Noname manuscript No.
(will be inserted by the editor)

Stubbifier: Debloating Dynamic Server-Side JavaScript Applications

Alexi Turcotte* · **Ellen Arteca*** · **Ashish Mishra** ·
Saba Alimadadi · **Frank Tip**

Abstract JavaScript is an increasingly popular language for server-side development, thanks in part to the Node.js runtime environment and its vast ecosystem of modules. With the Node.js package manager `npm`, users are able to easily include external modules as dependencies in their projects. However, `npm` installs modules with *all* of their functionality, even if only a fraction is needed, which causes an undue increase in code size. Eliminating this unused functionality from distributions is desirable, but the sound analysis required to find unused code is difficult due to JavaScript's extreme dynamicity.

We present a fully automatic technique that identifies unused code by constructing static or dynamic call graphs from the application's tests, and replacing code deemed unreachable with either file- or function-level *stubs*. Due to JavaScript's highly dynamic nature, call graph construction may suffer from unsoundness, i.e., code identified as unused may in fact be reachable. To handle such cases, if a stub is called, it will fetch and execute the original code on-demand to preserve the application's behavior. The technique also provides an optional *guarded execution mode* to guard application against injection vulnerabilities in untested code that resulted from stub expansion.

This technique is implemented in an open source tool called *Stubbifier*, designed to help package developers to produce a minimal production distribution. *Stubbifier* supports the ECMAScript 2019 standard. In an empirical evaluation on 15 Node.js applications and 75 clients of these applications, *Stubbifier* reduced application size by 56% on average while incurring only minor performance overhead. The evaluation also shows that *Stubbifier*'s guarded execution mode is capable of preventing several known injection vulnerabilities that are manifested in stubbed-out code. Finally, *Stubbifier* can work alongside *bundlers*, pop-

* These authors contributed equally to the work.

This research was supported in part by Office of Naval Research (ONR) grants N00014-17-1-2945 and N00014-21-1-2491, and by National Science Foundation grant CCF-1907727. E. Arteca and A. Turcotte are supported in part by the Natural Sciences and Engineering Research Council of Canada.

A.Turcotte, E.Arteca, F.Tip
Northeastern University, Boston, MA, USA,
E-mail: {turcotte.al, arteca.e, f.tip}@northeastern.edu

A.Mishra
Purdue University, West Lafayette, IN, USA
E-mail: mishr115@purdue.edu

S.Alimadadi
Simon Fraser University, Vancouver, Canada
E-mail: saba@sfu.ca

ular JavaScript tools for bundling an application with its dependencies. For the considered subject applications, we measured an average size reduction of 37% in bundled distributions.

Keywords debloating, program analysis, JavaScript, Node.js

1 Introduction

JavaScript is one of the most popular programming languages, and has been the lingua franca of client-side web development for years (GitHub, 2020; Stack Overflow, 2020). More recently, platforms such as Node.js (OpenJS Foundation, 2021) have made it possible to use JavaScript 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 a result, server-side JavaScript development has experienced an exponential growth in recent 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 myriad other modules. While an essential attribute of this ecosystem, in practice, *m* typically uses only a small fraction of the functionality of its dependencies, while still encompassing all of their code. In turn, clients of *m* inherit the unused functionality of *m* and its dependencies, as well as that of its own 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 the severity of this issue, there is currently no technique available that can significantly debloat a modern Node.js application while fully preserving its original behavior.

Previous work on debloating JavaScript applications has been done in the context of JavaScript *bundlers* Rollup (2021); webpack (2021). The primary goal of bundlers is to create self-contained application distributions, but they typically perform an optimization known as "tree-shaking" (MDN, 2021) on imported external modules, by removing modules or functions that are unreachable in an application's import graph. Unfortunately, the size reduction achieved by bundlers is limited by the all-or-nothing nature of their code minimization technique: code that the bundler removes must *never* be called, else the bundled application will crash. Moreover, tree-shaking can only be applied to modern JavaScript code that uses the ECMAScript module system MDN (2021).

Another approach to debloating JavaScript applications was developed by Koishybayev and Kapravelos (2020), who developed Mininode, a tool for reducing the size of *development distributions* of Node.js applications. In the JavaScript `npm` package ecosystem, a distinction is made between an application's dependencies and *development* dependencies: A dependency is another package that the application needs to function (e.g., a utility library such as `lodash`), whereas a development dependency *is only needed during development* (e.g., a test runner such as `mocha` that is needed to run the application's tests) and is not normally part of a production distribution. Mininode assumes an application's development distribution as the starting point and considers development dependencies and package tests as targets for removal. Further, Mininode completely removes code deemed unreachable through (unsound) static analysis and it only supports the ECMAScript 5 version of

JavaScript (which dates back to 2009), which lacks modern JS features such as ES6 modules, classes, and async/await. Section 4.3 reports on an experiment in which we applied Mininode to the subject applications that we used to evaluate *Stubbifier*.

Previous work on debloating in the context of other languages has focused on the use of static analysis to determine unreachable code (Agesen and Ungar, 1994; Vis, 1997, 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 (Livshits and Kiciman, 2008; Andreasen and Møller, 2014; Jensen et al., 2011; Li et al., 2018a,b; Sridharan et al., 2012), the development of a static analysis for Node.js that is simultaneously sound, precise, and scalable remains beyond the current state of the art.

This paper presents a practical technique for reducing the size of production distributions of Node.js applications while preserving their original behavior. Core application functionality, as well as the extent to which an application uses its dependencies, is inferred automatically from dynamic or static call graphs constructed from the application's own test suites (which can be comprehensive, end-to-end test suites). Rather than using a sound call graph analysis and removing the code entirely, our approach relies on a fast, scalable, unsound call graph analysis, and untested, unreachable code is replaced by *stub* versions in a technique known as *code splitting*, pioneered by the DOLOTO tool (Livshits and Kiciman, 2008). If a function or file stub is executed, it will dynamically fetch and execute the original code so as to preserve application functionality. The technique has been implemented in a tool called *Stubbifier*, designed to be used by package developers looking to prepare a minimal production distribution for their package. *Stubbifier* improves on DOLOTO by: (i) supporting all features of modern JavaScript ECMA International (2019), including classes, promises, async/await, generators, and modules, (ii) introducing file-level stubs in addition to function-level stubs (so as to achieve additional debloating by stubbing all code in files where no code is used, instead of stubbing each of the functions in these files individually), and (iii) providing an (optional) *guarded execution mode*, where stubbed-out code is automatically instrumented to intercept calls to functions such as `eval` and `exec` that may introduce injection vulnerabilities. Most importantly, (iv) *Stubbifier* is *fully automatic* by relying on static analysis or dynamic analysis of the application's test suite to identify code that is likely to be unreachable, whereas DOLOTO required traces of users interacting with the subject application to establish core application functionality.

Stubbifier was evaluated on 15 of the most popular Node.js applications, using five clients for each subject application to evaluate how much code is loaded dynamically. This evaluation found that *Stubbifier* achieves significant size reductions (56% on average), that the number of stubs expanded during the execution of client applications is relatively small, and that minimal performance overhead is incurred. Further, experiments with *Stubbifier*'s guarded execution mode confirmed that it is capable of preventing known injection vulnerabilities. Finally, we confirmed experimentally that, when used in conjunction with the popular `Rollup` bundler, *Stubbifier* achieves significant additional size reductions on previously bundled applications (37% on average).

In summary, this paper makes the following contributions:

- A fully automated technique for reducing the size of Node.js applications while preserving their original behavior, based on a combination of static or dynamic analysis and code splitting.

- The implementation of this technique in a tool called *Stubbifier* that supports modern JavaScript ECMA International (2019). *Stubbifier* is publicly available as an open-source tool[1], and a self-contained code artifact including reproducible experiments is also available on Zenodo (Turcotte and Arteca, et al., 2021).
- An empirical evaluation of *Stubbifier* on 15 open source Node.js applications and 75 clients of these subject applications (five clients per subject), showing that *Stubbifier* reduces the size of Node.js applications by 56% on average while incurring only minor performance overhead. The evaluation also shows that *Stubbifier*'s guarded execution mode is capable of preventing several known injection vulnerabilities that are manifested in stubbed-out code.

