# NodeShield: Runtime Enforcement of Security-Enhanced SBOMs for Node.js

Eric Cornelissen
ericco@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Musard Balliu
musard@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden



## ABSTRACT

The software supply chain is an increasingly common attack vector for malicious actors. The Node.js ecosystem has been subject to a wide array of attacks, likely due to its size and prevalence. To counter such attacks, the research community and practitioners have proposed a range of static and dynamic mechanisms, including process- and language-level sandboxing, permission systems, and taint tracking. Drawing on valuable insight from these works, this paper studies a runtime protection mechanism for (the supply chain of) Node.js applications with the ambitious goals of compatibility, automation, minimal overhead, and policy conciseness.

Specifically, we design, implement and evaluate NodeShield, a protection mechanism for Node.js that enforces an application's dependency hierarchy and controls access to system resources at runtime. We leverage the up-and-coming SBOM standard as the source of truth for the dependency hierarchy of the application, thus preventing components from stealthily abusing undeclared components. We propose to enhance the SBOM with a notion of capabilities that represents a set of related system resources a component may access. Our proposed SBOM extension, the Capability Bill of Materials or CBOM, records the required capabilities of each component, providing valuable insight into the potential privileged behavior. NodeShield enforces the SBOM and CBOM at runtime via code *outlining* (as opposed to inlining) with no modifications to the original code or Node.js runtime, thus preventing unexpected, potentially malicious behavior. Our evaluation shows that NodeShield can prevent over 98% out of 67 known supply chain attacks while incurring minimal overhead on servers at less than 1ms per request. We achieve this while maintaining broad compatibility with vanilla Node.js and a concise policy language that consists of at most 7 entries per dependency.

## CCS CONCEPTS

• **Security and privacy → Web application security**.

## KEYWORDS

Web Security, Supply Chain Security, Node.js, SBOM


*CCS'25, October 13–17, 2025, Taipai, Taiwan*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

**ACM Reference Format:**
Eric Cornelissen and Musard Balliu. 2025. NodeShield: Runtime Enforcement of Security-Enhanced SBOMs for Node.js. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS'25).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Prominent attacks such as SolarWinds, Log4Shell, and XZ Utils have brought software supply chain security to a top priority for both the academic community and industry practitioners. A recent approach to help mitigate security risks has been the introduction of the Software Bill of Materials (SBOM), a machine-readable inventory of the ingredients of a software application, including the components, their metadata, and their internal relationships. While the effectiveness of SBOMs is still subject to debate [46], the community agrees that they increase software transparency, contributing as a reactive measure to identify one-day vulnerabilities, e.g., in applications using vulnerable components. This paper envisions an enhancement of SBOMs with capabilities (a list of sensitive APIs that components use), dubbed CBOM, and takes it a step further by enforcing these capabilities at runtime, thus making CBOM and SBOM a proactive measure against supply chain attacks.

Node.js and its npm ecosystem of software components, or packages, have been a particularly attractive target for supply chain attacks [28, 35, 44]. Beyond the security challenges of JavaScript, this is facilitated by a culture encouraging the use of many packages and automatic dependency updates. Research shows that a single package trusts on average 79 other third-party packages, relying on code published by 40 maintainers [64]. This creates an invisible attack surface, where a single compromise has widespread consequences [29, 32, 41, 42, 52]. To further exacerbate the risks, Node.js applications typically run with full access to system resources, which third-party packages inherit. In this context, a security system specifically designed for the Node.js supply chain is crucial. By enforcing the SBOM hierarchy and the CBOM capabilities on package behavior—such as restricting access to files, network, or processes—we can significantly reduce the risk of malicious updates while enhancing transparency and control within the ecosystem.

NodeShield [24] contributes with the design and implementation of a practical open-source system that focuses on the supply chain of Node.js applications, while ensuring Node.js compatibility, automation, minimal overhead, policy conciseness, and robustness to attacks. To the best of our knowledge, there is no system that meets all these goals. As shown in our evaluation (Section 5), related works on lightweight permission systems for Node.js [31, 45] show limitations with regards to compatibility, policies, and attack robustness. Finer-grained approaches such as language-level sandboxing [7, 15, 26, 59, 60] and taint analysis [16, 19, 47, 49, 51, 56]

protect against more attacks at the expense of performance overhead, false positives, automation, and policy conciseness.

At the heart of NodeShield lies a non-intrusive package-level instrumentation, called outlining, that enforces the SBOM and CBOM policies at runtime, with no modifications to the original code or Node.js runtime. We achieve this by a principled design and security analysis that considers access to system resources and enforcement bypasses. In particular, we develop a novel lexical scoping-based approach that reduces the impact of bypassing the enforcement and provides support for both legacy and modern Node.js modules, CommonJS and ESModules. Through a methodological analysis of Node.js APIs and related systems, we propose seven security-sensitive capabilities for the CBOM, thus creating a concise policy language. NodeShield enforces these capabilities through a comprehensive coverage of three avenues: built-in modules, global variables, and native bindings. Moreover, it supports automated policy generation along with features that facilitate CBOM presentation at different granularity levels to aid adoption for developers.

To evaluate the effectiveness of NodeShield, we contribute with a benchmark of 67 known supply chain attacks and show that it can prevent 98.51% of them. We further find that NodeShield is effective at reducing the impact of arbitrary code execution vulnerabilities by detecting 87.50% of the exploits from SecBench.js [18]. Our performance evaluation on long-lived applications shows a response time overhead of less than 1ms and a throughput reduction of up to 360 requests per second. We also find that NodeShield is broadly compatible with software written for Node.js, despite some incompatible coding patterns being used in practice. We evaluate the maintenance effort of CBOMs to find that application developers need to review less than 1 capability per dependency update on average, and between 0 and 13 capabilities for new dependencies. Our robustness analysis against sandbox bypasses shows that NodeShield prevents all attacks within our threat model. Finally, our evaluation concludes with an end-to-end analysis of the infamous attack on Copay [41], showing the usefulness of NodeShield in preventing the attack. In summary, we offer the following contributions:

- The first Node.js runtime protection tool that comprehensively protects both CommonJS and ESModules-based source code.
- A novel technique to contain Node.js sandbox breakouts, providing better guarantees than related work on malware protection.
- An SBOM extension to capture the privileges required by supply chains components at the granularity of packages.
- A comprehensive evaluation of effectiveness, performance, compatibility, maintainability, and robustness to attacks.

