Mininode: Reducing the Attack Surface of Node.js Applications

Igibek Koishybayev
North Carolina State University
ikoishy@ncsu.edu

Alexandros Kapravelos
North Carolina State University
akaprav@ncsu.edu

Abstract

JavaScript has gained traction as a programming language that qualifies for both the client-side and the server-side logic of applications. A new ecosystem of server-side code written in JavaScript has been enabled by Node.js, the use of the V8 JavaScript engine and a collection of modules that provide various core functionality. Node.js comes with its package manager, called `NPM`, to handle the dependencies of modern applications, which allow developers to build Node.js applications with hundreds of dependencies on other modules.

In this paper, we present Mininode, a static analysis tool for Node.js applications that measures and removes unused code and dependencies. Our tool can be integrated into the building pipeline of Node.js applications to produce applications with significantly reduced attack surface. We analyzed 672k Node.js applications and reported the current state of code bloating in the server-side JavaScript ecosystem. We leverage a vulnerability database to identify 1,660 vulnerable packages that are loaded from 119,433 applications as dependencies. Mininode is capable of removing 2,861 of these vulnerable dependencies. The complex expressiveness and the dynamic nature of the JavaScript language does not always allow us to statically resolve the dependencies and usage of modules. To evaluate the correctness of our reduction, we run Mininode against 37k Node.js applications that have unit tests and reduce correctly 95.4% of packages. Mininode was able to restrict access to the built-in `fs` and `net` modules in 79.4% and 96.2% of the reduced applications respectively.

1 Introduction

Node.js [10] is an open-source JavaScript runtime engine typically used to build scalable network applications. The JavaScript runtime that powers Node.js is based on Chrome's V8 engine. Despite Node.js' young age, it has become very popular among the open-source community and enterprises. Moreover, big companies such as Microsoft, IBM, PayPal [22, 27, 39] are among others who use Node.js in their products. One of the reasons for its popularity is in Node.js architecture choice. Node.js uses a non-blocking event-based architecture which gives an ability to developers to scale up Node.js applications easily. Nowadays Node.js is used to develop critical systems [49] that require security attention.

Node.js developers distribute community-developed libraries using an in-house built package manager system called `NPM`. `NPM` is considered to be the largest package manager by the number of packages [12] it hosts (over million) and growth rate of almost 800 pkg/day [9]. Since 2014, the `NPM` registry traffic has grown 23,500%, which shows its increasing popularity among developers [47]. This staggering amount of packages hosted in `NPM` gives developers the power to build apps very quickly by using already implemented functionality by others. In this paper, we argue that overusing third-party libraries comes with its own security risks.

The drawbacks of extensive dependence on third-party packages are: (1) developers need to trust others on the security and maintenance of the libraries; (2) the popularity of `NPM` makes it lucrative for adversarial users to distribute malicious libraries using attacks such as typosquatting [20, 43, 44], ownership takedown and introducing a backdoor [45, 52]; (3) upgrade or removal of the package from `NPM` may break the build pipeline of an application [46].

Our study of 1,055,131 packages shows that on average only 6.8% of the code in the application is original code according to source logical lines of code (LLOC) or putting in different words 93.2% of the code in Node.js application is developed by third-parties. One of the reasons why developers tend to use "trivial" third-party packages, is the belief that they are well managed and tested. Despite the belief, the study shows that only 45.2% of "trivial" packages have tests implemented [19].

Previous works on Node.js security mostly concentrate on architecture choice of Node.js and, therefore, on attacks that target the main thread of Node.js applications [23–25, 38, 42]. Others have conducted research on the reasons why developers use "trivial" dependencies [19] and security implications of depending on `NPM` packages [52]; however, no research

on an attack surface that extensive usage of third-party libraries may bring and ways of reducing the attack surface of a Node.js application was conducted before.

Due to all of the above, it is important to know the attack surface of third-party packages and to reduce them ahead of time during a development process.

Our main contributions are the following:

- We developed a system that reduces the attack surface of Node.js applications by removing unused code parts and modules from the dependency graph. The system can use one of two different reduction modes: (1) *coarse-grain*; (2) *fine-grain*; Our system is publicly available here: https://kapravelos.com/projects/mininode
- We analyzed 672,242 Node.js applications from the NPM repository and measured their attack surface. Our findings show that at least 119,433 (11.3%) of applications depend on vulnerable third-party packages. Also, on average only 9.5% of all LLOC is used inside analyzed packages.
- We created a custom build of Node.js that can restrict the access to the built-in modules using a whitelist generated by Mininode. Our evaluation experiment shows that Mininode successfully restricted the access to `fs` and `net` modules for 79.4% and 96.2% of packages, respectively.

2 Background

In this section, we describe the technical details of Node.js modules, how NPM works, and how Node.js resolves imported modules, both built-in and third-party dependencies. We use the term **module** to describe anything under the *node_modules* folder that can be loaded using the `require` function. A **package** is everything that is hosted on NPM, but not necessarily can be loaded by using `require` function, *e.g.* CSS files. In this paper, we refer to a package as a directory of files with a JavaScript entry point that can be loaded with the `require` function. We treat applications the same as packages, *i.e.* they both have a JavaScript entry point, from which Mininode can start its analysis.

2.1 Node.js module system

JavaScript has been traditionally the language of the browser. Web applications build their front-end logic in JavaScript by leveraging browser APIs. Node.js applications rely on a completely different ecosystem that is built to assess the needs of server-side applications. Instead of the DOM and other Web Platform APIs, Node.js relies on built-in modules that provide functionalities like networking and filesystem access. These modules are based on the CommonJS module system and only recently we have seen experimental support for ECMAScript modules [11].