2 Background and Motivation

The `npm` ecosystem includes more than 1.7 million modules[2] that provide a wealth of convenient features. By importing these libraries and reusing their functionality, programmers can focus their efforts on features that are unique to their application. However, this convenience does not come without its price: importing modules can cause projects to become excessively large due to the transitive importing of other projects that they depend on. In practice, it is often the case that a module only uses a small subset of the functions in the transitive closure of its dependencies.

To illustrate this, 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`. `css-loader` is one of the most popular modules on `npm`, with nearly 15 million weekly downloads, and it is imported by over 15,000 modules.

`css-loader` has 13 third-party production dependencies[3] (i.e., modules upon which its functionality depends). The stand-alone `css-loader` module contains only 16 files comprising 110KB. However, installing `css-loader` with direct and transitive production 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 part of the resulting installation constitutes application bloat, we examined `css-loader` to determine which functions and files are reachable from the application's test suite. Using a simple static analysis that traces function calls to build a list of unreachable functions and files, 209 files were found to be potentially unreachable, and 6 unreachable functions were identified in otherwise reachable files. Given the extreme dynamicity of JavaScript, sound static analysis is not possible (see, e.g., Richards et al. (2011); Sridharan et al. (2012); Jensen et al. (2012); Madsen et al. (2015); Stein et al. (2019)). Since in practice all static analyses for JavaScript suffer from unsoundness, some of the functions and files that they identify as being unreachable may indeed be reachable. Nevertheless, if one *could* devise a technique to remove all of this code, the application's size would be reduced by 80%.

Consider `semver` (npm, 2021b), a package that `css-loader` depends on. Only *two* functions from `semver` are used in `css-loader`: The `satisfies` function is imported

[1] See https://github.com/emarteca/stubbifier.

[2] See `http://www.modulecounts.com/`

[3] Many npm modules rely on additional development dependencies (sometimes referred to as "devDependencies") that are needed only for development purposes, e.g., for running tests. These dependencies are typically not installed by clients.

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`.

In subsequent sections, we will describe our approach to addressing this issue of code bloat, and describe our implementation of this approach in a tool called *Stubbifier*. The intended user of our tool is a package developer looking to prepare a minimal production distribution for their package, e.g., a developer of `css-loader` might run *Stubbifier* on the package before a release. *Stubbifier* identifies the extent to which `css-loader`'s dependencies are used. For example, one of `css-loader`'s dependencies is `semver`: this developer would note that *Stubbifier* identifies 27 of `semver`'s files and six `semver` functions inside of `css-loader` to be potentially unreachable, i.e., `css-loader` imports `semver` but only uses a subset of imported functionality[4]. 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. Overall, the size of `css-loader` as a whole is reduced by 80%, from 2.8MB to 0.6MB.

Note that the majority of the code removal is in the *dependencies* of `css-loader`. To illustrate, note that the initial size of `css-loader`, before installing any dependencies, is 110KB. The size of `css-loader` with all dependencies installed is 2.8MB, and *Stubbifier* reduces this to 0.6MB. This is 2.2MB of reduction, and since the original size of `css-loader` is just 110KB, *Stubbifier* must have mostly removed code in the dependencies of `css-loader`.

`css-loader` has approximately 14.8 million weekly downloads, so an 80% size reduction would translate to a reduction in weekly data transfer from 41.4TB to 8.8TB. We will elaborate on this in Section 4.

The next sections will present our debloating technique and its evaluation.

3 Approach

The debloating technique presented in this paper involves several key steps, illustrated in the diagram in Figure 1. First, a call graph is computed for the project using its own tests as entry points, either dynamically by running the tests with instrumentation, or statically by running a static analysis. The project source code and call graph are input to the debloating algorithm: the technique essentially replaces functions and files that are *not* in the call graph with stubs, which are smaller but are equipped to fetch and load the code dynamically if they are invoked. The end result is a debloated project that is ready to deploy.

We envision this technique to be used by developers that wish to create minimal distributions for their applications. The purpose of using an application's tests to infer unused functionality is to automatically determine the extent to which the application exercises its direct and transitive dependencies: If application tests had 100% coverage and the application fully exercised its dependencies, then no stubs would be introduced. In practice, however, a package will not use all of the code in its dependencies (e.g., `css-loader` includes `semver` but uses only a few of its functions). Note that the ideal scenario is when an application's tests have 100% *application* code coverage: in such cases, the unused parts of the application's *dependencies* would be replaced with stubs and nothing would ever be loaded dynamically.

The remainder of this section will discuss each step of the approach in detail.

[4] There is more unreachable code in `css-loader`, but we focus on `semver` for the sake of illustration

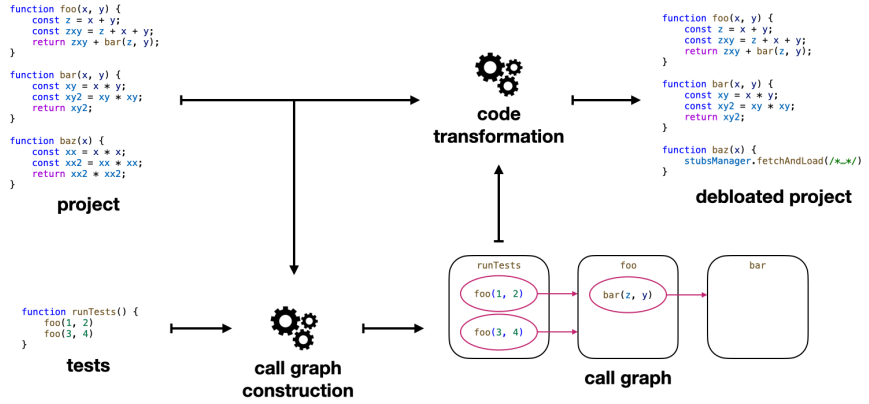

Fig. 1: Overview of approach. A **call graph** is computed from a **project** using its **tests** as entry points. A **call graph construction** algorithm maps call sites to functions; e.g., here the `runTests` function contains two function calls `foo(1, 2)` and `foo(3, 4)`, both of which are mapped to the `foo` function, which contains a call `bar(z, y)` that is mapped to `bar`. Then, our debloating technique performs a **code transformation** to replace unreachable code (according to the call graph) with stubs that can dynamically load the code on-demand. Function `baz` is deemed unreachable since it does not appear in the call graph, and hence is replaced with a stub.

3.1 Call Graph Construction

In principle, any call graph can be used to determine which files and functions should be replaced with stubs. The soundness and precision of the call graph will impact the size of the initial distribution and the amount of code that needs to be loaded dynamically.

The implementation of *Stubbifier* includes mechanisms for constructing a static or dynamic call graph. In each case, *Stubbifier* uses the test suite of the input application as the entry point for the analysis, and so the call graph represents the *tested* code. Any function that is not in the call graph is deemed unreachable and untested and will be replaced with a (*function-level* or *file-level*) stub. Both analyses are configured to consider depended-upon modules (in the `node_modules` directory), though note that development dependencies are excluded as they are typically not packaged and shipped with the subject application.

Below, we provide some further detail on the specific static and dynamic call graph construction techniques that *Stubbifier* supports.

Dynamic Call Graphs. To compute *dynamic* call graphs, code coverage is determined using Istanbul's command line tool `nyc` (Istanbul, 2021), that computes statement, line, branch, and function coverage for Node.js applications. By default, `nyc` ignores a project's dependencies, but *Stubbifier* automatically generates a configuration file that specifies that coverage of *non-development*, production dependencies should be computed. *Stubbifier* then runs `nyc` on the application's tests, to determine which functions and files are invoked during testing (and by exclusion, which were not invoked).

Static Call Graphs. To compute the *static* call graphs, we developed an analysis using CodeQL (Avgustinov et al., 2016), GitHub's declarative language for static analysis, using its extensive libraries for writing static analyzers (GitHub, 2021). In particular, CodeQL's dataflow library contains functionality for tracking calls through local module imports, and we implemented an extension to recognize modules in a project's `node_modules` directory, and extended CodeQL's libraries to track data flow through these modules. Then, a call graph construction algorithm was implemented on top of this analysis, which uses the application's tests as entry points for the analysis.

This is an unsound analysis, as the use of dynamic features such as `eval` and dynamic property access expressions may give rise to missing edges in the call graph. We found that these dynamic features are so prevalent in modern JavaScript applications that using a sound, conservative call graph analysis is impractical (making conservative assumptions in the presence of these dynamic features would result in almost all code to be deemed reachable). In our approach, reachable code mistakenly classified as unreachable due to the unsoundness of the analysis does not result in an error when called: rather, it is dynamically loaded via the stub.