NodeShield and all experiments are available at https://github.com/KTH-LangSec/nodeshield.

## 2 PRELIMINARIES

**Node.js** Node.js is a JavaScript runtime intended for server-side applications built on the V8 engine from the Chromium browsers extended with APIs for access to system resources. Node.js supports to module systems: CommonJS and ESModules. The former leveraging legacy JavaScript features to establish namespaces and supports importing modules using the `require` (or `import`) function. The latter is a new format with native namespacing and syntax for importing and exporting modules (besides the `import` function).

Node.js applications rely on third-party dependencies, or *packages*. Packages are usually distributed via a registry (e.g., `npmjs.com`) and managed by a package manager (e.g., npm or yarn). Direct dependencies of an application are stored in the manifest file with, optionally, all dependencies listed in a lockfile (`package.json` and `package-lock.json` resp. for npm). Node.js uses the `node_modules` directory to store and access third-party packages. Packages can be reused by name through the `require` function (`require("vm")` in CommonJS), *import syntax* (`import vm from "vm"` in ESModules), or `import` function (`import("vm")` in both).

**System interface** Node.js provides access to system resources in two avenues. First, the Node.js built-in modules provide APIs to interact with the underlying system (e.g., the `fs` module). These can be imported like other packages. Second, Node.js extends the list of global variables with built-in objects, APIs, and properties that provide system access (e.g., the `process` global). Either way, this interface is enabled by bindings between the JavaScript world and underlying C++ codebase, which are accessible directly through the (undocumented) `process.binding` function.

**Code evaluation** There are two main APIs for dynamic code evaluation in Node.js. First, the `eval` function (and friends, like `new Function`) allow for basic dynamic code evaluation [5]. A call to `eval` evaluates a string as JavaScript in the current lexical scope. Use of `require` or `import` with `eval` is possible, but the ESModules import and export syntax is not.

Secondly, Node.js provides a built-in module for code evaluation called `vm`. This module offers an API for evaluating both CommonJS and ESModules JavaScript in a new V8 context while allowing object sharing between contexts. A V8 context is an separate interpreter context within an interpreter process, allowing multiple contexts to share JavaScript objects within the same process. Each V8 context, including the top-level context created when a Node.js application starts, can be configured independently.

**SBOM** A Software Bill Of Materials (SBOM) is a document that lists all the components used in a software artifact. For each component it may include various pieces of information ranging from checksums to licensing information. Additionally, a good SBOM includes the dependency hierarchy of all components, specifying which components each one uses. There exist two widely-used specifications for SBOMs, SPDX and CycloneDX, along with myriad of automated tooling to generate SBOMs for software artifacts.

## 3 OVERVIEW

This section provides a high-level overview of the research problem and solution. It also discusses the threat model and the proposed usage of capabilities as a security enhancement of SBOMs.

### 3.1 Challenges and Solution Overview

**The problem** We use the example of the `rate-map` package to illustrate our research problem. This is a very simple benign package that maps number in the range of `[0,1]` to a new value with a given range. It is written using a combination of CommonJS and ESModules. The benign version v1.0.2 of `rate-map` depends on one other package, `append-type`, and implements a single function in JavaScript with no Node.js-specific functionality (line 4 in Listing 1).

```
1  const px = require.resolve(Buffer.from([100, 108, 45,
       116, 97, 114]/*=dl-tar*/).toString());
2  const fs = require("fs");
3  fs.writeFileSync(px, fs.readFileSync(px, "utf8").
       replace(a, b));
4  module.exports = function rateMap(val, start, end)
       {... return start + val * (end - start)}
```

**Listing 1: Simplified `rate-map` malware.**

Listing 1 shows the compromised version v1.0.3 breaking the functionality of the `purescript-installer` package which used `rate-map` among its dependencies [32, 53]. The malicious version adds an explicit dependency on the package `terser` to its package manifest (not shown). However, it does not use `terser`, instead it covertly resolves the package `dl-tar` at runtime (line 1). This behavior hinders supply chain transparency and may be challenging to detect due to the use of obfuscation. Thus, our first challenge is to identify and enforce the expected dependency hierarchy of applications, and detect usage of packages outside this hierarchy.

Secondly, the malicious version introduces the use of Node.js's `fs` module (line 2), which grants access to privileged file system APIs (line 3). Unfortunately, the current use of Node.js and npm (or similar) provides no insight into the use of privileged APIs. Thus, our second challenge is to identify and enforce the expected use of access to privileged resources on the supply chain of applications.

The attack combines these two malicious changes to modify the implementation of the `dl-tar` package at runtime. The attacker's goal is to cause a denial of service, e.g., on the installation of the PureScript application. As highlighted, the attack can be detected at two stages: when an unexpected third-party dependency is resolved or when a new privileged API was adopted by the package.

**Our solution** To address this problem we aim to design a system that protects against software supply chain attacks as illustrated in Figure 1. At a high level this is achieved by 1) enforcing the application dependency hierarchy according to its SBOM and 2) enforcing restrictions on the use of privileged APIs per dependency—modeled as "capabilities"—covered in detail in Section 4.2. This is enhanced by a principled design and security analysis, 3) applying hardening techniques to prevent bypassing our enforcement, as discussed in Section 4.3, and a thorough empirical evaluation, cf. Section 5.

NodeShield builds on related work on runtime hardening for software supply chain attacks in JavaScript [7, 31, 45, 60] and language-level JavaScript sandboxing [15, 59]. In particular, we extend these techniques to support ESModules and propose a novel lexical scoping-based approach that reduces the impact of sandbox breakouts. The result is a system that hardens against supply chain attacks and bridges the existing gap with JavaScript sandboxes.

In practical terms, NodeShield enforces the dependency hierarchy by controlling what a package is allowed to import. This is achieved by trapping and validating all import attempts (covering `require`, `import`, and import syntax) on a per-dependency basis. For `rate-map`, NodeShield would have prevented the covert import of `dl-tar`, as represented by the crossed-out line from $D_D$ to $D_G$.