Node.js treats every JavaScript file as a CommonJS module. Node.js has built-in `require()` function to import both built-in and developer-created modules into code. The `require` function behaves differently depending on the type of the module requested. If the requested module is on the list of built-in modules, then it is returned directly from the modules written in C++. If the requested module is not part of the built-in modules, `require` will wrap the imported module with a function wrapper, as shown in Listing 1, before executing the code. This ensures that variables from the imported modules are not placed unintentionally in the global scope. Despite this, modules can declare variables and functions in the global scope, which poses a challenge in accurately determining the used APIs of the module (§5.2).

```
1 (function(exports, require, module,
      __filename, __dirname) {
2 // Module code lives in here
3 });
```

Listing 1: Function wrapper to execute module code

Every module that wants to provide some of its functionality to other modules can use the `exports` object. For example, in Listing 2 *b.js* exports two functions. However, *a.js* uses only function `foo()` after importing *b.js*. Thus, function `bar()` can be removed without impacting the behavior of the *a.js*. We discuss how we leverage this mechanism to restrict access to built-in modules Section 6.1.

```
1 //inside b.js
2 exports.foo = function foo(){}
3 function bar(){}
4 exports.bar = bar;
5 //inside a.js
6 var b = require("./b.js")
7 b.foo()
```

Listing 2: Example of CommonJS module and common ways to export the functionality

2.2 Node Package Manager (NPM)

Node.js comes with a built-in package manager called NPM, which hosts aside from JavaScript libraries also front-end CSS, JavaScript frameworks and command-line tools. In this paper we focus only on server-side Node.js packages that are distributed over NPM. Developers can install a package using the command "`npm install <name>@<version>`", where "`<name>`" is the name of the package. By default, if version is missing, NPM will install the latest version of the package. If the package name is not given, NPM will look for the *package.json* file inside the current working directory and will install all packages listed as dependencies in the file. The *package.json* file also contains metadata about the Node.js application. These metadata can contain the main file of the application (*i.e.* entry point), the version number, a short de-

scription, a list of dependencies, and other information about the Node.js application. `NPM` installs dependencies transitively. For example: if package *A* depends on package *B*, and package *B* depends on package *C*, `NPM` will install packages *A*, *B*, *C*. The `NPM`'s transitive installation of dependencies creates a serious problem of bloated code, as it makes it really hard for the developer to understand on how many packages the code depends. De Groef *et al*. states that some popular packages may in total depend on more than 200 packages [26].

3 Threat Model

Our threat model targets vulnerable Node.js applications that are susceptible to arbitrary code execution vulnerabilities. The main premise of this paper is that we can 1) reduce the capabilities of the attack by restricting the application's functionality and 2) eliminate further exploitation of the application that would elevate the attacker's capabilities by targeting vulnerabilities in unused dependencies.

We reduce the attack surface of applications by restricting the available built-in modules that can be loaded to the absolute necessary. This results in removing classes of capabilities from the application if they are not already used, like filesystem access or networking.

We mitigate chained exploitation by removing unused vulnerable modules and restricting built-in modules. Our assumption here is that the application is exploitable, but with certain restrictions, i.e., code can be injected, but without arbitrary execution due to unsafe regular expression checks. In that particular scenario, the attacker can take advantage of other existing vulnerable packages that are now reachable and gain full control of the application. The attacker can take advantage of the existing modules in two scenarios: 1) directly manipulating the input to the `require` function to load any modules, 2) indirectly manipulate the input to the `require` function to load any modules, except built-ins. Our system is capable of stopping further exploitation of the unused parts of the application, but it does not prevent the initial vulnerability that leads to partial code execution.

When the attacker can directly manipulate the `require` function and load additional built-in modules, Mininode restricts the access to unused built-in modules, even if they are used in one of the unused transitive dependencies. This significantly reduces the capabilities of the attack.

Listing 3 shows a theoretical motivating example of chained exploitation when the attacker can indirectly control the input to the `require` function. In the example, the attacker can inject malicious data to the `fs.linkSync` function (line 6), which is used to create a symbolic link, by manipulating the request data. For example, the attacker can replace the entry point of *header-parser* with a symbolic link to *unused.js* by manipulating `dst` and `src` fields. Therefore, next time when the attacker navigates to `"/exploit"` endpoint, Node.js will load the *unused.js* module instead of the *header-parser* package, and the application passes data provided by the attacker to the *unused.js* (lines 11-12). Note that an attacker cannot manipulate symbolic links to load built-in modules because they are part of the Node.js binary. For this kind of chained exploitation, Mininode can remove unused packages from the application and restrict the attacker's ability to load modules that are not used by the application; thus, making the attack less effective. Some vulnerable packages though, like *fast-http* [4] , *marscode* [2] and *marked-tree* [3] are directly exploitable by just loading their module, but not all vulnerabilities can be exploited via chained exploitation, as they can depend on additional constraints that might not be available to the attacker.

```
1  const fs = require('fs')
2  const express = require('express')
3  const app = express()
4
5  // some code parts omitted for brevity
6  app.get('/vulnerable', (req, res) => {
7    fs.linkSync(req.body.dest, req.body.src);
8    res.send('Hello World!')
9  });
10
11 app.get('/exploit', (req, res) => {
12   let parser = require('header-parser');
13   let result = parser(req.headers);
14   res.send(result);
15 });
```

Listing 3: Motivating vulnerable example of chained exploitation for loading unused packages

One of the great advantages of Mininode is that it restricts the attacker from using *any* unused module, including built-in modules, *e.g.* `fs`, even if it is used in a transitive dependency. When these modules are not used from the application, Mininode can have a significant impact on the attack.

4 Design Goals and Architecture Overview

4.1 Design Goals

There were two main design goals that we followed during the implementation of the Mininode.

Effectiveness. Mininode should reduce the attack surface of the Node.js application as much as possible. To achieve the *effectiveness* goal, we implemented two modes of reduction: (1) coarse-grain; (2) fine-grain; and added a built-in module restriction mechanism into Node.js. We provide Mininode's reduction effectiveness results later in the paper (§7.1 and 7.2).

Correctness. Mininode must remove only unused code parts, *i.e.* should not break the original behavior of the application. To validate that Mininode meets the *correctness* goal, we automatically verified the original behavior of 37,242 packages after reduction (§7.1).

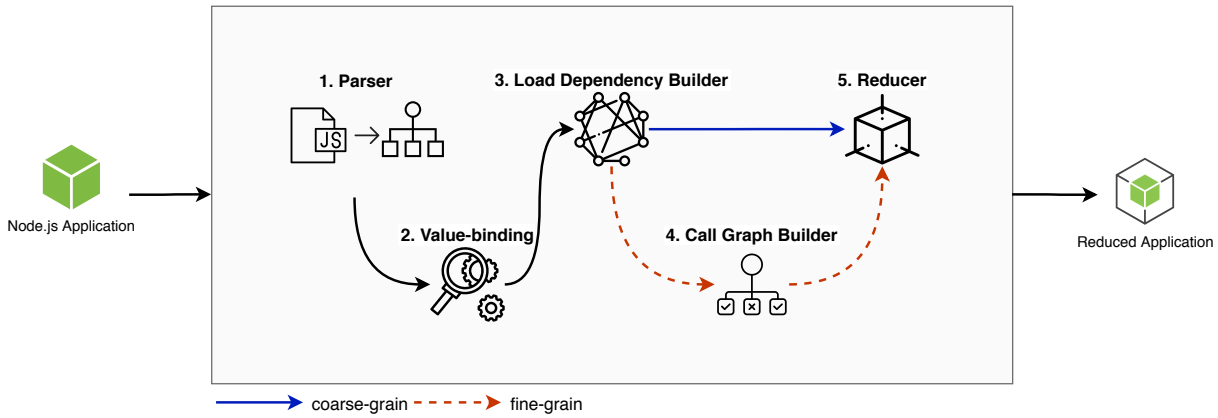

Figure 1: Mininode consists of main 5 parts: ❶ Parser; ❷ Value-binding; ❸ Load Dependency Builder; ❹ Call Graph Builder; ❺ Reducer and supports *coarse-grain* and *fine-grain* reductions

4.2 Architecture Overview

Mininode consists of five different stages as shown in Figure 1, that will run during the reduction of the application.