3.2 Introducing Stubs

After constructing a call graph, *Stubbifier* creates lists of unreachable functions and files. Here, *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.

Next, *Stubbifier* parses the application's source code, including any dependencies, and replaces unreachable functions and files with *stubs* via transformations on the program's Abstract Syntax Tree (AST). Note that *Stubbifier* does not replace functions or files with stubs if they are shorter than the stubs that would replace them.

File Stubs. Each unreachable file is replaced with a *file stub*. The code in this 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: (i) performing all imports that were in the original code (line 1), (ii) fetching the original code and evaluating it (lines 2-3), (iii) replacing the contents of the stubbed file with the original file (line 4), and finally (iv) performing necessary exports (line 5). More specifically, in files that rely on the CommonJS mechanisms (i.e., `require` for importing and `module.exports` for exporting), simply storing the original code elsewhere and `eval`-ing it as needed suffices, as these mechanisms can be used anywhere in a source file. However, the ECMAScript Module System (ESM) (ECMA International, 2021)'s static `import`/`export` constructs cannot be executed in an `eval` (see

section 15.2 of ECMA International (2019)), so all `import` and `export` statements are hoisted 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 2.

```
1  // file.js before stubs are introduced
2  export function foo() { /* ... */ }
3  import { A };
4  function bar() { /* ... */ }
5  export default bar;
6
7
8  // file.js after stubs are introduced
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. 2: File before and after stubs are introduced.

In Figure 2, we see `import` and `export` statements interspersed through the file before stubs are introduced. 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).

To allow this exporting, the original code from Figure 2 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 3.

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

Fig. 3: 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 2).

Function Stubs. Functions deemed unreachable are replaced with *function 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,

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;
```

either by retrieving 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 (the conditions where this is or is not possible are discussed below). If not, the code is cached, and then the function is executed.

Stubbifier's caching strategy differs from DOLOTO's (Livshits and Kiciman, 2008): where DOLOTO caches function objects, *Stubbifier* caches the code, and we discuss the reasoning behind this shortly.

A concrete example of a function stub can be found in Figure 4, where we show the stub for `getValidHeaders` from the `node-blend` (nod, 2021) project.

```
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. 4: Example of a stubbed function.

First, note that *Stubbifier* outfits each file with a global `stubs` object containing the code cache and functionality to fetch code. We see a call to `stub.getCode("UID_for_LOC")` on line 21, which fetches the *original function definition* (found through `"UID_for_LOC"`, a unique ID for the 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[5]. Finally, line 23 redefines the `getValidHeaders` with the expanded stub,

[5]Recall that in JavaScript functions are objects, and can have properties assigned dynamically.

```
26  function(headers) {
27      let toExecString = stubs.getStub("UID_for_LOC");
28      if (! toExecString) {
29          toExecString = stubs.getCode("UID_for_LOC");
30          stubs.setStub("UID_for_LOC", toExecString);
31      }
32      let toExec = eval(toExecString);
33      toExec = stubs.cpFunProps(this, toExec);
34      return toExec.apply(this, arguments);
35  }
```

Fig. 5: Example of stubbed anonymous function.

and line 24 calls the function with its original arguments[6]. 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.

The above discussion covered the general approach for introducing function stubs. However, several types of functions require special treatment, as will be discussed next.

Anonymous Functions. In JavaScript it is possible to create a function without a name, an idiom that is commonly seen when functions are passed as callback arguments to higher-order functions. In these cases, the function cannot reassign itself as is done on line 23 in the above example (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 5 displays the `getValidHeaders` stub that we would create if this function did not have a name. Here, 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.

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. This is necessary because, in JavaScript, functions are *closures* that *close* variables from surrounding scopes directly into the object. Therefore, generating a stub for a function that is nested inside another would include the *function arguments* of the latter in its 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 does not arise for functions with a name, as the function reassigning itself does not store a closure.

DOLOTO cached function closures, which is problematic for the reasons discussed above; we conjecture that the authors did not evaluate their tool on code where this issue would arise.

Class and Object Methods. When replacing object or class methods with stubs, an issue arises that relates to 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.

[6] `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.

Fortunately, class and object methods can be accessed as *properties* of `this`, and so if `getValidHeaders` were a method in a class, 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. Specifically, this means that instead of `this.functionName` we use `this[${generate(key)}$]`, where `${generate(key)}$` is a string generated at parsing runtime, to reference the function as a dynamic property access on `this`.

Classes and objects can also have *getter* and *setter* methods, as is illustrated in the example below:

```
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` were an arrow function, it would be written as:

```
let getValidHeaders = (headers) => { /* elided function body */ }
```

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 fairly rare; we only encountered a few instances of unreachable constructors or generators in our evaluation.

Manually specifying functions *not* to stub. Users may be interested in specifying some functions that should never be replaced with stubs, regardless of their classification in the generated call graph. To accommodate this, we added functionality to allow users to manually flag a function so it will be ignored by *Stubbifier*. To illustrate the usefulness of this feature, consider a developer that has been using *Stubbifier* for some time. This developer may note that *Stubbifier* classified some function as unreachable, but that function is nearly always loaded dynamically in clients of the developer's package. Instead of writing tests for this function (e.g., perhaps the function is difficult to write unit tests for), the developer can configure *Stubbifier* to ignore that function to avoid it needing to be loaded dynamically.

3.3 Guarded Execution Mode

Since *Stubbifier* builds the input call graph using the application's tests, the stubbed code is also the *untested* code. Dynamically loading and executing this code could pose a security risk, as it may include injection vulnerabilities that were not encountered during testing.

To address such concerns, *Stubbifier* includes an option to detect calls to a pre-specified list of "dangerous" functions in expanded code. This is achieved by intercepting all function calls and checking 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`, and `child_process.{fork, exec, execSync, spawn}`, common functions that enable the execution of 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, when a stub is expanded, the loaded code includes guards.

Since these functions could be aliased, we must wrap *every* function call with these checks. As such, the size of the loaded code (i.e., the expanded stubs) is increased dramatically. The guards also incur more runtime overhead, as will be discuss in Section 4.

3.4 Asynchrony

JavaScript is a single-threaded language, and so our approach need not deal with the multi-threaded setting. That said, JavaScript does provide several mechanisms for asynchronous programming (event-driven programming, promises and async/await), and the use of these features may give rise to imprecision and unsoundness during call graph construction. Our approach addresses this by not assuming soundness in the first place–stubs are introduced in cases where the analysis has identified a function as being unreachable. Unsoundness will cause more stubs to be introduced, which increases runtime overhead when they are expanded.

3.5 Bundler Integration

Many JavaScript projects use *bundlers* such as `webpack` (webpack, 2021) and `Rollup` (Rollup, 2021) to package an application along with all the modules that it depends on into a single-file distribution that includes all required functionality. Such a bundle can be

included in another application using `require` or `import`, so that users do not need to go through additional installation steps.

Bundlers perform a limited form of code debloating known as "tree-shaking", which identifies functions and classes that are unused based on a static analysis of the import relationships between modules. If the project relies on `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, parts of an imported module can be removed if they are not referenced in the importing module. The size reductions that can be achieved using tree-shaking can be significant, but they are still limited by the fact that soundness is required, because the removal of code that is used could cause a bundled application to crash.

The use of *Stubbifier* in combination with a bundler requires a few additional steps in the previously discussed transformation pipeline. First, bundling must always happen *before* applying *Stubbifier*, as bundlers perform their own code transformations. For example, when merging all the application code into a single file, bundlers often refactor the code so as to avoid variable name conflicts, repeated imports, etc. If *Stubbifier* were run on the application before bundling, the bundler would only perform its analysis on the code that is not replaced with stubs, since the code to be loaded dynamically is just stored as plain text. As a result, expanding a stub would result in code that does not match, e.g., the changed variable names in the bundle, which is likely to result in errors.

To prevent such issues, *Stubbifier* should be applied to an application *after* it has been processed by a bundler. One minor obstacle here is that *Stubbifier* uses an application's tests as the entry points for call graph construction, and tests are nearly always based on the original project source code, and not on a bundle. To address this, *Stubbifier* determines a mapping of the functions to be stubbed from their positions in the original code to their positions in the bundle. Then, it constructs call graphs from tests as discussed before, and it consults the mapping to determine where stubs should be introduced in the bundle. This is all illustrated in the diagram in Figure 6, which is expanded from the diagram in Figure 1. The major difference with the approach illustrated in Figure 1 is the use of a bundler *after* the call graph is computed, but *before* debloating; the result of this entire process is a debloated application bundle.