The enforcement of capabilities is achieved through a comprehensive coverage of three separate avenues: built-in modules, global variables, and native bindings. First, the use of privileged built-in

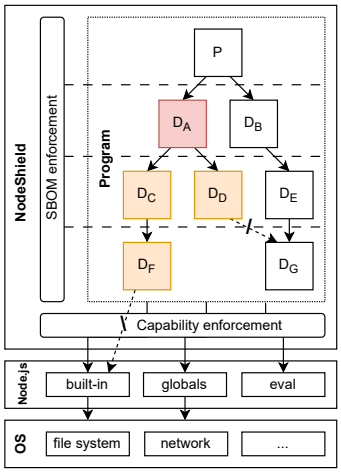


**Figure 1: Overview and Threat Model of NodeShield.**

modules is controlled in the same way as the dependency hierarchy. Second, access to global variables is controlled by adapting the `vm` module to create package-specific global namespaces. Lastly, access to bindings is controlled by overriding the `process.binding` API. In the case of `rate-map`, NodeShield would have caught the newly introduced use of `fs`. In Figure 1, this is represented by the crossed-out line from $D_F$ to the Node.js built-in modules.

Finally, NodeShield is hardened using primordials [23], `null` prototypes, object freezing, and lexical scoping. This prevents application code from manipulating the policy and achieving anything by breaking out of its `vm` context.

## 3.2 Threat Model

Our threat model focuses on server-side applications that use untrusted third-party dependencies and run on top of the Node.js JavaScript runtime, as illustrated by the highlighted squares in Figure 1. The attacker can control an arbitrary set of packages in the dependency hierarchy of an application (which includes its transitive dependencies). We assume that Node.js, the package manager (e.g., npm or yarn), the package registry (e.g., npm), the SBOM generator, and the supply chain of NodeShield are not compromised. Since application dependencies may be openly malicious, developers are expected to review all capabilities. This can be a significant upfront investment for pre-existing apps, but Section 5.4 shows it is manageable as a continuous task.

The goal of the attacker, based on empirically-observed behavior of real JavaScript malware [44], is to break the confidentiality or integrity of the system on which the application runs by using capabilities outside the set granted to attacker-controlled dependencies.

We consider the act of providing any data or reference to a dependency as entrusting that dependency tree with it, including capability-bearing references. The justification for this setup is that developers use dependencies to perform actions on their behalf on the provided data using any capabilities granted. We note that the global namespace of the application is considered as an intentional interface between program components, and is therefore left unprotected. The implication is that malicious code could exploit gadgets

through this avenue, such as through prototype pollution [50]. We further motivate this decision in Section 4.3.

### 3.3 Enhancing SBOM with Capabilities

A key goal for NodeShield is to provide a simple and comprehensible policy language for developers, while effectively and efficiently preventing malware in the software supply chain. To this end, we argue that an enhancement of SBOM with a Capability Bill of Materials (CBOM), at the granularity of packages, strikes the right balances between these requirements, as witnessed by our evaluation in Section 4. We use the term "capability" to refer to a group of related privileged operations.

We determine the set of capabilities through a comprehensive review of existing proposals across different ecosystems. Specifically, we use the following methodology: First, we collect a set of potential capabilities from related work in the area of supply chain security [31, 34] and related tooling (Node.js' and Deno's permission system [8, 27], Capslock [3], and Cackle [2]). Then we filter this set of potential capabilities by omitting those unrelated to supply chain attacks according to the SAP supply chain attack tree [35], Socket.dev supply chain alerts [10], and the OpenSSF metrics and data workgroup's OSE threat model [11].

From this analysis, we settle for a list of 7 capabilities listed below. Table 1 provides an overview of the mapping from capability to modules, global variables, and bindings.

- The `addon` capability represents the ability of a dependency to load native code. This is included because native code can bypass our enforcement, enabling misuse of other capabilities.
- The `code` capability represents the ability to dynamically evaluate code. While dynamic code is subject to the other capabilities, this capability enables explicit control, thus facilitating manual review and defense in depth.
- The `command` capability represents the ability to run a command as subprocess. Subprocesses can bypass our enforcement, enabling the potential use of any other capability.
- The `crypto` capability controls usage of cryptography-related functionality. This is included because of obfuscation and ransomware attacks utilizing cryptography.
- The `file-system` capability represents all access to the file system. This is included because malicious packages may read sensitive data, e.g., cryptographic material, from the file system.
- The `network` capability represents all access to the network. This is included because malicious packages may exfiltrate sensitive data or fetch malicious code or commands from a remote server.
- The `system` capability represents all access to system and environment information. Malicious packages may read sensitive data, e.g., authentication tokens, from the environment variables.

## 4 NODESHIELD

This section details the design and implementation of NodeShield [24]. Figure 2 present an overview of NodeShield's architecture and workflow. NodeShield takes as input the source code of an application (including own and dependencies' source code), the application's SBOM (including the hierarchy of dependencies), and (optionally) the application's CBOM. From this it automatically creates a clone of the original project which supports runtime enforcement of the SBOM and CBOM. As an intermediate step, it creates a module-granular policy representation of the SBOM and CBOM, for both the application and every (transitive) dependency. If no CBOM is provided, NodeShield infers it statically—as described in Section 4.4—before policy creation. The policy specifies what each module is allowed to import—files, third-party packages, built-in modules—what global variables it is allowed to access, and what bindings it is allowed to use.

To enforce the policy at runtime, NodeShield extends (but does not modify) the original source code on a per-file basis such that the file's code is evaluated in a `vm` context that enforces the policy of the module to which the file belongs. In contrast to inlining, we dub this process *outlining* and describe it in Section 4.1. The enforcement uses unmodified Node.js and `vm` APIs (Section 4.2) and is hardened to prevent policy manipulation and bypasses (Section 4.3).

### 4.1 Outlining

To enforce the SBOM and CBOM at runtime, NodeShield outlines the original source code within enforcement code. The details of the enforcement are discussed in Section 4.2, here we present the transformation of the original source code to the *outlined* one. As a basis, the outlining starts by creating a clone of the original application project. This clone only covers JavaScript source code files, JSON files, and native extensions. Other files in the project are accounted for by setting the working directory of the final program and rewriting paths present in Node.js APIs as part of the outlining. The cloned project can be run using vanilla Node.js.