Mininode takes a directory path of a Node.js application, which contains *package.json* file, as an input. Then, Mininode traverses the directory, parses all JavaScript files and generates abstract syntax trees (ASTs) for each JavaScript file (❶). We use AST representation of the source code because it is easier to analyze statically and convenient to transform the tree structure to modify the initial source code. To parse JavaScript code, we use the popular open-source *esprima* [8] library. The final outcome of the *Parser* is an array of all modules inside the application folder and each module's AST representation. After *Parser* completes processing the code, it will pass the generated ASTs to the next stages of Mininode.

The *Value-binding* (❷) is a pre-reduction stage that collects metadata about each module in the application from their AST representation, which is used in reduction stages. Also, *Value-binding* collects overall statistics on each module regarding the number of logical lines of code, dynamic imports, dynamic export.

The main reduction process consists of three different stages: (1) *Load Dependency Builder*, (2) *Call Graph Builder* and (3) *Reducer*.

The *Load Dependency Builder* (❸) builds a file-level dependency graph of the application by traversing the AST generated by *Parser*. To build the dependency graph, *Load Dependency Builder* starts from the entry point(s) and detects all `require` function calls from AST. All modules that are recursively accessible from the application's entry point(s) are marked as used by *Load Dependency Builder*. Despite the simplicity of the algorithm, there are challenges that need to be addressed to construct a complete dependency graph (§5) and are further discussed in Section 6.

Mininode supports two reduction modes for the application: (1) *coarse-grain* reduction; (2) *fine-grain* reduction. The coarse-grain reduction mode works at file-level and removes unused files in the application, while the fine-grain reduction works at function-level and removes functionalities from individual modules that are never used. As shown in Figure 1, Mininode skips the *Call Graph Builder* stage, and proceed directly to the *Reducer* stage in coarse-grain reduction mode.

The *Call Graph Builder* (❹) is responsible for detecting used and unused functions, exports and variables of all modules that are part of the dependency graph generated by *Load Dependency Builder* (§6.4). To achieve effectiveness design goal, *Call Graph Builder* may perform several passes on the AST of each module until no change to the final usage graph is made.

The final stage in our reduction pipeline is the *Reducer* (❺) stage. The *Reducer* is responsible for removing AST nodes and modules that are marked as unused by *Call Graph Builder* and *Load Dependency Builder*, respectively (§6.5). After finishing all of the reduction stages, Mininode generates source code for each module from their updated AST.

5 Challenges

The dynamic nature of JavaScript introduces several challenges to static analysis [33, 37, 50] and this is also true for the module system used by Node.js (§2.1). In this section, we list some of the research challenges that we faced during the implementation of the attack surface reduction tools using static analysis for Node.js applications. Overall, we divide the challenges into two categories: (1) export-related challenges; (2) import-related challenges.

5.1 Export-Related Challenges

The export-related challenges relate to the way how a module is exporting its functionality, and thus directly affects the Mininode's *effectiveness* design goal as defined earlier (§ 4.1).

Failure to deal with export-related challenges will mostly lead to the *under*-reduction of the attack surface, *i.e.* not removing unused functionalities from the module. However, in some rare cases may lead to *over*-reduction of the functionality, *i.e.* removing used functionality. We give an example of both cases in follow up subsections.

Unusual use of the `export` object. JavaScript allows developers to modify an object in several different ways. Consequently, the `export` object can also be modified in several different ways to export functionality. For example, developers can create an alias for the `export` object and use the alias instead to export the module's functionalities.

```
// inside request.js
exports.post = function(){}
exports.get = function(){}
// inside request-v2.js
exports = module.exports = require("request")
exports.patch = function(){}
// inside index.js
var req = require("request-v2")
req.get();
req.patch();
```

Listing 4: Example of re-exporting the imported module

Extension of a module with re-export. In CommonJS module system one can extend other modules by re-exporting it and adding additional functionality. In the example given in Listing 4, one can see how *request-v2.js* module extends *request.js* module with `patch` function. During analysis, Mininode should detect that `get` function used in *index.js* is actually coming from *request.js*. This behavior prevents *over*-reduction of the *request.js* module, and it ensures that Mininode meets the *correctness* goal.

5.2 Import-Related Challenges

The import-related challenges affect the static analysis's performance in the detection of used functionalities of the imported CommonJS modules. Failure to detect used functionalities of imported modules will lead to *over*-reductions, *i.e.* removing used functionalities. Thus, it will lead to breaking the original behavior of the Node.js application. The rest of the section is describing some of the import-related challenges.

```
//inside request.js
exports.post = function(){}
exports.get = function(){}
// inside index.js
request = require("request");
request.post()
// inside util.js
request.get();
```

Listing 5: Example of importing a module in the global scope.

Dynamically importing the module. It is common for Node.js applications to load different modules depending on the execution environment or the user's input. Dynamically importing a module restricts the ability for simple static analysis to detect which module was loaded, which may lead to the removal of the whole used module. Therefore, it is vital to resolve dynamic imports to build a complete load dependency graph of the application.

```
// inside parent.js
module.exports = require("child.js")
exports.foo = function(){
  exports.childFoo(); //defined in child.js
}
exports.parentBar = function() {
}
//inside child.js
exports.childFoo = functon() {
  exports.parentBar(); //defined in parent.js
}
```

Listing 6: Example of invisible parent-child dependency

Importing as a global variable. As discussed in Background section 2.1, Node.js wraps the modules with wrapper function to avoid collision of variable and function names and create separate scope for each module. Despite this, developers can import a module into a global scope, as shown in Listing 5. If a module is imported into a global scope, any other module can have direct access to the module's functionality without importing it. Listing 5 gives an example of loading the *request.js* in global scope in *index.js*, which makes it possible for *util.js* to use `get` function of the *request.js* without importing it.

```
//in index.js entry point
var foo = require("foo")
var bar = require("bar")
foo.x()
bar.z()
//inside foo.js module
var bar = require("bar")
exports.x = function(){}
exports.y = function(){
  bar.w()
}
//inside bar.js module
exports.w = function(){}
exports.z = function(){}
```

Listing 7: Example of cross-reference dependence.