The evaluation presented in Section 4 will examine how much additional code size reduction can be achieved by *Stubbifier* on applications after they have been bundled using `Rollup`.

4 Evaluation and Discussion

This section presents an evaluation of *Stubbifier* that aims to answer the following research questions:

- **RQ1.** How much does *Stubbifier* reduce application size, and which type of call graphs (static or dynamic) is more effective for reducing application size?
- **RQ2.** How much code is dynamically loaded due to stub expansion?
- **RQ3.** How much overhead is incurred due to stub expansion?
- **RQ4.** How much time does *Stubbifier* need to transform applications?
- **RQ5.** How much run-time overhead is incurred by guarded execution mode and can it detect security vulnerabilities?
- **RQ6.** How much does *Stubbifier* reduce the size of applications that have been bundled using `Rollup`?

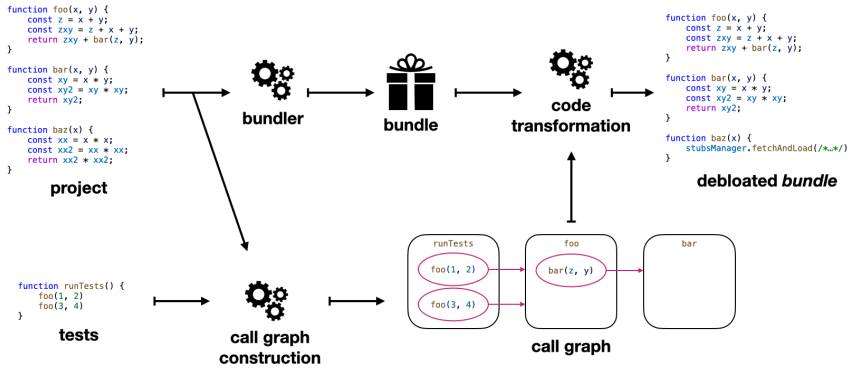

Fig. 6: *Stubbifier* overview with bundler integration. As in Figure 1, a call graph is computed from the project and its test suite, but here the project is bundled before being transformed. The code transformation is applied to the bundle w.r.t. the call graph computed from the application (note: test suites typically rely on the non-bundled application, which is why the call graph is not computed over the bundle directly).

4.1 Experimental Setup and Methodology

To evaluate *Stubbifier*, we selected 15 projects from the most popular projects published by `npm`; specifically, we listed projects in descending order by number of weekly downloads, and went down this list, selecting a project if it met the following two criteria: first, we required that the project installed, was able to build without error, and had a running test suite with no failing tests (as *Stubbifier* uses the test suite to generate call graphs). If a project satisfied these criteria, we then randomly selected from its *dependents*, or *clients*, and attempted to install, build, and run their tests; if the project had five such clients, it was selected for our evaluation. The selection criterion that was the most difficult to satisfy is that subject applications needed to have at least 5 client packages that had fully passing test suites. The availability of such client packages is critical to our evaluation since this provides us with a way to assess frequency and cost of stub expansion in a realistic setting (since we introduce stubs in an application based on its own tests, running the same tests to evaluate stub expansion would have yielded biased results).

The intended user of *Stubbifier* is a package developer: when the developer is ready to prepare a production distribution of their package, they can run *Stubbifier*. Based on the application's tests, *Stubbifier* will determine the extent of the package code that is reachable, *as well as the extent that the package exercises its dependencies*. Thus, it is likely that *Stubbifier* will remove large swathes of the package's production dependencies. The developer is left with a minimal distribution that they should feel safe distributing to users. The evaluation described here is intended to simulate that experience: we run *Stubbifier* on a package, and then insert the stubbified version of that package in five of the package's dependents to confirm that the debloated distribution works, and evaluate the extent to which our technique was effective (by running the tests of the dependents).

Table 1a lists the projects used for the evaluation, as well as some relevant metrics. The first row reads: the project `memfs` has 18k lines of code (LOC) in the analyzed files[7], and there are 133 files analyzed (Num files). The `memfs` test suite has 284 tests, which have a coverage of 80.7% of the source code of the project (Coverage: Src), and a coverage of 37.23% of its production dependencies (Coverage: Deps). `memfs` has one production dependency (Deps), and its analyzed code comprises 146 KB (Size). Note that the number of dependencies includes both direct and transitive dependencies.

We have created a code artifact (Turcotte and Arteca, et al., 2021) to accompany this paper: the artifact includes each project cloned at the version on which we ran the evaluation, the experimental infrastructure used to conduct said evaluation, as well as the full source code of *Stubbifier*.

Selecting subject applications. Each subject application was processed twice with *Stubbifier*, once using static call graphs and once using dynamic call graphs. In each case, files and functions deemed unreachable were replaced with stubs. To address **RQ4**, the time required for the entire process was measured. For **RQ1**, the size of the application before and after introducing stubs was compared. We compute the size of source code (excluding tests), *including* production dependencies and *excluding* development dependencies.

To address **RQ2** and **RQ3**, we selected five clients of each subject package from its list of dependents that is published on `npm`. These clients were essentially selected randomly, but we excluded clients without tests or with failing tests. We also confirmed that the dependency is actually used in the client: there are some projects that list a package as a dependency but no longer use in the source code, and we excluded these. Finally, we exclude clients that require the use of older versions of Node.js. We did not run an application's own test suite to explore **RQ2** and **RQ3** because it was debloated based on those very same tests. Further, the use case envisioned is that of a developer debloating their own project, and clients importing the debloated version; examining how the debloated version of a project behaves in the setting of one of its clients replicates this.

Conducting performance measurements. To determine the performance overhead caused by stub expansion, we compared the runtime of each of these clients' tests when using the stubbed and original subject application. When running the test suite with the stubbed application, we also tracked the total size and number of stub expansions to determine *how much* code is loaded dynamically. In our evaluation, stubs were loaded from local storage on the machine running the evaluation.

To mitigate noise and bias caused by caching, all test suites were executed 10 times after two test runs before the timed experiments; the reported results are the average of these 10 test runs. Furthermore, since some of the tests generate files in `/tmp`, this directory is cleared 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.

[7] The metrics in the table reflect the project's own source code (excluding tests), and all its (transitive) production dependencies, but excluding `devDependencies`. .

Project (citation)	Commit	LOC	# files	# tests	Test Coverage Src	Test Coverage Deps	Deps	Size (KB)
memfs (mem+2021a)	a9d2242	18k	133	284	80.7%	37.2%	1	146
fs-nextra (fsn+2021)	6565c81	11k	184	138	99.0%	99.0%	0	52
body-parser (bod+2021)	480b1cf	20k	210	231	99.7%	29.6%	21	364
commander (com+2021b)	327a3dd	13k	177	351	48.8%	48.8%	0	70
memory-fs (mem+2021b)	3daa18e	14k	167	44	97.4%	58.9%	11	120
glob (glo+2021)	f5a57d3	13k	175	1706	95.9%	72.0%	10	86
redux (red+2021)	b5d07e0	105k	4491	82	96.9%	0.5%	2	267
css-loader (css+2021)	dcce860	71k	1299	430	99.3%	4.88%	36	2764
q (q+2021)	6bc7f52	16k	135	243	42.9%	14.9%	0	281
send (sen+2021)	de073ed	14k	157	152	100%	68.5%	17	97
serve-favicon (ser+2021a)	15fe5e3	10k	121	30	100%	58.8%	5	20
morgan (mor+2021)	19a6aa5	14k	159	81	100%	73.6%	8	55
serve-static (ser+2021b)	94feedb	13k	160	90	100%	48.4%	19	106
prop-types (pro+2021)	d62a775	15k	152	287	98.0%	1.48%	4	106
compression (com+2021a)	3fea81d	13k	149	38	100%	40.6%	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

- Transform the dependency. The symbolic link means the client needs no change to use the stubbed version of the dependency.
- Rerun the client's tests, now with the stubbed version of the dependency.

All our experiments were conducted on a Thinkpad P43s with an Intel Core i7 processor and 32GB RAM, running Arch linux, using 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 the results.