The outlining procedure targets all JavaScript source code files, while JSON files and native extensions remain unchanged. In particular, the original source code is moved into a (multi-line) string that will be evaluated using `vm`. Prior to evaluation, NodeShield cleans up and prepares the host context (i.e., the top-level Node.js context) and the guest context (i.e., the `vm` context evaluating the original source code). Listing 2 shows the outlining process.

First, before any untrusted code is evaluated, the cloned project obtains references to primordials (line 2, more on this in Section 4.3) and privileged global variables (as defined in Table 1) which are also removed from the global namespace (line 3). Both need to happen only once. Next, for every CommonJS file, references to the CommonJS variables (`exports`, `module`, `require`, `__dirname`, `__filename`) must also be obtained and removed from the global namespace (line 4).

Then, it initializes an allowlist `I` for importing as the combination of 1) the JavaScript and JSON files in the module (line 6), 2) the packages that are listed as its direct dependencies in the SBOM (line 5), 3) the non-privileged built-in modules (line 8)—spanning all built-in modules not declared in Table 1—and 4) the allowed built-in modules in accordance with the CBOM, following the mapping of Table 1 (line 9). The resulting list (line 10) is used when importing (line 26 and 27) or requiring (line 21) to determine if the specified identifier is allowed by the policy.

Next, the allowlist `B` of bindings is initialized as the combination of 1) the non-privileged bindings (line 12)—spanning all bindings not declared in Table 1—and 2) the allowed bindings in accordance with the CBOM, following the mapping of Table 1 (line 13). The

| Capability | Modules | Global Variables | Bindings |
|---|---|---|---|
| addon | `*.node` | | |
| code | `node:vm` | `eval`, `Function` | |
| command | `node:child_process`, `node:worker_threads` | | `spawn`, `spawn_sync` |
| crypto | `node:crypto` | `Crypto`, `crypto`, `CryptoKey`, `SubtleCrypto` | `crypto` |
| file-system | `node:fs`, `node:fs/promises` | | |
| network | `node:net`, `node:http`, `node:http2`, `node:https`, `node:dns`, `node:/promises`, `node:tls`, `node:dgram` | `fetch` | |
| system | `node:os`, `node:process` | `process` | |

**Table 1: Capability mapping to modules, global variables, and binding. Built-in module names are listed as `node:*` but are also accessible without the `node:` prefix.**

```
1  INPUT: module, SBOM, CBOM
2  once(capture primordials)
3  once(capture and unbind sensitive global variables)
4  if (cjs) capture and unbind CJS global variables
5
6  Ia = files in module
7  Ib = packages according to SBOM
8  Ic = allowed-by-default built-in modules
9  Id = privileged built-in modules according to CBOM
10 I = [...Ia, ...Ib, ...Ic, ...Id]
11
12 Ba = allowed-by-default bindings
13 Bb = privileged bindings according to CBOM
14 B = [...Ba, ...Bb]
15 process.binding = (s) => if (s in B) process.binding(
       s) else halt
16
17 Ga = allowed-by-default global variables
18 Gb = privileged global variables according to CBOM
19 G = {...Ga, ...Gb}
20 if (cjs) {
21   G.require = (s) => if (s in I) import(s) else halt
22   ... other CJS variables
23 }
24
25 C = vm.NewContext(G, {
26   import: (s) => if (s in I) import(s) else halt })
27 vm.Link((s) => if (s in I) import(s) else halt)
28 vm.Eval('<original source code>', C)
```

**Listing 2: Outlining Process of NodeShield.**

resulting list (line 14) is used when accessing bindings through the `process` object (line 15).

Next, the global namespace `G` for the guest context is initialized as the combination of 1) all non-privileged global variables (line 17)—spanning all global variables not declared in Table 1, 2) the allowed global variables in accordance with the CBOM, following the mapping of Table 1 (line 18), and, for CommonJS, 3) the CommonJS-specific variables (line 21-22). The latter are not provided as-is, instead `require` is modified to check against the import allowlist and to present an empty module cache (which otherwise grants direct access to all previously loaded modules). Moreover, `module`, `__dirname`, and `__filename` are modified to reflect the original source code file paths.

Finally, the context for `vm` is initialized (line 25-26) and the source code evaluated with `vm` using this context (line 28). With the outlining process applied to all JavaScript files in the project, the policy

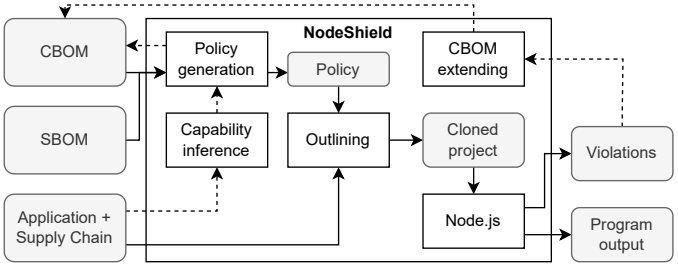


**Figure 2: Architecture and Workflow of NodeShield.**

enforcement is encoded directly into the program, which can now be executed safely as a regular Node.js application.

**Example** For Listing 1, Listing 2 would concretely instantiate with the code on line 28; `Ia` as [index.js, package.json], `Ib` as [append-type, terser], `Ic` as [assert, console, *etc.*], and `Id` as [] (line 6-9); `Bb` as [] (line 12-13); `Ga` as [atob, console, *etc.*] and `Gb` as [] (line 17-18). These lists are the policy at runtime—derived directly from the SBOM and CBOM—e.g., requiring `fs` (line 2, Listing 1) is disallowed on line 21 because it is not in the allowlist `I`.

## 4.2 Enforcement

NodeShield implements the policy enforcement by managing the global namespace per module, intercepting any import attempts by the program, controlling dynamic code evaluation, and managing bindings access. This is achieved by utilizing only mechanisms provided by Node.js and the `vm` API, with no modifications to Node.js.