Invisible parent-child dependency. This issue arises when the imported module (child) is using the functionality defined inside the module (parent) that imports the child module as shown in Listing 6. Because of the absence of a clear dependency link from child to parent, this challenge is counter-intuitive in nature. From Listing 6 one can see that, even if *child.js* is not importing *parent.js*, the child module is using `parentBar` that was defined in the parent module. We saw this behavior in one of the most popular `NPM` package `debug` [5].

Cross-reference dependency. The cross-reference dependency problem happens when two different modules import the same module, but they use different parts. For example

in Listing 7 *index.js* and *foo.js* are referencing *bar.js*, however using different parts of it. If Mininode preserves all used functionality, it will preserve `exports.w` function of *bar.js*, because function `exports.w` was used inside *foo.js*. However, function `exports.w` of *bar.js* should be removed because it is not reachable from the entry point (`index.js`) of the application.

6 Implementation

Mininode takes as input a Node.js application and makes three different reduction stages to produce a reduced version of the application (§4.2). This section gives implementation details of each reduction stage and how certain challenges described in Section 5 are resolved. Additionally, the following subsections give details about how access to built-in modules is restricted, and what kind of metadata is collected during *Value-binding* stage.

6.1 Restricting Access to Built-in Modules

The `require` function in Node.js checks if the requested module is in a built-in modules list before trying to resolve it (§2.1). We modified the original behavior of the `require` function at the Node.js C++ level. The patched `require` function restricts access to the built-in modules by checking if the requested module is not in a whitelist of built-in modules generated by Mininode. The whitelist is generated only once during the reduction of the application and kept unchanged. If the application does not have previously generated whitelist, our custom-built Node.js will allow all built-in modules to avoid breaking the application.

6.2 The *Value-binding* Details

The *Value-binding* (❷) is a preprocessing step that collects metadata about each module in order to help other reduction stages to overcome challenges listed previously (§5). *Value-binding* collects an array of *aliases* for `exports` object and `require` function, because developers may rename these CommonJS APIs by assigning them to another variable and using an *alias* for the API instead of using API directly. Especially, this behavior can be seen in the case of packages that provide a minified version. Minification usually substitutes longer names with shorter ones to decrease the size of the file.

Value-binding also collects a dictionary of identifier names with their corresponding values, which are used to detect the possible values of dynamically imported modules. Possible values of identifiers could be literals (strings) or other identifier names that were assigned to the original identifier. If an identifier's value depends on any dynamic expression, *e.g.* a function call, *Value-binding* will mark the identifier as *non-resolvable*. However, if the identifier's value depends on binary expression, *e.g.* `var a=b+"-production"`, *Value-binding* tries to resolve the possible values by getting the values of `"b"` from the dictionary and adding the `"-production"` to it. Note that variable `"b"` in the dictionary must be resolvable. Otherwise, *Value-binding* will mark the variable `"a"` as non-resolvable.

6.3 The *Load Dependency Builder* Details

The *Load Dependency Builder* (❸) is responsible for building the file-level dependency graph by looking for `require` (or its *aliases* (§6.2)) function calls in AST and by resolving the function's argument to one of the existing modules (§4.2). The *Load Dependency Builder* resolves `require`'s argument using Node.js default resolution algorithm if the argument's type is literal. In other cases, it will use a simple algorithm to resolve dynamic import.

Resolving dynamic imports. To resolve the dynamic imports, the *Load Dependency Builder* uses the dictionary generated by *Value-binding* in the previous stage. If the argument's type is an identifier and the dictionary contains values for the identifier, *Load Dependency Builder* iterates through the possible values and resolves to possible modules in the application. This process is one of Mininode's advantages over other open-source `NPM` packages implemented to build the dependency tree of an application [13, 29, 31]. On the other hand, if the identifier does not exist in the dictionary, or the identifier is marked as non-resolvable, *Load Dependency Builder* will mark a module as using complicated dynamic import that can not be resolved reliably using only static technique. If the dependency graph of the application contains a module with complicated dynamic import, Mininode stops performing further analysis and exits because the application under analysis cannot be reduced reliably without breaking its original behavior.

6.4 The *Call Graph Builder* Details

The *Call Graph Builder* (❹) runs only during fine-grain reduction mode, as can be depicted from Figure 1. The goal of the *Call Graph Builder* is to detect which parts of the code are used for each module and mark unused ones. To achieve the goal, it performs two separate tasks on the module's AST each time during analysis.

The first task, which is called *marking unused*, is responsible for marking exports, functions, and variables as unused if they are not used inside or outside of the module according to an array of used exports of the analyzed module.

The second task, which is called *usage detection*, is responsible for constructing the used exports array for each imported modules of the currently analyzed module. It achieves this by recording the variable names initialized by `require` (or the aliases (§6.2)) function calls and detecting all member expressions (*i.e.* property accesses) for all recorded variable names.

Resolving cross-reference challenge. To achieve the *effectiveness* design goal discussed in Section 4.1, Mininode needs to resolve the *cross-reference* challenge. The cross-reference challenge can be solved by always running the *marking unused* task before the *usage detection* task for each module's AST. One benefit of running *marking unused* before *usage detection* is that during *usage detection*, we can skip functions that are marked as unused. For example, for the module *foo.js* from Listing 7, by first running *marking unused* task, we mark `exports.y` as unused. Thus, *usage detection* will skip traversing `exports.y` during analysis, and, therefore, `exports.w` will not be included in an array of used exported functions of the *bar.js* module.

Resolving extension of the module by re-exporting. The *Call Graph Builder* internally keeps track of used re-exported modules during an analysis of the AST. Later on, *Call Graph Builder* will add re-exported modules into the *descendants* array of the *currently* analyzed module. In the case of Listing 4, *request.js* module will be added into the descendants array of *request-v2.js* module. Additionally, *request-v2.js* becomes part of the ancestors array of *request.js* automatically. Later, when *Call Graph Builder* analyzes *request.js* module, it passes all used properties of *request.js*'s ancestors (*i.e. request-v2.js*) to the *marking unused* task as an extra argument. In this particular case, the extra array argument will include `get` and `patch` function names. Thus, exported `get` function of *request.js* will not be marked as unused during the *marking unused* task. Therefore, *Reducer* will not remove used exported functions, and eventually, Mininode will preserve the correctness design goal (§4.1).