Guarded execution mode. For **RQ5**, we repeated the experiments with guards enabled, and measured the running time and size of expanded code for the client test suites to determine the increase in overhead due to these extra checks.

In addition, we report on a case study involving `depd` (dep, 2021e), a subject application with a known vulnerability, and on experiments with `osenv` and `node-os-uptime`, two `npm` modules with confirmed vulnerabilities that were used as experimental subjects in Karim et al. (2018)[8].

Bundlers. For **RQ6**, each subject application was bundled use the `Rollup` bundler (Rollup, 2021). This involved the creation of a bundler configuration file (which we generated automatically given the application's `package.json` file) to bundle the application based on its listed entry points and to create a single bundle that also includes all of its production dependencies[9]. We measure and report on the sizes of the resulting bundle, both with and without having applied *Stubbifier*, to determine what additional size reduction is enabled by *Stubbifier*.

4.2 Overview of Results

The results of running *Stubbifier* on the projects are displayed in Tables 1b and 1c. 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 call graphs.

Note that the size immediately after transformation is only representative of the stubbed application size if no stubs are expanded. To gain a realistic estimate of the size reduction in a standard use-case of the application, we identified five clients for each application and tracked how many stubs were expanded during the execution of the test suites of these clients. 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 size of the `memfs` source code is reduced 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.

[8] Of the subject applications reported on in Karim et al. (2018), these were the only two that had a confirmed vulnerability and a test suite with passing tests.

[9] The default behavior of `rollup` is to ignore dependent modules in `node_modules`, but the bundle should all code in which stubs may be introduced, to be able to determine *Stubbifier*'s effectiveness.

In the remainder of this section, we will address each research question in order.

RQ1: How much does *Stubbifier* reduce application size, and which type of call graphs (static or dynamic) produces smaller applications?

We refer the reader to Tables 1b and 1c. In these tables, it can be seen that, using static call graphs, size reductions ranging from 2% to 87% are achieved (56% on average). The case where a size reduction of only 2% is achieved is `commander`, which has no dependencies and appears to be a bit of an outlier. Using dynamic call graphs, size reductions ranging from 3% to 89% achieved (31% on average).

Overall, the use of static call graphs results in larger size reductions in 11/15 cases, and in larger size reductions on average (56% on average when static call graphs are used vs. 31% when dynamic call graphs are used). This is not surprising, as both static and dynamic call graph constructions use the test suite as the entry point of the application, and the static analysis suffers from unsoundness due to the dynamic nature of JavaScript. Since the static analysis is constructing a call graph, unsoundness might cause some functions to be excluded from the call graph when they are actually executed in the test suite. As a result, the initial code size reduction is therefore usually larger, but more stubs need to be expanded at run time. The dynamic analysis finds every function that is called during the test suite execution, since it is constructed with a coverage tool. If the static analysis was perfectly precise then it would produce the exact same call graph as the dynamic analysis.

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 (for one project!).

On average, *Stubbifier* reduces initial application size by 56% when using static call graphs, and by 31% when using dynamic call graphs.

RQ2: How much code is dynamically loaded due to stub expansion?

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 than (or equal to) 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 maximum size increase is $>$ 2x in only two cases.

The case where `glob` is processed using a dynamic call graph is an interesting outlier, as its size is *larger* than the original code after all stubs have been expanded. This is because not much of `glob` is stubbed (the initial size reduction is only 4%, or 2KB), and the code required to support stub expansion is larger than the initial size reduction due to the extra boilerplate that was introduced by *Stubbifier* (import statements, `eval` call, reassignments to imports, etc.).

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 to

	Client		Stubbed code: effect of expansions				
Proj	**Client Proj**	**Time (s)**	**Time (s)**	**Slowdown (%)**	**Files**	**Fcts**	**Exp (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**	**Exp (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

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 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.

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

RQ3: How much overhead is incurred due to stub expansion?

To determine the performance overhead introduced by stub expansion, we measured the running times of the test suites of clients of applications processed by *Stubbifier*.

We decided not to 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. Instead, we conducted a case study on the effect of the dynamic code loading for the individual clients of the three projects presented in Tables 2a and 2b. The results for all test applications are included in the supplementary material, but the trends are upheld across the full data.

Referring to the time columns of Tables 2a and 2b, the following conclusions can be drawn. First, a correlation between the slowdown and the number of stub expansions can be observed: as more code is dynamically loaded, the performance overhead increases. This aligns with our expectations, as stub expansions involve additional I/O and compute time. That said, the runtime overhead is never extreme, and the slowdowns still leave the running times 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.

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

	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

RQ4: How much time does *Stubbifier* need to transform applications?

Table 3 shows the time needed by *Stubbifier* to process each of the 15 projects. Here, we distinguish between the time needed to construct call graphs, and the time needed to transform the source code. Note that as the execution of the project test suite is a necessary step for constructing the dynamic callgraph, the dynamic callgraph generation time includes the time required to run the tests.

The first row of the table reads: for `memfs`, generating the static call graph takes 740.18 seconds and applying transformations based on this call graph takes 2.46 seconds. Furthermore, generating the dynamic call graph takes 15.97 seconds and applying transformations based on this call graph takes 2.72 seconds.

From the table, it can be seen that the cost of the code transformation itself is negligible. The longest runtime is 15 seconds on the `css-loader` project, which is unsurprising given that `css-loader` is the largest subject application (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, a similar number of stubs is 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 when using static call graphs).

The cost of call graph construction is more noteworthy. 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 CodeQL 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, CodeQL must build a database of the application's code—this is a one-time cost as long as the code does not change).

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

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 that a build-time of 5-15 minutes to be prohibitive. If a user wanted to apply *Stubbifier* more frequently, they could opt for using dynamic call graphs.

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.

RQ5: How much run-time overhead is incurred by guarded execution mode and can it detect security vulnerabilities?

The use of "dangerous" functions such as `eval` and `exec` that interpret string values as code is known to cause injection vulnerabilities in JavaScript applications Karim et al. (2018). It is particularly concerning if such functions are invoked from untested code, because it means that the developers may not have considered all situations where calls to such functions are executed. *Stubbifier*'s guarded execution mode aims to mitigate this risk, by adding dynamic checks for such functions in stubbed-out code so that a warning can be issued or execution can be terminated when such calls are encountered. These dynamic checks may have a noticeable impact on code size and execution times, and research question RQ5 aims to establish the magnitude of that effect.

We first consider performance and code size by repeating the experiments in guarded execution mode. The initial distribution sizes for the 15 applications is the same, but we noted an increase in expanded code sizes, which in many cases now exceeds the size of the original application. This is unsurprising, as the code size overhead of the guards is significant. Consider Tables 4a and 4b[10], which report experimental data for the `q` package's five clients. The first row of Table 4a reads: for the `decrompress-zip` client of `q`, the test suite runs in 1.22s which is a slowdown of 43% over running the test suite with the original `q` package. A performance hit is expected, as the expanded code is now running an

[10]The full data for all applications is included in the supplemental material.

additional conditional check around *every* function call to check if the function being called is in the specified list of dangerous functions. Moreover, during these tests 240.9KB of code is expanded, as compared to 63.25KB of code being expanded without guarded execution mode (this last column is also included in Table 2a). Note that, when using static call graphs, the expanded code size is almost 4x larger when guards are 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 fact that 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, respectively (when using static call graphs), as compared with 4% and 3% respectively without guarded execution mode.

Detecting security vulnerabilities. When guarded execution mode was enabled, calls to `eval` were intercepted in running the test suites of three subject applications: `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 one their dependencies*. Specifically, all these packages rely on an *old version* of `depd` (dep, 2021e): `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 examining the `depd` project repository on Github and found that the problematic `eval` 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. To fix this issue, we removed the lock on the `depd` version (i.e., set it to `*`) to get the applications to use the current version of `depd`, and confirmed that all client tests still pass.

To further test the effectiveness of guarded execution mode, we ran another experiment involving two other applications with known vulnerabilities: `osenv` and `node-os-uptime`. These projects were used as experimental subjects[11] in the evaluation of a dynamic taint analysis (Karim et al., 2018) that detected vulnerabilities in them. In both projects, a function containing a call to a dangerous function (`exec` in the case of `osenv` and `execSync` in the case of `node-os-uptime`) was stubbed out by *Stubbifier*. We created a new test containing the same code fragment that was used in (Karim et al., 2018) to detect the vulnerability, and confirmed that the guard introduced by *Stubbifier* was triggered when the test was executed.