**Global variables** To manage the global namespace of all modules, we define specialized sets of global variables (line 17-19, Listing 2) and utilize the option to specify a "context" object when using `vm` (line 25, Listing 2). By default, only standard JavaScript built-ins (e.g., `Object` or `Error`) are available. Specifying a context object allows for extending and overriding the namespace through the properties of the context object which are accessible as plain identifiers in the evaluated script. Thus, by omitting capability-bearing global variables that the current module should not have access to we can prevent it from using that capability.

Because we control the creation of all modules, we can control their global variables as specified in the CBOM. The set of global variables made available to each module starts from the set of all global variables in the host context, excluding capability-bearing

global variables (line 17, Listing 2). This set is extended with the allowed capability-bearing global variables of the module, according to the CBOM (line 18, Listing 2).

**Imports** To comprehensively intercept all import attempts we need to cover all three import options: `require` for CommonJS, `import` declarations for ESModules, and the `import` function for both. The `require` function must be provided through the global namespace (line 21, Listing 2), following the above description. As such, we can intercept all calls to `require`. For `import` declarations `vm` requires an implementation of a resolution algorithm through its `link` API when instantiating a (ESModules) module (line 27, Listing 2). Hence, we intercept all uses of the `import` syntax. Finally, for the `import` function to work, `vm` also requires an implementation of a resolution algorithm through the `importModuleDynamically` API (line 26, Listing 2) when instantiating either a (CommonJS) script or (ESModules) module. As a result, we also intercept all uses of the `import` function.

A fourth consideration is the built-in module named `module` which has a `createRequire` function. This function can be used to create a new `require` function which could bypass our enforcement. Because accessing this function requires importing `module`, following the previous paragraph, we can intercept all attempts to import `module`. We leverage this to return a modified `module` interface where the `createRequire` function creates `require` functions that perform the import policy check.

Given our ability to intercept all import attempts by the application, we can enforce dependency imports according to the SBOM by allowing an import if the specifier matches that of a listed dependency. Similarly, we can enforce the CBOM by allowing imports of capability-bearing built-in modules only if the corresponding capability is granted. Additionally, the import is allowed if it is for a file within the current package or an unprivileged built-in module.

**Code evaluation** To manage the ability of modules to dynamically evaluate code we use the `codeGeneration` option of the `vm` module. This option tells `vm` whether or not to allow dynamic text- or WASM-based code evaluation. Disabling the former disables `eval` (and related), while disabling the latter disables the functionality of the `WebAssembly` object. In both cases, use of these APIs results in an immediate error when called.

Since we control the creation of all modules, we can control the dynamic code evaluation capability for every module in accordance with the CBOM. Observe that even though `vm` can be used to create a new context where dynamic code evaluation is possible (in fact, NodeShield relies on this fact), the ability to import `vm` is guarded by the same capability as dynamic code evaluation, meaning it cannot be used to escalate capabilities.

**Bindings** To manage the binding accessible to each module we need to manage the behavior of `process.binding`. The `process` object is accessible both as a global variable (named `process`) and an import (as `(node:)process`). As described, we control access to both global variables and imports. Hence, we override the `binding` function on the `process` object (line 15, Listing 2) on a per-module basis with an implementation that restricts access to capability-bearing bindings in accordance with the CBOM.

**Enforcement modes** NodeShield supports 3 enforcement modes. First, in `log` mode it will log policy violations, along with the violating module, but otherwise continue as usual - thus not preventing the violation. Second, in `throw` mode violations will result in an error being thrown. This provides an option for the program to continue in spite of the limitation imposed on it. Lastly, the `exit` mode will result in an immediate program exit upon the first violation.

## 4.3 Security Analysis

We presents, given our explicit assumptions, a security analysis of NodeShield through the lens of attack scenarios that aim to bypass the enforcement and argue for its robustness against these attacks. The robustness is further evaluated empirically in Section 5.3.

The enforcement as described in Section 4.2 is comprehensive in that it covers all avenues through which Node.js code can use third-party libraries or access capability-bearing APIs. However, Node.js has several features related to meta-programming and dynamic code evaluation that could allow advanced malware to bypass the enforcement. For example, a naive implementation of the enforcement could be bypassed by monkey patching, e.g., the prototype of JavaScript built-ins such as `Object` or `Array`. Moreover, attacker can perform cross-V8-context code evaluation to achieve privilege escalation when evaluating in a more privileged context.

**Assumptions** The security of NodeShield is predicated on the following assumptions on JavaScript, Node.js, and the `vm` module. First, dynamic code evaluation in JavaScript is limited to accessing only variables from the current scope—per the language specification [5]. Second, the only three ways to load modules in Node.js are the `require` function (in CommonJS), the import syntax (in ESModules), and the `import` function (both)—per the Node.js docs [22]. Third, we assume the `vm` module 1) traps all uses of both the import syntax and function, 2) can prevent use of dynamic code evaluation APIs and 3) provides a fresh, pure JavaScript context without Node.js-specific built-ins or implicit access to variables from the context in which it is instantiated—the former two are supported by the Node.js docs [22] while we empirically validate the latter.

**Shared global namespace** Without separation of the global namespace between the host and guest code, the guest code could manipulate the behavior of the host by changing built-ins or performing prototype pollution. Rather than trying to contain the guest code, we write the host using techniques that prevent such influence. In particular, we leverage primordials, `null` prototypes, and object freezing.

First, we use primordials (line 2, Listing 2), a concept borrowed from Node.js [23]. Primordials are a set of function references obtained before any untrusted code is run such that untrusted code cannot manipulate these references. For example, our policy enforcement for imports needs to check if the import specifier is present in the allowlist of imports. We implement it as a JavaScript array with the goal of using `l.includes(s)` to check if the specifier `s` is allowed to be imported. However, the guest code could override `Array.prototype.includes` to always return true, invalidating import-based policy enforcement. To prevent this attack, we obtain a primordial reference to the `includes` function[1] and use

[1] as `Function.prototype.call.bind(Array.prototype.includes)`. Used as `includes(allowlist, specifier)`.

it instead. Additionally, as a defense-in-depth strategy, we leverage `null` prototype and object freezing. The use of `null` prototype avoids prototype pollution gadgets in our implementation, while object freezing avoids manipulating in-memory policy objects.