Resolving invisible parent-child dependency. *Call Graph Builder* resolves the invisible parent-child dependency challenge almost the same way it resolves extension by re-export challenge. Because Mininode already has information about ancestors and descendants of the module, *Call Graph Builder* can pass as an extra argument all used exports of all ancestors and descendants of the module during the *marking unused* task. In the case of Listing 6, *Call Graph Builder* pass as an extra argument *child.js*'s used exports array, which contains `parentBar` function, into the *marking unused* task for *parent.js*.

Resolving importing as a global variable. To resolve the importing module as a global variable challenge, *Call Graph Builder* keeps track of leaked global variables during the *usage detection* task and stores used members, *i.e.* accessed properties, of a variable inside a dictionary of leaked global variables. In the example from Listing 5, *Call Graph Builder* creates an entry in the dictionary with a key `request` after analyzing *index.js*. The value of the entry is an array of used members, which contains `post` and `get` after analyzing *index.js* and *util.js*, respectively. Next, when *Call Graph Builder* performs the *marking unused* task for *request.js*, it passes the corresponding members' array of the dictionary as an extra argument.

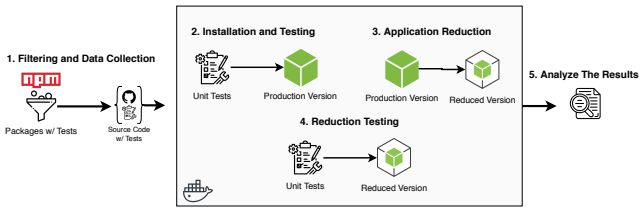

Figure 2: Validation experiment setup

6.5 The *Reducer* Details

The *Reducer* (❺) is responsible for removing the AST nodes marked as unused without breaking the valid syntax of the AST and generating code from the AST. We are using the open-source *escodegen* [7] library to generate the source code from the AST.

Resolving unusual use of `exports` object Currently, *Reducer* can reduce exporting logic for the most common three ways to statically define a property for the object in JavaScript. These are: (1) defining property using dot notation, *e.g.* `exports.a=1`; (2) defining property using array notation, *e.g.* `exports["b"]=2`; and (3) defining property using `Object.defineProperty` function. In addition to the listed ones, *Reducer* tries to resolve the value for dynamically defined properties, *e.g.* `var c='c'; exports[c]=3`, using a similar algorithm as in the *resolving dynamic import challenge*. If the *Reducer* cannot resolve the dynamically defined property, it will not reduce the property, which may cause *under*-reduction. However, this behavior will not break the original code of the application.

7 Mininode Validation and Measurement

Overall, we run two experiments: (1) to validate the correctness of the Mininode in reducing the attack surface of the application; (2) to measure the bloated code and to check the effectiveness of the system in reducing the attack surface and vulnerabilities in the `NPM` registry packages. In the next subsections, we will give more details of experiments' setup and results.

7.1 Mininode Reduction Validation

Experiment Setup. We performed the validation experiment to evaluate the *effectiveness* and *correctness* of the Mininode reduction (§4.1). We measured the effectiveness by calculating the total number of removed files, removed LLOC, and removed exports. The correctness of the reduction, *i.e.* whether Mininode reduced the package without changing its original behavior, is measured by the success rate of passed original unit tests of the packages after the reduction of their attack surface. The validation experiment consists of five steps as shown in Figure 2.

The goal of the first *Filter and Data Collection* step was to gather package names for which we could run unit tests, and calculate the tests coverage metrics automatically. The most popular test coverage package on `NPM` is `nyc`, previously known as `istanbul`, which is advertised as a tool where no configuration is needed to calculate unit tests' code coverage metrics. Therefore, we selected packages that list as one of their dependencies `nyc` and/or unit test package that is compatible with `nyc`. In total, 225,449 out of 1,055,131 packages depend on one of the packages required for automatic testing and coverage calculation. Next we collected packages' source code from Github, installed them, and ran their original unit tests without performing any reduction. We decided to collect source code from Github because not all developers publish package's test code into `NPM`. As a result of this, we were left with only 49,535 packages that were successfully installed, and passed their original unit tests before reduction.

Initially, we tried to reduce the full version of the packages and run the unit tests on the final results. However, this approach failed, because during the reduction step Mininode removed all code responsible for unit tests. To resolve this issue, we leveraged the way Node.js looks for a correct dependency by traversing the *node_modules* directory in the same location where the file requesting the dependency is located. We installed both the full and production versions of the package and created a symbolic link from the main file (*i.e.* entry point) of the full version to the main file of the production version. In this way, we could run the package's unit tests in the full version but test its production version. During the test of packages' production version, we noticed that some packages in production version require developer-only dependencies that are not installed (§2.2). Usually, these cases of counter-intuitive dependencies are used in the packages that are implemented as a plugin to other developer-only packages, *e.g.* *eslint-plugin-jest* is a plugin for *eslint*. Another challenge we faced was that *babel* [1], a popular package to transpile JavaScript, needed a special configuration for projects located in a different folder or symlinked [14]. After eliminating packages that failed during the test of their production version using a symbolic link from the full version, we were left with 45,045 packages in our validation dataset.

The final steps in the validation experiment, before result analysis, were package reduction and unit tests validation of the reduced version of the packages (Figure 2). In 6579 out of 45,045 packages Mininode detected dynamic import that could not be resolved with the current implementation (§6.3) and for 2.7% of packages Mininode threw runtime errors, such as heap out of memory. The final dataset has 37,242 packages that we tested for correctness and effectiveness.

Results. For the final dataset of 37,242 packages, we performed both coarse-grain and fine-grain reduction and ran unit tests to verify that Mininode did not break the original functionality of the reduced packages. The results of both modes of reduction are shown in Table 1. As it may be expected, the coarse-grain reduction (96.0%) has a higher success rate than the fine-grain reduction (95.4%). This is due to the behavior of the fine-grain reduction trying to reduce the individual modules on function-level, compared to coarse-grain reduction, which only performs reduction on file-level. Reducing in the fine-grain mode may cause *over*-reduction of the used functions, which leads to breaking the original behavior of the package and, thus, failing the unit tests. However, despite the higher failure rate, the fine-grain reduction performed better in terms of reducing unused code parts. Fine-grain reduction removed almost 8% more LLOC compared to coarse-grain reduction. Also, in other reduction categories such as reduction of the files and the exported functionalities, fine-grain shows better results. As shown in Table 1 Mininode was able to restrict the access to the built-in modules at least in 28,144 (78.7%) of packages during the coarse-grain reduction, and in 28,196 (79.4%) of packages during the fine-grain reduction. The high results of reduction may be counter-intuitive, especially in case of reducing a lot of files and LLOC from the package. That is why we randomly selected three packages that have a more than 99% reduction rate and manually verified the results. Both of the packages *mfdc-router* and *middleware-chain-js* were shipping a bundled version along with their source code. In these cases, Mininode removed almost all of their dependencies from the node_modules folder and unnecessary source code files. In the last case, after installation, *cpr* had 35,911 test files out of all 35,982 JavaScript files, which were removed by Mininode.