Guarded execution mode allows developers to detect injection vulnerabilities in imported modules of which developers may be unaware, and we found several examples of this in our experiments.

RQ6: How much does *Stubbifier* reduce the size of applications that have been bundled using `Rollup`?

To answer this research question, we conducted an experiment where we applied the the `Rollup` bundler to each subject application, and applied *Stubbifier* to the resulting 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

[11] Of all the subject applications considered in (Karim et al., 2018), these are the only two that still build, install, and have a test suite with passing tests, as required by *Stubbifier*.

			Stubbed Bundle			
	Bundle		Dynamic CG		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%	47	35%	47	35%
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%	53	19%	53	19%
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

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.

Not all of the applications lend themselves well to bundling. `commander` and `q` are configured such that when the bundler is applied, the entire package is wrapped in a single function that is called to generate the module exports. Since this function does not exist in the original module, it is not detected as reachable from the application's tests (since these exercise the original, un-bundled application). To address this, we configured *Stubbifier* to prevent it from replacing 4 functions with stubs (one in `commander`, and three in `q`) (recall from Section 3.2 that programmers can specify in a comment that *Stubbifier* should not stub a function or file). Beyond these, `prop-types` could not be bundled as it depends on some BabelJS libraries that throw errors when the code format is changed by the bundler, and `fs-nextra` has no dependencies so bundling it does not reduce its size at all.

That said, in every case, we see that *Stubbifier* achieves additional size reductions on applications after they are bundled, with an average of 37% further size reduction. 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, and *Stubbifier* reduces the size of all of these bundles.

To confirm that the debloated bundles behave as expected, we conducted an experiment in which we reconfigured the test suites of `commander`, `body-parser`, and `node-glob` to use the debloated bundle[12], and found that the project tests executed as expected.

Stubbifier achieves significant code size reductions when applied to bundled applications, by reporting a further size reduction of 37% on top of the reduction already afforded by bundlers.

[12] In general, adapting application test suites to work with a bundled version of the application instead of the original version can be a complex and error-prone process, as test suites may import specific functions (that may be renamed by the bundler) from specific files (that may be combined by the bundler). For the applications mentioned here, this conversion was straightforward.

4.3 Comparison with Mininode

Like *Stubbifier*, Mininode (Koishybayev and Kapravelos, 2020) is a tool for reducing the size of Node.js applications, but there are fundamental differences between the two tools, which we explore and evaluate in this section.

Mininode relies on a static analysis to determine code that is unused and that should be removed. Code can be removed at one of two levels of granularity: "coarse", where entire modules are removed, or "fine", where individual functions are removed. For the "fine" mode, Mininode makes use of an unsound static analysis to build a call graph of the application, using as the entry point the main file specified in the `package.json` of the project. Mininode also removes non-code artifacts such as license and configuration files.

There is a significant difference in the types of distributions used to evaluate Mininode and *Stubbifier*. In the JavaScript npm package ecosystem, a distinction is made between an application's dependencies and development dependencies: A *dependency* is another package that the application needs to function (e.g., a utility library such as lodash), whereas a *development dependency* is only needed during development (e.g., a test runner such as mocha that is needed to run the application's tests) and is not normally part of a production distribution. Mininode assumes an application's development distribution as the starting point and considers development dependencies and package tests as targets for removal. By contrast, in our work, we assume the production distribution of a package to be the starting point (which already excludes development dependencies and tests). Therefore, we do not consider development dependencies and test code when reporting results obtained with *Stubbifier*. Mininode also only supports the ECMAScript 5 version of JavaScript (which dates back to 2009), whereas *Stubbifier* can debloat JavaScript ES2019 applications that use modern JS features such as modules, classes, async/await, etc.

We tried running Mininode on all 15 subject applications that we used to evaluate *Stubbifier*. Of these, 4 used features specific to JavaScript ES6+,[13] causing Mininode to fail on parsing the source code; 2 of them crashed Mininode with a runtime exception[14] because Mininode dispatches a malformed call to `fs.stat`, and in 1 of them[15] Mininode removed one a production file `factory.js` file, rendering the debloated application non-functional. In the remaining 8 projects where Mininode ran successfully, we noted that the only files that it removed were development dependency module files, test files, and non-code files such as `.eslintignore` and `LICENSE`.

Fundamentally, Mininode and *Stubbifier* have different objectives and apply different techniques. Mininode completely removes code and other files such as license files. On the other hand, *Stubbifier* replaces code that is likely to be unused with stubs. Note that it is not possible to apply *Stubbifier* after applying Mininode because Mininode removes the application's tests, which *Stubbifier* needs for call graph construction. Conversely, *Stubbifier* creates production distributions that already exclude development dependences, so applying Mininode after applying *Stubbifier* does not make sense.

5 Threats to Validity

Our approach relies on 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

[13] `memfs,fs-nextra,commander.js,redux`

[14] `memory-fs,serve-favicon`

[15] `prop-types`

with a low-quality test suite may generate a call graph that does not represent a comprehensive usage of the application functions, thus leading to more stubs and likely more stub expansion. To mitigate against bias, we did not consider the *quality* of an application's tests when selecting projects for our evaluation, only that the application had tests at all (and that these tests passed). 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. We also see a large variation in the *coverage* achieved by these test suites over the code available to be stubbed (i.e., the source code and production dependencies of an application): we see coverage as high as 99.04% and as low as 0.52%. This suggests that the quality of the test suites of the projects in our evaluation varies considerably.

Also, we are cognizant that we are drawing generalized conclusions based on a limited set of JavaScript projects. To mitigate potential bias in project selection, we selected 15 projects in a systematic manner from the most popular projects published by `npm`: from a list of projects sorted in descending order by number of weekly downloads, we attempted to install, build, and run project test suites. If a project satisfied all these criteria, we then randomly selected from its clients and attempted to install, build, and run their tests; if the project had five such clients, it was selected. We also note that the subject applications vary considerably in size, in number of dependencies, as well as application domains: e.g., `memfs` is an in-memory file system, `body-parser` is a parser for request bodies, and `css-loader` is a custom loader for `css` files. In a similar vein, we are cognizant of the fact that the five chosen client applications might not be representative clients of the projects. To mitigate potential bias here, we chose the clients randomly, and we chose five of them to try and get a variety of use cases. We note that there is a range in the amount of code loaded dynamically across the clients, so we see that not all the clients use the same features of a package. We do also note that there is often overlap in the stubs expanded across clients: this is unsurprising, as we expect some overlap in the ways clients use a project, and it indicates untested functionality in the project.

It is also possible that the reported runtimes are subject to measurement bias. We mitigate this by running all performance experiments on a machine with no other processes running. We also report the average run time over 10 runs, after discarding two initial runs, which minimizes risk of long experiment startup time.

In our experiments with the Rollup bundler, we had to manually configure *Stubbifier* to avoid stubbing four functions in the bundles for `commander` and `q` that were introduced by the bundler. Since these functions did not occur in the call graphs created by *Stubbifier*, they would otherwise have been replaced with stubs, resulting in size reductions in excess of 95%. However, such a size reduction would have been counterproductive—these functions are always executed when the bundles are used, and thus the introduced stubs would always have to be expanded. There is a potential for human error here, but identifying these four functions was not difficult: for `commander`, the bundler wrapped the entire module in an immediately invoked function expression (IIFE), and in the case of `q` the bundler included large swaths of code in the exported object of the bundle. Longer term, an automated solution to this problem could be devised.

6 Related Work

Our work was inspired by DOLOTO (Livshits and Kiciman, 2008), a tool that applies code-splitting to an application based on "access profiles" obtained from users interacting with an instrumented version of the 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 when a stubbed function is invoked.

There are several factors that make *Stubbifier* more practical than Doloto. Most importantly, *Stubbifier* is fully automatic, debloating an application based on call graphs that were constructed from its tests. *Stubbifier* handles modern JavaScript ECMA International (2019), which includes many features (e.g., classes, promises, async/await, generators, modules etc.) that were not present when Doloto was developed in 2008. Moreover, *Stubbifier* supports not only the function-level stubs that were used by Doloto, but also file-level stubs to handle the common case where all functions in a file are found to be unreachable. *Stubbifier* also provides a guarded execution mode, which prevents injection vulnerabilities resulting from calls to functions such as `eval` and `exec` when they are invoked from within untested code that resulted from expanding stubs. Lastly, *Stubbifier* has been developed to be used in conjunction with bundlers.

