GitHub. https://octoverse.github.com/#top-languages.

GitHub. 2021. CodeQL. https://github.com/github/codeql. Accessed: 2021-04-16.

David Hovemeyer and William Pugh. 2001. More Efficient Network Class Loading Through Bundling. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. 127–140.

IBM Corporation 1997. *VisualAge for Smalltalk Handbook Volume 1: Fundamentals* (first edition ed.). IBM Corporation. Available from http://www.redbooks.ibm.com/redbooks/4instantiations/sg244828.pdf.

ECMA International. 2021. ECMAScript 2020 Language Specification. https://www.ecma-international.org/ecma-262/#sec-modules. Accessed: 2021-04-16.

Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 59–69.

Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of 12th European Workshop on Systems Security (EuroSec '19)*.

Chandra Krintz, Brad Calder, and Urs Hölzle. 1999. Reducing Transfer Delay Using Java Class File Splitting and Prefetching. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999*. 276–291. https://doi.org/10.1145/320384.320412

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018a. Precision-guided context sensitivity for pointer analysis. *PACMPL* 2, OOPSLA (2018), 141:1–141:29.

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018b. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 129–140.

V. 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, 2008, Atlanta, Georgia, USA, November 9-14, 2008*. 350–360. https://doi.org/10.1145/1453101.1453151

Mozilla. 2021. Rest Parameters. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters. Accessed 2021-04-16.

Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 914–926. https://doi.org/10.1145/2810103.2813644

npm. 2021a. npm. https://www.npmjs.com/. Accessed 2021-04-16.

npm. 2021b. semver. https://www.npmjs.com/package/semver. Accessed 2021-04-16.

OpenJS Foundation. 2021. Node.js. https://nodejs.org/en/. Accessed 2021-04-16.

ParcPlace-DigiTalk 1995. *VisualWorks User's Guide* (software release 2.5 ed.). ParcPlace-DigiTalk. Chapter 13: Application Delivery Tools. Available from http://esug.org/data/Old/vw-tutorials/vw25/vw25ug.pdf.

Derek Rayside and Kostas Kontogiannis. 2002. Extracting Java library subsets for deployment on embedded systems. *Sci. Comput. Program.* 45, 2 (2002), 245–270. https://doi.org/10.1016/S0167-6423(02)00059-X

Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 329–339. https://doi.org/10.1145/3238147.3238160

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. 435–458.

Stack Overflow. 2020. Developer Survey. https://insights.stackoverflow.com/survey/2020#most-popular-technologies.

Peter F. Sweeney and Frank Tip. 2000. Extracting library-based object-oriented applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering, an Diego, California, USA, November 6-10, 2000, Proceedings*. 98–107.

Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. 1999. Practical Experience with an Application Extractor for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999*. 292–305. https://doi.org/10.1145/320384.320414

Frank Tip and Jens Palsberg. 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000*. 281–293. https://doi.org/10.1145/353171.353190

Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. 2002. Practical extraction techniques for Java. *ACM Trans. Program. Lang. Syst.* 24, 6 (2002), 625–666. https://doi.org/10.1145/586088.586090

Gregor Wagner, Andreas Gal, and Michael Franz. 2011. “Slimming” a Java virtual machine by way of cold code removal and optimistic partial program loading. *Sci. Comput. Program.* 76, 11 (2011), 1037–1053. https://doi.org/10.1016/j.scico.2010.04.008

webpack-contrib. 2021. css-loader. https://www.npmjs.com/package/css-loader. Accessed 2021-04-16.

C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy*. 559–573.