	Coarse-grained	**Fine-grained**
Passed test	35,762	35,531
Removed fs module	28,144	28,196
Removed net module	33,262	34,180
Removed http module	32,878	32,795
Removed https module	33,137	33,044
Total removed files	86.9%	87.3%
Total removed LLOC	85.4%	92.2%
Total removed exports	86.7%	89.0%
Failed test	**1,480**	**1,711**
TOTAL	37,242	37,242

Table 1: Coarse and fine grain reduction results on validation set

Coverage	**Coarse-grained**	**Fine-grained**
100%	13,561	13,548
Between 90-99.9%	8,413	8,290
Between 50-90%	6,915	6,797
Unknown or below 50%	6,873	6,896
Total	**35,762**	**35,531**

Table 2: Coverage statistics of successfully passed test samples

In addition, we calculated the test coverage of the success-

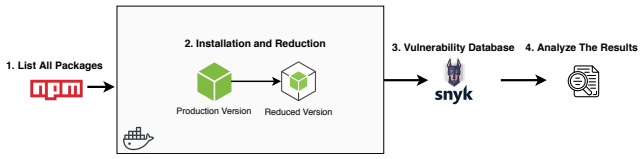

Figure 3: NPM measurement experiment setup

Job statuses and reasons	Packages
Succeeded packages	672,242
Failed packages	382,889
Package does not have main entry point	188,630
Non-resolvable dynamic import detected	128,533
Failed to install	26,875
Package's main entry point is not CommonJS	20,977
Others	5,013
TOTAL	1,055,131

Table 3: NPM measurement experiment overall status

fully reduced packages for both reduction modes. From all packages that successfully passed the validation test after reduction, more than third has 100% test coverage and almost forth have coverage between 90-99.9% for both coarse and fine-grain reductions, as shown in Table 2. This shows that Mininode can successfully reduce the packages without breaking the intentional behavior.

7.2 Attack Surface Reduction in NPM

Experiment Setup. The setup and stages of the measurement experiment are shown in Figure 3. First, we collected all package names from NPM. Second, we tried to install the production version of all packages and to run reduction logic on successfully installed ones. Finally, we analyzed the results and measured the vulnerabilities and their reduction.

We gathered all package names from NPM using the open-source package *all-the-package-names* [12] that contains the list of all package names sorted by dependent count. The list contained 1,055,131 package names from NPM as of 19th September 2019.

After gathering all the package names, we tried to install and reduce packages using the coarse-grain reduction method. Table 3 shows that only 672,242 out of 1,055,131 were successfully installed and reduced. Table 3 lists the most common reasons why not all of the packages were analyzed. Top two most common reasons are: (1) installed packages are not Node.js application, which means they are not intended to run on the server-side, *e.g.* theme's CSS files; (2) packages that can not be reduced with Mininode, due to non-resolvable dynamic import. One interesting failed category is packages' for which entry point is not CommonJS, *e.g.* ES6, or even not JavaScript file, *e.g.* TypeScript, JSON and so on.

In the fourth step, as shown in Figure 3, we gathered a vulnerability database from *snyk.io* [16] and mapped vulnerabilities with packages by calculating if specific vulnerable dependency is part of the dependency chain inside the package. In addition to mapping vulnerability, we calculate if Mininode removed the particular vulnerability during the reduction process. We consider that a specific vulnerable dependency is removed if Mininode removes all source files from it. Otherwise, we say that the package still depends on vulnerable dependency. Note that this is a conservative approach and gives us a *lower bound* reduction number because certainly Mininode *may* have removed a vulnerable file from vulnerable dependency, and left only safe files.

	Number
Removed fs built-in module	549,254
Removed net built-in module	623,646
Removed http built-in module	606,981
Removed https built-in module	614,030
Percentage of removed JavaScript files	79.1%
Percentage of removed LLOC	90.5%
Percentage of removed exports	90.4%
TOTAL	672,242

Table 4: NPM measurement experiment results

Results. The NPM measurement experiment reduction results are shown in Table 3. As discussed earlier, only 672,242 out of 1,055,131 were successfully installed and reduced. From all successfully installed and reduced packages, Mininode restricted access to `fs` built-in module in 81.7% packages, and it also restricted access to network-related built-in modules such as `net`, `http`, `https` in 92.8%, 90.3%, 91.3% packages, respectively. We discussed how Mininode restricts access to built-in modules in Section 6.1.

One question we tried to answer during the NPM measurement experiment was how significant is the severity of bloated-code in NPM packages. To answer this question, we calculated the relationship between declared and installed dependencies of the packages. On average, successfully analyzed packages declared 1.9 dependencies but installed 27.3 dependencies, which means NPM installed x14 times more dependencies than declared. This behavior is the result of the transitive dependency installation process discussed in Section 2.2. On NPM public registry, the package's detailed information shows the number of declared, *i.e.* direct dependencies, but not the number of actually installed dependencies. As a consequence, developers may choose packages with lower declared, but higher installed dependencies instead of packages with higher declared, but lower installed dependencies.

To give a more detailed insight of the bloatedness of NPM packages, we calculated the ratio between third-party and original code base's logical lines of code. On average, from all code-base, only 6.8% was original code, while 93.2% was external code from third-party dependencies, and from all

LLOC, only 9.5% were left after coarse-grain reduction by Mininode. This result clearly shows that more and more applications are developed as a mash-up of third-party packages and the need for reduction techniques.

We also measured the effectiveness of Mininode in reducing unused vulnerable dependencies from packages. These vulnerabilities are present in the application's code, but are not reachable. In order to get exploited, the attacker needs to chain vulnerabilities together (§ 3), something that might not be always possible. The results of vulnerability reduction analysis in `NPM` packages are given in Table 5. We used the vulnerability database from `snyk.io`, which contains 1,660 vulnerable packages grouped by categories. In total, we found that 119,433 of packages have at least one active vulnerable dependency by the time of writing. This corresponds to 17.8% of all successfully analyzed and reduced 672,242 packages. Table 5 shows the top ten most common vulnerability categories sorted by the number of unique packages that have a dependency from a specific vulnerability category. For example, 91,184 packages have *at least one* dependency vulnerable to Prototype Pollution. Partially removed column of the Table 5 shows the number of unique packages from which Mininode removed at least one vulnerability of a specific category. For example, if the package *@chrismlee/reactcards* had two vulnerable dependencies from the Arbitrary Code Injection category and Mininode was able to remove one of the vulnerable dependencies, then we count the package as partially removed. On the other hand, the fully removed column shows the number of unique packages where *all* vulnerabilities of the specific category were removed. Also, in Table 5, one can see the percentage of partially and fully removed packages from the total number of vulnerable packages. On average, Mininode was able to partially remove vulnerabilities from across all categories in 13.8% cases, and remove all vulnerabilities in 13.65% cases. In conclusion, Mininode was able to remove at least one vulnerability from 10,618 and remove all vulnerabilities from 2861 unique packages from all 119,433 vulnerable packages.