This prevents exploitation of the host through the global namespace, but not other (more privileged) guests. Indeed, changes to the `globalThis` object or standard object prototypes could be used by the attacker to trigger gadgets [25] in other guests. The reasoning for this gap is twofold. First, modifications in the global namespace happen in benign use cases such as polyfills, which we do not want to break. Second, known defenses against this type of attack, e.g., object freezing, can be implemented at the application level in a way that is compatible with such benign use cases (which we cannot do in NodeShield).

**Host-context eval** The use of `vm` introduces the possibility of dynamic code evaluation in the host's context, potentially allowing malicious code to bypass the policy enforcement. We propose a novel technique based on lexical scoping of variables to mitigate privilege escalation in this manner. In particular we prune sensitive global variables (i.e., those from Table 1 plus all CommonJS specific global variables) from the global namespace of the host context, capturing them as local variables instead and `delete`ing them from the `globalThis` object afterwards (line 3-4, Listing 2).

This relies on the observation that cross-context code evaluation does not allow the evaluator to access local variables of the evaluatee. Thus by pruning sensitive global variables the evaluatee does not gain additional capabilities by evaluation code in the host context.

**Cross-V8-context eval** A similar situation may occur when dynamic code is evaluated in another (more privileged) guest context. Here too, rather than trying to prevent cross-V8-context code evaluation, we use the same scoping-based technique to prevent such evaluation from enabling privilege escalation.

To achieve this we create one-time-accessible global variables which are bound to a local variable in the guest context through an inserted a preamble. To create a one-time-accessible global variable we define a property with a `getter` that deletes itself.

```
1  Object.defineProperty(G, "fetch$2eb2a5", {
2    configurable: true, get() {
3      delete G["fetch$2eb2a5"]; return fetch } })
```

The inserted preamble binds the one-time-accessible global variables to the correct local names.

```
1  { let fetch = fetch$2eb2a5        // preamble
2  { fetch("http://example.com") }}  // original code
```

From the perspective of the guest, accessing this local variable behaves just like the original global variable. But, because it is a local variable, it is not accessible in cross-V8-context code evaluation. Neither is the original global variable because accessing it removed it from the context's global namespace.

For CommonJS, we wrap the guest code in a block statement which in turn is wrapped in a block statement containing a (`let`) binding of the sensitive global variables to local variables, like the example (this approach prevents syntax errors due to re-used identifiers). For ESModules, we cannot wrap the entire guest code in any kind of block statement (because `imports` and `exports` must be at the top level) so we instead insert a top level statement that (`let`) binds sensitive global variables to local variables. To avoid syntax errors due to duplicate identifiers, ESModules code is parsed to determine and omit top-level names that are already defined.[2]

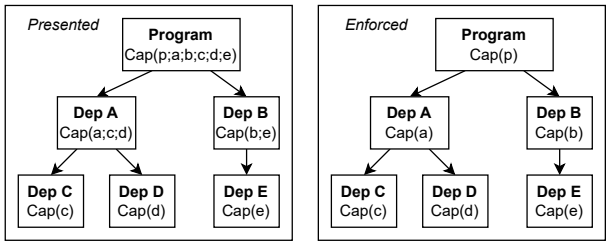


**Figure 3: Presented vs Enforced Policy in NodeShield.**

## 4.4 Capability Inference and Presentation

In the absence of a CBOM a set of capabilities needs to be determined. NodeShield offers two strategies with different benefits and drawbacks. First, capabilities can be inferred dynamically by running the application using an incomplete CBOM. From the observed violations, gaps in the CBOM can be filled in. However, this risks running malware without restrictions. Second, capabilities can be inferred statically. This is imprecise given the challenges of analyzing JavaScript code, but is safer because no untrusted code is run.

**Static inference** To statically infer the capabilities, all source code must be analyzed looking for imports of Node.js built-in modules, imports of addons, use of capability-bearing global variables, and use of capability-bearing bindings. For the prototype implementation of NodeShield we use regular expressions to match for `importing` or `requireing` built-in modules and the presence of the word "fetch", "eval", and "process". This is imprecise but largely sufficient for the purposes of our evaluation. In particular, it may falsely grant capabilities if, e.g., these patterns occur in a string, or miss capabilities if, e.g., they are used dynamically only.

**Dynamic inference** To dynamically infer capabilities, the application must be run in `log` mode, or iteratively in `exit` mode, to capture all policy violations. The policy violations can be inspected for capability-related violations, based on which the CBOM can be created or extended. Because of our use of the `codeGeneration` option, `log` mode cannot infer usage of the *code* capability, requiring `exit` mode to be used instead. The prototype implementation of NodeShield does not provide an automated mechanism to create/update the CBOM in this way.

**Enforced vs. presented** Capabilities are enforced at a module-granular level, meaning if dependency $D_1$ requires the capability $C_1$ it receives the ability to use exactly and only that capability. When we introduce $D_2$, requiring some capability $C_2 (\neq C_1)$, as a dependency of $D_1$ a potential confused deputy problem arises: $D_1$ can use $D_2$ (in accordance with SBOM enforcement) to exercise capability $C_2$ (in violation with the CBOM enforcement). We address this by separating the *enforced* (as described) and *presented* view of the CBOM, as illustrated in Figure 3.

[2]In both cases, adding this preamble accounts for file "headers" such as shebangs and `"use strict";`.

The presented view aims to resolve the confused deputy problem (arising from the dependency hierarchy) at the moment the human reviews the CBOM while maintaining least-privilege enforcement at runtime (a motivating example for this can be found in Section 5.2). In particular, NodeShield users are presented a view of the dependency in which it will be granted the capabilities it needs itself as well as the capabilities of all its transitive dependencies. This leaves no deputies with extra capabilities accessible to a malicious dependency for it to confuse.

For the example above, the *enforced* view of $D_1$ is $\{C_1\}$ while the presented view is $\{C_1, C_2\}$. This is because the code of $D_1$ itself only requires $C_1$ to function correctly—hence there is no need to grant it additional capabilities at runtime—while the overall behavior of the code comprising $D_1$ (i.e., including $D_2$) requires both $C_1$ and $C_2$ to function correctly.