In Section 4.3, we compared *Stubbifier* with Mininode (Koishybayev and Kapravelos, 2020), another tool for debloating Node.js applications and noted several significant differences between the two debloating techniques: (1) Mininode targets development distributions, which include application tests as well as dependencies only needed during development (e.g., test suite runners like `jest`), whereas *Stubbifier* targets production distributions, which already exclude tests and development dependencies; (2) Mininode completely removes code, and can introduce application crashes if the removed code is called, whereas *Stubbifier* replaces code with stubs that can fetch the original code as needed; (3) Mininode targets the ECMAScript 5 version of JavaScript, which lacks many widely used features such as classes, async/await, and modules, and these are all supported by *Stubbifier* which supports the ECMAScript 2019 version of JavaScript. We ran Mininode on the 15 subject applications in the evaluation, and found that Mininode successfully debloated only 8 of them, and in those it only removed development dependencies, test files, and non-code files. There is also recent work on debloating other languages: JShrink (Bruce et al., 2020) is a tool for debloating the bytecode of Java applications. Their technique makes use of a combination of both static and dynamic analyses, to use both the strong type guarantees of the Java language, and to also deal with dynamic language features that are becoming more prevalent in modern Java use.

Building minimal application bundles is both well-studied and prevalent in industry. Several implementations of Smalltalk developed in the 1990s (e.g., (Vis, 1995, 1997)) include features for "packaging" or "delivering" applications, and IBM's 1997 Handbook for VisualAge for Smalltalk (Vis, 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.

6.1 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.

6.2 Vulnerability Detection and Reduction

Guarded execution mode's ability to intercept dangerous function execution in dynamically loaded code is intended to reduce the attack surface of applications, and ultimately make them less vulnerable to attacks.

There is a wealth of existing work in this area. On the topic of traditional injection vulnerabilities, Gauthier et al. (2018) describe an approach for detecting injection vulnerabilities through a mix of white-box analysis (of application code), and black-box analysis (of third party modules), and Nielsen et al. (2019) present a static dataflow analysis tool which overcomes scalability issues by analyzing a limited amount of third-party modules. Taint analysis is a popular method for detecting these types of vulnerabilities, and

Staicu et al. (2020) describe an approach to automatically extracting taint specifications for JavaScript libraries, which is important as taint analysis require taint specifications to report taint flows, and manually coming up with taint specifications is tedious at best, and error-prone at worst. Injection vulnerabilities are not alone in plaguing JavaScript code, and Li et al. (2022) present a novel data structure constructed from various static analysis, model a variety of vulnerabilities (e.g., beyond injection), and use abstract interpretation to detect them. Node.js allows JavaScript programs to execute arbitrary shell commands, and Vasilakis et al. (2021) detail an approach specifying read-write-execute permissions for third-party libraries, noting that much third-party code executes with more elevated permission than is required. Further, Staicu et al. (2018) report on a study of over 200k Node.js applications, arguing that command-line injection vulnerabilities are common, and present a system that synthesizes grammar-based policies from template values (that are generated as abstractions of values likely to result in vulnerabilities). Another interesting vulnerability specific to languages with prototype-based inheritance (like JavaScript) is reported on by Li et al. (2021); known as *prototype pollution*, base object prototypes are modified to introduce new attack vectors.

The `npm` ecosystem does provide a "security audit" of packages it installs, and typically reports that dozens of vulnerabilities exist in installed dependencies. Zimmermann et al. (2019) conduct a study of security threats in the `npm` ecosystem, and determine that a lack of maintenance contribute to many present vulnerabilities. Updating packages is important to keep up with security patches, and semantic versioning helps developers determine the work involved in downloading a new version of a package; Møller and Torp (2019) argue that semantic versioning is poorly used in JavaScript, and propose a technique to detect breaking changes in security patches using fuzz testing of API models.

7 Conclusions and Future Work

JavaScript is an increasingly popular language for server-side development, thanks in part to the Node.js runtime environment and the vast ecosystem of modules available on `npm`. Unfortunately, `npm` installs modules with *all* of their functionality, even if only a fraction is needed, which causes an undue increase in code size. In this paper, we presented a fully automatic technique that identifies dead code by constructing static or dynamic call graphs from the application's tests, and replaces code deemed unreachable with either file- or function-level stubs that can fetch and execute the original code dynamically. The technique also gives users the option to guard their applications against injection vulnerabilities in untested code that result from stub expansion. This technique is implemented in a tool called *Stubbifier*, which supports the ECMAScript 2019 standard.

In an empirical evaluation on 15 Node.js applications and 75 clients of these applications, *Stubbifier* reduced application size by 56% on average while incurring only minor performance overhead. The evaluation also showed that *Stubbifier*'s guarded execution mode is capable of preventing several known injection vulnerabilities that are manifested in stubbed-out code. Finally, *Stubbifier* works alongside bundlers, and for the subject applications under consideration, we measured an average size reduction of 37% in distributions produced by bundlers.

Future work includes the application of similar debloating techniques to other programming languages. A key enabling factor for our technique is the availability of a mechanism for executing arbitrary code at run time, similar to JavaScript's `eval` feature. While such mechanisms tend to create significant challenges for sound static analysis, they enable the

implementation of stubs that load missing code at run time. The use of a fast program analysis techniques that generate an unsound call graph is generally also well suited for dynamic languages.

One concrete avenue for exploration would be adaptation of our technique to Python Lutz (2013), where stubbing can be implemented using the `exec` function [16]. Similar functionality also exists in the R programming language, using its `eval` construct (rlang, 2022).

Declarations

Funding and/or Conflicts of Interests/Competing Interests

The authors declared that they have no conflict of interest.

References

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

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

(2021) bdistin/fs-nextra. https://github.com/bdistin/fs-nextra, accessed: 2021-10-25

(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) expressjs/body-parser. https://github.com/expressjs/body-parser, accessed: 2021-10-25

(2021a) expressjs/compression. https://github.com/expressjs/compression, accessed: 2021-10-25

(2021) expressjs/morgan. https://github.com/expressjs/morgan, accessed: 2021-10-25

(2021a) expressjs/serve-favicon. https://github.com/expressjs/serve-favicon, accessed: 2021-10-25

(2021b) expressjs/serve-static. https://github.com/expressjs/serve-static, accessed: 2021-10-25

(2021) facebook/prop-types. https://github.com/facebook/prop-types, accessed: 2021-10-25

[16] See https://docs.python.org/3/library/functions.html#exec

(2021) isaacs/node-glob. https://github.com/isaacs/node-glob, accessed: 2021-10-25
(2021) kriskowal/q. https://github.com/kriskowal/q, accessed: 2021-10-25
(2021) mapbox/node-blend. https://github.com/mapbox/node-blend, accessed: 2021-04-16
(2021) pillarjs/send. https://github.com/pillarjs/send, accessed: 2021-10-25
(2021) reduxjs/redux. https://github.com/reduxjs/redux, accessed: 2021-10-25
(2021a) streamich/memfs. https://github.com/streamich/memfs, accessed: 2021-10-25
(2021b) tj/commander.js. https://github.com/tj/commander.js, accessed: 2021-10-25
(2021) webpack-contrib/css-loader. https://github.com/webpack-contrib/css-loader, accessed: 2021-10-25
(2021b) webpack/memory-fs. https://github.com/webpack/memory-fs, accessed: 2021-10-25
Abadi M, Budiu M, Erlingsson U, Ligatti J (2009) Control-flow integrity principles, implementations, and applications. ACM Trans Inf Syst Secur 13(1), DOI 10.1145/1609956.1609960, URL https://doi.org/10.1145/1609956.1609960
Agesen O, Ungar D (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, pp 355–370, *ACM SIGPLAN Notices* 29(10).
Agesen O, Palsberg J, Schwartzbach MI (1993) Type inference of SELF. In: ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings, pp 247–267, DOI 10.1007/3-540-47910-4_14
Andreasen E, Møller A (2014) Determinacy in static analysis for jQuery. In: Proc. 29th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA)
Avgustinov P, de Moor O, Jones MP, Schäfer M (2016) QL: Object-oriented queries on relational data. In: 30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy, pp 2:1–2:25, DOI 10.4230/LIPIcs.ECOOP.2016.2
Bacon DF, Sweeney PF (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., pp 324–341, DOI 10.1145/236337.236371
Bhattacharya S, Gopinath K, Nanda MG (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 and Applications, ACM, New York, NY, USA, OOPSLA '13, pp 745–764, DOI 10.1145/2509136.2509522
Bruce BR, Zhang T, Arora J, Xu GH, Kim M (2020) JShrink: In-depth investigation into debloating modern Java applications. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 135–146
De Sutter B, De Bus B, De Bosschere K (2002) Sifting out the mud: Low level C++ code reuse. SIGPLAN Notices 37(11):275–291, DOI 10.1145/583854.582445
Dean J, Grove D, Chambers C (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, pp 77–101, DOI 10.1007/3-540-49538-X_5