8 Related Work

Attack Surface Reduction. Howard *et al*. [30] introduced the notion of the attack surface, a way to measure the security of the system. Manadhata [35] generalized Howard's approach and introduced a step by step mechanism to calculate the attack surface of the system. Theisen *et al*. [48] came up with an attack surface approximation technique based on stack traces. There are several attempts both to reduce and to measure the attack surface of the different systems, such as OSes, websites, mobile applications [28, 40, 41, 51]. While all of the above works are related to attack surface reduction, we concentrate on the attack surface reduction of the Node.js applications. Azad *et al*. [21] showed that debloating the web application improves its security. They debloated the PHP application by recording the web application's code coverage from client-side interaction, which may break the website if rarely used functionality was not triggered during recording step. On the other hand, we use static analysis to create the dependency graph of the application, which covers all use-cases accessible from the application's entry point.

Node.js and `NPM` Security. Previous researchers on the security of Node.js concentrate more on injection attacks [17, 37, 42] and event poisoning attacks [23–25]. Ojamaa *et al*. was the first to assess the security of Node.js [38]. They conclude that denial of service is the main threat for Node.js. On the other hand, we concentrate on reducing the overall attack surface of Node.js rather than on specific attack or vulnerability.

NodeSentry [26] is a permission-based security architecture that integrates third-party Node.js modules with least-privilege. While NodeSentry also reduces the attack surface by using least-privilege modes for Node.js modules, we approached the problem from a different angle. Mininode removes unused functionality from third-party dependencies instead of restricting their functionality as NodeSentry does.

On the NPM side, researchers try to answer why developers use trivial packages [19] and the security implications of depending on NPM packages [52]. Zimmermann and *et al*.'s results supplement our results that depending on too many third-party packages significantly increase the attack surface [52].

JavaScript Application Analysis. In the past, researchers tried to come up with static [33,34,37] and dynamic [6,36,37] techniques that help developers with analysis of the application written in JavaScript. Madsen *et al*. [33] focuses on static analysis of JavaScript applications using traditional pointer analysis and use analysis. The key insight of the paper is the idea of observing the uses of library functionality within the application code to better understand the structure of the library code. Madsen *et al*. [34] introduced an event-based call graph representation of Node.js application that is useful to detect various event-related bugs. The advantage of the event-based call graph is that it contains information about listener registration and event emission that can be used to detect dead events and emits. Sun *et al*. [6] introduced a dynamic analysis framework called NodeProf that can be used for profiling, for locating bad coding practices, and for detecting data-race in Node.js applications. Mezzetti *et al*. [36] introduced a technique called type regression testing, which automatically determines if `NPM` package's update affects the types of its public interface, which eventually will introduce breaking changes for clients. While there exists many other JavaScript static analysis tools, Mininode differs because it mostly concentrates on building dependency graphs to reduce the attack surface.

JavaScript bundlers. Traditionally bundlers are used on the client-side to combine all the source code files into a single file to reduce network requests back to the server. One

Category names	Vulnerable packages	Partially removed	%	Fully removed	%
Prototype Pollution	91,184	5,333	5.85%	3,633	3.98%
Regex Denial of Service	42,163	3,930	9.32%	1,228	2.91%
Denial of Service	21,312	403	1.89%	370	1.74%
Uninitialized Memory Exposure	6,433	690	10.73%	592	9.20%
Arbitrary Code Execution	5,324	413	7.76%	396	7.44%
Cross-Site Scripting	5,142	665	12.93%	590	11.47%
Arbitrary Code Injection	3,451	1,715	49.70%	1649	47.78%
Remote Memory Exposure	3,323	16	0.48%	15	0.45%
Arbitrary File Overwrite	3,240	383	11.82%	381	11.76%
Information Exposure	3,088	47	1.52%	47	1.52%

Table 5: Common vulnerability categories and their reduction results. Some vulnerabilities might not be exploitable since their code is not directly reachable and it might not be possible to chain the vulnerabilities due to additional constrains.

of the most popular bundlers is *webpack* [18], that supports plugins and different file types, *e.g.* CSS, HTML. While the latest version of *webpack* can perform dead-code elimination, which is eliminating declared but unused functions and variables, Mininode removes exported functionalities that are never used outside the module, in addition to dead-code elimination. Another popular bundler is *rollup* [15] which can also remove unused exported functions from modules. However, *rollup* works only for ES6 module system, while Mininode was designed to work with CommonJS module system which is the most widely used in `NPM`. There are open-source plugins for both *webpack* and *rollup* tools that try to convert CommonJS module into ES6 module, but to our best of knowledge, they do not try to resolve the dynamic challenges that Mininode resolves (see §5 and §6). We envision that our work will be integrated into existing JavaScript bundlers.

9 Limitations

In this section, we discuss some of our evaluation and implementation limitations. First, using a test coverage metric to detect if Mininode breaks the original behavior can be misleading. For example, in the case of dynamic code generation, *i.e.* `eval`, test coverage may give 100% coverage even if it is not covering all functions. However, we argue that test coverage is the most appropriate mechanism that we can use to automatically perform a large-scale evaluation.

Second, we employed the *snyk.io* database in our vulnerability analysis measurement instead of the well-established CVE-DB or NIST. Unfortunately, despite the high quality of reports, both contain less number of reports related to third-party Node.js package vulnerabilities [32].

Third, the dynamic nature of JavaScript is a well-known challenge for static analysis. In this paper we tried to solve some Node.js specific challenges, such as *dynamic import*, and defining *aliases*, by using static analysis. However, there are challenges that cannot be easily resolved with static analysis. For example, one of those challenges is dynamic code generation using various JavaScript APIs, *e.g.* `eval`, `Function`, `setTimeOut`. Another challenge is patching Node.js specific APIs, *e.g.* `require`, as shown in Listing 8. In this case, Mininode will not be able to resolve a module inside a different folder, because it uses an unpatched version of `require`.

```
1 // patching the require
2 require = function(arg) {
3     return {mocked: true};
4 }
```

Listing 8: Example of patching the `require()`

A solution to this challenge can be to dynamically execute the patched code in Mininode to resolve the dynamically required module. Another approach is to forbid patching of `require` function in Node.js application by creating a constant global object `require` that can be accessed by all modules. This way, the function wrapper (See Listing 1) discussed in Section 2.1 does not need to pass `require` as an argument.