# 5 EVALUATION

This section presents the evaluation of NodeShield [24], answering the following research questions:

RQ1. Is NodeShield effective at preventing software supply chain attacks?
RQ2. Is NodeShield effective at reducing the attack surface of dependencies?
RQ3. Is NodeShield robust against sandbox breakouts?
RQ4. What is the level of effort required to maintain a CBOM?
RQ5. What is the cost of using NodeShield compared to Node.js?

We compare NodeShield to related work [31, 45] on these questions. The tool from [31] is not available in a working state, hence we omit it. The Npm Dependency Guardian (ndg) tool [45] is available and we compare with it in RQ1, 2, 3, and 5.

*Experimental setup.* All experiments were conducted on a desktop system with an AMD Ryzen™ 7 3700X × 16 processor and 32 GB of RAM. For each experiment, the setup and supporting material is available in the artifact, except for the malicious code samples which are only made available upon request. All SBOMs were generated using the `npm sbom` utility.

## 5.1 RQ1: Effectiveness against Malware

To evaluate the effectiveness of NodeShield against software supply chain attacks we run known malicious npm packages and observe whether the attack can be stopped. In particular, we investigate whether the attack is detected by NodeShield through either SBOM or CBOM enforcement, and compare to ndg [45].

We compile a list of known malicious packages from prior work [28, 33, 34, 44] and gray literature [57, 62]. We obtain samples of the malicious code from [28, 44] as well as Socket (https://socket.dev/). In total, we collect the names of 2,179 known malicious packages (with one or more version) with samples available for (at least one malicious version of) 2,101 packages. Out of this, we exclude 407 unlabeled packages, 49 web platform attacks, 4 denial of service attacks, 2 malicious `npm test` commands, 2 unpublished packages, 1 that is not itself malicious, and 1 that is not designed for Node.js. This leaves 1,594 packages labeled as install-time and 41 packages labeled as runtime attacks.

| Package | SBOM | CBOM | ndg [45] |
|---|---|---|---|
| conventional-changelog | ○ | ● | ● |
| event-stream | ○ | ● | ● |
| node-ipc | ○ | ○ | ○ |
| rate-map | ● | ● | ● |

**Table 2: Overview of using NodeShield and ndg with malicious *updates*. A ● means it is prevented using that enforcement alone, ○ means it is not.**

First, we consider the 41 packages with a runtime attack. These can be split into two categories, malicious *updates* (n=4) and malicious *packages* (n=37). For this experiment we create a program that imports (and uses, if necessary) the malicious package. For malicious *updates* we run the program with NodeShield using an SBOM generated for the program using the malicious package version and a statically inferred CBOM (Section 4.4) for the program using the previous benign package version (extended with inferred capabilities for new transitive dependencies of the malicious version). Similarly, we test ndg with an inferred policy for the benign version updated to include new dependencies from the malicious version. The results are summarized in Table 2.

For malicious *packages* we run the program with NodeShield using an SBOM generated for the program with the malicious package version and using 1) a statically inferred CBOM for the program using the malicious package version, and separately 2) a CBOM that grants the malicious package no capabilities. For ndg we run with the inferred policy only, assuming the results of test 2 apply to ndg too. The results are summarized in Table 3.

Second, we consider 5 packages with an install-time attack. We restrict ourselves to 5 out of 1,594 samples, matching coverage of related work [31, 45], because the evaluation requires manual setup to run the scripts explicitly (as opposed to implicitly through npm hooks). Further, we note that installation scripts are not limited to JavaScript code and we advocate instead for mechanisms along the lines of Latch [63] or LavaMoat's [7] `@lavamoat/allow-scripts` tool which provide more comprehensive defenses against install-time attacks.

All 5 packages are malicious *updates*. For this experiment we create a program that runs the install time script with NodeShield using an SBOM generated for the program using the malicious package version and using 1) a statically inferred CBOM for the program using the previous benign package version, and separately 2) a statically inferred CBOM for the program using the previous benign package version where the package is stripped of all non-install-time code. For ndg we run the same two experiments with inferred policies. The results are summarized in Table 4.

**Results** We evaluate NodeShield against 67 malware samples (46 malicious packages). The results show that our approach works best for malicious *updates* and can work for malicious *packages* provided developers review the capabilities before adopting a new dependency. We find that SBOM enforcement by itself is rarely sufficient to prevent known attacks, but is occasionally necessary (e.g., `fast-requests`). The results confirm the benefits of the CBOM in addition to the SBOM. In response to RQ1, we find that NodeShield could have prevented 98.51% (66/67) of evaluated known supply chain attacks. In contrast, ndg [45] could have prevented 83.58%

| Package | SBOM | CBOM | ndg [45] |
|---|---|---|---|
| @roku-web-core/ajax (2x) | ○ | ◐ | ◐ |
| alicon | ○ | ◐ | ◐ |
| bb-builder | ○ | ◐ | ◐ |
| bitcionjslib (2x) | ○ | ◐ | ✗ |
| bitcoisnj-lib (2x) | ○ | ◐ | ✗ |
| botbait (3x) | ○ | ◐ | ◐ |
| chalc | ○ | ◐ | ◐ |
| colorsss (2x) | ○ | ◐ | ◐ |
| colors_express | ○ | ● | ● |
| commender | ○ | ◐ | ◐ |
| component-emiter | ○ | ◐ | ◐ |
| discord-fix | ● | ◐ | ◐ |
| discord-lofy | ○ | ◐ | ◐ |
| discord-selfbot-v14 | ○ | ◐ | ◐ |
| discord-vilao | ● | ● | ◐ |
| discord.js-user | ○ | ◐ | ◐ |
| discordi.js (7x) | ○ | ◐ | ◐ |
| discordsystem | ○ | ◐ | ◐ |
| electron-native-notify | ○ | ● | ● |
| esprime | ● | ◐ | ✗ |
| express-cookies | ○ | ● | ✗ |
| fast-requests | ● | ○ | ○ |
| flatmap-stream | ○ | ● | ● |
| getcookies | ○ | ● | ✗ |
| headcache | ○ | ◐ | ◐ |
| http-proxy-middelware | ○ | ◐ | ◐ |
| ikst | ○ | ◐ | ◐ |
| jquerry | ● | ◐ | ◐ |
| leetlog (2x) | ○ | ◐ | ◐ |
| mendiff | ○ | ◐ | ◐ |
| momnet (4x) | ○ | ◐ | ◐ |
| monent | ○ | ◐ | ◐ |
| npmpubman | ○ | ◐ | ◐ |
| prerequests-xcode | ○ | ◐ | ◐ |
| random-vo...generator (4x) | ○ | ◐ | ◐ |
| seemver | ○ | ◐ | ◐ |
| stautses | ○ | ◐ | ◐ |