ECMA International (2019) ECMAScript 2019 language specification. https://262.ecma-international.org/10.0/, accessed: 2021-04-16

ECMA International (2021) ECMAScript module system. https://www.ecma-international.org/ecma-262/#sec-modules, accessed: 2021-04-16

Gauthier F, Hassanshahi B, Jordan A (2018) A¡span class=”smallcaps smallercapital”¿ffogato¡/span¿: Runtime detection of injection attacks for node.js. In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, Association for Computing Machinery, New York, NY, USA, ISSTA '18, p 94–99, DOI 10.1145/3236454.3236502, URL https://doi.org/10.1145/3236454.3236502

GitHub (2020) Language trends on GitHub. https://octoverse.github.com/#top-languages

GitHub (2021) CodeQL. https://github.com/github/codeql, accessed: 2021-04-16

Hovemeyer D, Pugh W (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, pp 127–140

Istanbul (2021) nyc. https://www.npmjs.com/package/nyc, accessed: 2021-10-12

Jensen SH, Madsen M, Møller A (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, pp 59–69

Jensen SH, Jonsson PA, Møller A (2012) Remedying the eval that men do. In: Heimdahl MPE, Su Z (eds) International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012, ACM, pp 34–44, DOI 10.1145/2338965.2336758, URL https://doi.org/10.1145/2338965.2336758

Karim R, Tip F, Sochůrková A, Sen K (2018) Platform-independent dynamic taint analysis for JavaScript. IEEE Transactions on Software Engineering 46(12):1364–1379

Koishybayev I, Kapravelos A (2020) Mininode: Reducing the Attack Surface of Node.js Applications. In: Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)

Koo H, Ghavamnia S, Polychronakis M (2019) Configuration-driven software debloating. In: Proceedings of 12th European Workshop on Systems Security (EuroSec '19)

Krintz C, Calder B, Hölzle U (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., pp 276–291, DOI 10.1145/320384.320412

Li S, Kang M, Hou J, Cao Y (2021) Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis, Association for Computing Machinery, New York, NY, USA, p 268–279. URL https://doi.org/10.1145/3468264.3468542

Li S, Kang M, Hou J, Cao Y (2022) Mining node.js vulnerabilities via object dependence graph and query. In: 31st USENIX Security Symposium (USENIX Security 22), USENIX Association, Boston, MA, URL https://www.usenix.org/conference/usenixsecurity22/presentation/li-song

Li Y, Tan T, Møller A, Smaragdakis Y (2018a) Precision-guided context sensitivity for pointer analysis. PACMPL 2(OOPSLA):141:1–141:29

Li Y, Tan T, Møller A, Smaragdakis Y (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, pp 129–140

Livshits VB, Kiciman E (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, pp 350–360, DOI 10.1145/1453101.1453151

Lutz M (2013) Learning Python 5th Edition

Madsen M, Tip F, Lhoták O (2015) Static analysis of event-driven Node.js JavaScript applications. In: Aldrich J, Eugster P (eds) Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015, ACM, pp 505–519, DOI 10.1145/2814270.2814272, URL https://doi.org/10.1145/2814270.2814272

MDN (2021) Tree shaking. https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking, accessed: 2021-10-11

Møller A, Torp MT (2019) Model-based testing of breaking changes in node.js libraries. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2019, p 409–419, DOI 10.1145/3338906.3338940, URL https://doi.org/10.1145/3338906.3338940

Mozilla (2021) Rest parameters. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters, accessed 2021-04-16

Nielsen BB, Hassanshahi B, Gauthier F (2019) Nodest: Feedback-driven static analysis of node.js applications. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2019, p 455–465, DOI 10.1145/3338906.3338933, URL https://doi.org/10.1145/3338906.3338933

Niu B, Tan G (2015) Per-input control-flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, CCS '15, p 914–926, DOI 10.1145/2810103.2813644, URL 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

Rayside D, Kontogiannis K (2002) Extracting Java library subsets for deployment on embedded systems. Sci Comput Program 45(2):245–270, DOI 10.1016/S0167-6423(02)00059-X

Richards G, Hammer C, Burg B, Vitek J (2011) The eval that men do—A large-scale study of the use of eval in JavaScript applications. In: Mezini M (ed) ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings, Springer, Lecture Notes in Computer Science, vol 6813, pp 52–78, DOI 10.1007/978-3-642-22655-7_4, URL https://doi.org/10.1007/978-3-642-22655-7_4

rlang (2022) Execute a function. See https://rlang.r-lib.org/reference/exec.html

Rollup (2021) Rollup. https://www.npmjs.com/package/rollup, accessed: 2021-10-11

Sharif H, Abubakar M, Gehani A, Zaffar F (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,

pp 329–339, DOI 10.1145/3238147.3238160

Sridharan M, Dolby J, Chandra S, Schäfer M, Tip F (2012) Correlation tracking for points-to analysis of JavaScript. In: Noble J (ed) ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings, Springer, Lecture Notes in Computer Science, vol 7313, pp 435–458, DOI 10.1007/978-3-642-31057-7_20, URL https://doi.org/10.1007/978-3-642-31057-7_20

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

Staicu CA, Pradel M, Livshits B (2018) Synode: Understanding and automatically preventing injection attacks on node.js. In: NDSS

Staicu CA, Torp MT, Schäfer M, Møller A, Pradel M (2020) Extracting taint specifications for javascript libraries. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '20, p 198–209, DOI 10.1145/3377811.3380390, URL https://doi.org/10.1145/3377811.3380390

Stein B, Nielsen BB, Chang BE, Møller A (2019) Static analysis with demand-driven value refinement. Proc ACM Program Lang 3(OOPSLA):140:1–140:29, DOI 10.1145/3360566, URL https://doi.org/10.1145/3360566

Sweeney PF, Tip F (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, pp 98–107

Tip F, Palsberg J (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., pp 281–293, DOI 10.1145/353171.353190

Tip F, Laffra C, Sweeney PF, Streeter D (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., pp 292–305, DOI 10.1145/320384.320414

Tip F, Sweeney PF, Laffra C, Eisma A, Streeter D (2002) Practical extraction techniques for Java. ACM Trans Program Lang Syst 24(6):625–666, DOI 10.1145/586088.586090

Turcotte A, Arteca E, Mishra A, Alimadadi S, Tip F (2021) Stubbifer: Debloating dynamic server-side JavaScript applications (artifact). https://doi.org/10.5281/zenodo.5599914

Vasilakis N, Staicu CA, Ntousakis G, Kallas K, Karel B, DeHon A, Pradel M (2021) Preventing dynamic library compromise on node.js via rwx-based privilege reduction. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, CCS '21, p 1821–1838, DOI 10.1145/3460120.3484535, URL https://doi.org/10.1145/3460120.3484535

Wagner G, Gal A, Franz M (2011) "Slimming" a Java virtual machine by way of cold code removal and optimistic partial program loading. Sci Comput Program 76(11):1037–1053, DOI 10.1016/j.scico.2010.04.008

webpack (2021) webpack. https://www.npmjs.com/package/webpack accessed: 2021-10-11

webpack-contrib (2021) css-loader. https://www.npmjs.com/package/css-loader, accessed 2021-04-16

Zhang C, Wei T, Chen Z, Duan L, Szekeres L, McCamant S, Song D, Zou W (2013) Practical control flow integrity and randomization for binary executables. In: 2013 IEEE Symposium on Security and Privacy, pp 559–573

Zimmermann M, Staicu CA, Tenny C, Pradel M (2019) Smallworld with high risks: A study of security threats in the npm ecosystem. In: Proceedings of the 28th USENIX Conference on Security Symposium, USENIX Association, USA, SEC'19, p 995–1010