10 Conclusion

In this paper, we presented a detailed evaluation of excessive functionality in Node.js applications. We presented a tool, called Mininode, that measures and effectively removes unnecessary code and dependencies by statically analyzing Node.js applications. We conducted an extensive analysis of 672,242 packages listed in the `NPM` repository and found 119,433 of them to have at least one vulnerable module dependency. Our tool is capable of statically removing all vulnerable dependencies from 2861, and removing partially from 10,618 applications. In addition to removing vulnerabilities, Mininode was able to restrict access to the file system for 549,254 packages. We envision our tool to be integrated into the building process of Node.js applications. Mininode is publicly available at `https://kapravelos.com/projects/mininode`.

Acknowledgments

We would like to thank our shepherd, Johannes Kinder, and the anonymous reviewers for their valuable feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541 and by the National Science Foundation (NSF) under grant CNS-1703375.

References

[1] Babel JavaScript compiler. https://babeljs.io/.

[2] CVE-2020-7681, marscode vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7681.

[3] CVE-2020-7682, marked-tree vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7682.

[4] CVE-2020-7687, fast-http vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7687.

[5] Debug. https://www.npmjs.com/package/debug.

[6] efficient dynamic analysis for node.js.

[7] Escodegen. https://github.com/estools/escodegen.

[8] Esprima. https://esprima.org/.

[9] Module Counts. http://www.modulecounts.com/.

[10] Node.js. https://nodejs.org/en/.

[11] Node.js Documentation, ECMAScript Modules. https://nodejs.org/api/esm.html#esm_ecmascript_modules.

[12] NPM package: all-the-package-names. https://www.npmjs.com/package/all-the-package-names.

[13] Npm package: detective. https://www.npmjs.com/package/detective.

[14] Official Babel Documentation. https://babeljs.io/docs/en/config-files#project-wide-configuration.

[15] RollupJS. https://rollupjs.org/guide/en/.

[16] Snyk: Vulnerability DB. https://snyk.io/.

[17] understanding and automatically preventing injection attacks on node.js. Technical report.

[18] webpack. https://webpack.js.org/.

[19] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? An empirical case study on npm. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2017.

[20] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. Seven months' worth of mistakes: A longitudinal study of typosquatting abuse. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

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

[22] Matthew Baxter-Reynolds. Here's why you should be happy that microsoft is embracing node.js. https://www.theguardian.com/technology/blog/2011/nov/09/programming-microsoft.

[23] James Davis, Christy Coghlan, Francisco Servant, and Dongyoon Lee. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[24] James Davis, Gregor Kildow, and Dongyoon Lee. The Case of the Poisoned Event Handler: Weaknesses in the Node.js Event-Driven Architecture. In *Proceedings of the ACM European Workshop on Systems Security*, 2017.

[25] James Davis, Eric Williamson, and Dongyoon Lee. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *Proceedings of the USENIX Security Symposium*, 2018.

[26] Willem De Groef, Fabio Massacci, and Frank Piessens. NodeSentry: Least-privilege Library Integration for Server-side JavaScript. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[27] Emily Mitchell. Support for Node.js when you need it. https://developer.ibm.com/articles/support-offering-for-nodejs/.

[28] Sumit Goswami, Nabanita Krishnan, Mukesh Verma, Saurabh Swarnkar, and Pallavi Mahajan. Reducing Attack Surface of a Web Application by Open Web Application Security Project Compliance. *Defence Science Journal*, 2012.

[29] TJ Holowaychuk. NPM package: requires. https://www.npmjs.com/package/requires.

[30] Michael Howard, Jon Pincus, and Jeannette M. Wing. Measuring relative attack surfaces. In *Computer security in the 21st century*. Springer, 2005.

[31] Joel Kemp. NPM package: dependency-tree. https://www.npmjs.com/package/dependency-tree.

[32] Sherif Koussa. 13 tools for checking the security risk of open-source dependencies. https://techbeacon.com/app-dev-testing/13-tools-checking-security-risk-open-source-dependencies.

[33] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the ACM Joint Meeting on Foundations of Software Engineering*, 2013.

[34] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices*, 2015.

[35] Pratyusa K Manadhata, Kymie M Tan, Roy A Maxion, and Jeannette M Wing. An Approach to Measuring A System's AttackSurface. Technical report, CMU-CS-07-146, 2007.

[36] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type regression testing to detect breaking changes in Node.js libraries. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2018.

[37] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: feedback-driven static analysis of Node. js applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[38] Andres Ojamaa and Karl Düüna. Assessing the security of Node.js platform. In *Proceedings of the IEEE International Conference for Internet Technology and Secured Transactions*, 2012.

[39] Paypal Engineering. Node.js at PayPal. https://medium.com/paypal-engineering/node-js-at-paypal-4e2d1d08ce4f.

[40] Sebastian Ruland, Géza Kulcsár, Erhan Leblebici, Sven Peldszus, and Malte Lochau. Controlling the Attack Surface of Object-Oriented Refactorings. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, 2018.

[41] Mohamed Shehab and Abeer AlJarrah. Reducing Attack Surface on Cordova-based Hybrid Mobile Apps. In *Proceedings of the International Workshop on Mobile Development Lifecycle*, 2014.

[42] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and automatically preventing injection attacks on node. js. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2018.

[43] Janos Szurdi, Balazs Kocso, Gabor Cseh, Jonathan Spring, Mark Felegyhazi, and Chris Kanich. The Long "Taile" of Typosquatting Domain Names. In *Proceedings of the USENIX Security Symposium*, 2014.

[44] The npm blog. 'crossenv' malware on the npm registry. https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry.

[45] The npm blog. Details about the event-stream incident. https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry.

[46] The npm blog. kik, left-pad, and npm. https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm.

[47] The npm blog. Why we created npm enterprise. https://blog.npmjs.org/post/183073931165/why-we-created-npm-enterprise.

[48] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating Attack Surfaces with Stack Traces. In *Proceedings of the IEEE International Conference on Software Engineering*, 2015.

[49] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *Proceedings of the IEEE Internet Computing*, 2010.

[50] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting. In *Proceedings of the USENIX Security Symposium*, 2019.

[51] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2018.

[52] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proceedings of the USENIX Security Symposium*, 2019.