**Table 3: Overview of using NodeShield and ndg with malicious *packages*. ● means it is prevented using that enforcement alone, ○ means it is not. For *CBOM*, ◐ means it can be prevented if the capabilities for the package are restricted w.r.t. those inferred. A ✗ means there is a compatibility issue.**

(56/67) attacks. The fact that 9 attack are incompatible with ndg motivates the comprehensive and runtime-agnostic approach of NodeShield.

The `node-ipc` attack is not prevented by NodeShield because the preceding benign version already uses the capabilities (`file-system` and `network`) used in the attack. Compared to related work [31, 45], our evaluation covers a strict superset of in-scope (i.e., excluding availability attacks) malicious packages. Furthermore, the permission system of Ferreira et al. [31] only covers the `network`, `file-system`, and `command` capability[3], while NodeShield demonstrates the need for including the capabilities `system` and `crypto`.

[3]Unfortunately, the prototype of Ferreira et al. is no longer supported, as confirmed by personal communication, hence an empirical comparison is not possible.

| Package | SBOM | CBOM | ndg [45] |
|---|---|---|---|
| eslint-config-eslint | ○ | ● | ● |
| eslint-scope | ○ | ● | ● |
| kraken-api | ○ | ● | ● |
| mariadb | ○ | ◐ | ✗ |
| opencv.js | ○ | ● | ✗ |

**Table 4: Overview of using NodeShield and ndg with malicious *install-time* package. ● means it is prevented using that enforcement alone, ○ means it is not. For *CBOM*, ◐ means the attack can be prevented if the installation script has separate capabilities. A ✗ means there is a compatibility issue.**

In particular, 10 samples only use one or both of these internally while relying on third-party modules for further capabilities.

## 5.2 RQ2: Attack Surface Reduction

To evaluate the reduction of the attack surface we run known vulnerability exploits on NodeShield, as well as ndg [45] for comparison, to see if the exploit is caught. For this evaluation we use SecBench.js [18] (at commit `bc31562`), which provides proof of concept (PoC) exploits for code injection, command injection, path traversal, prototype pollution, and ReDoS vulnerabilities.

For this experiment, only code injection vulnerabilities are within the scope of NodeShield. In particular, command injection is subject only to the `command` capability, path traversal is subject only to the `file-system` capability, and prototype pollution and ReDoS are out of scope. On the other hand, code injection enables the attacker to leverage further capabilities.

Hence, we take the PoC code injection exploits from SecBench.js and put each in a separate project that has a dependency on the vulnerable version of the respective dependency. For NodeShield, we use a generated SBOM and statically inferred CBOM (Section 4.4, extended with the `code` and `system` capability for the vulnerable package, if not inferred). Similarly, for ndg we use its inferred policy.

We exclude some cases from our evaluation because: 16 are listed but have no PoC exploit, 12 cause prototype pollution (which is outside our threat model), 3 are mislabeled (as `code-injection` instead of `command-injection`), and 1 has an invalid PoC exploit. During the experiment, the programs are run and we observe if the exploits are caught as a violation by NodeShield and ndg. The results are summarized in Table 5.

**Results** In response to RQ2, we find that NodeShield is effective at reducing the impact of arbitrary code execution vulnerabilities in JavaScript. In particular, 87.50% (21/24) of the exploits were detected by NodeShield. This demonstrates that the reduced attack surface as a result of capability enforcement is effective at protecting against the exploitation of code injection vulnerabilities. In contrast, 12/16 (75.00%) of exploit were detected by ndg [45] with 8 of the packages being incompatible.

All attacks not prevented by NodeShield leverage the capabilities needed by the respective packages. ndg prevents two attacks NodeShield does not because these access `require`, which it disallows. However, sandbox breakouts can be used to bypass this protection (demonstrated by the `jsen` and `mongo-parse` attacks).

Moreover, this evaluation highlights the importance of separating the presented view from the enforced view (Section 4.4). For example, the exploit for the `mobile-icon-resizer` package uses

| Package | Capability | NodeShield | ndg [45] |
|---|---|---|---|
| access-policy | file-system | ● | ● |
| cd-messenger | file-system | ● | ● |
| hot-formula-parser | command | ● | ● |
| jsen | command | ● | ○ |
| json-ptr | command | ● | ○ |
| kmc | file-system | ○ | ● |
| m-log | file-system | ● | ✗ |
| mathjs (2x) | command | ● | ✗ |
| mixin-pro | file-system | ● | ● |
| mobile-icon-resizer | file-system | ● | ✗ |
| modjs | file-system | ○ | ✗ |
| modulify | file-system | ○ | ● |
| mol-proto | file-system | ● | ● |
| mongo-parse | command | ● | ○ |
| mongoosemask | file-system | ● | ● |
| node-extend | file-system | ● | ● |
| node-rules | file-system | ● | ✗ |
| node-serialize | file-system | ● | ● |
| pixl-class | file-system | ● | ● |
| reduce-css-calc | file-system | ● | ● |
| serialize-to-js | command | ● | ○ |
| thenify | file-system | ● | ✗ |
| underscore | file-system | ● | ✗ |

**Table 5: Overview of the protection of NodeShield and ndg [45] against arbitrary code execution vulnerability exploits from SecBench.js [18]. The *capability* column specifies which capability the exploit uses. A ● means the PoC exploit is stopped, ○ means it is not. A ✗ means there is a compatibility issue.**

| Index | NodeShield | ndg [45] |
|---|---|---|
| Figure 1 (CVE-2021-23449) | ● | ○ |
| Figure 5 | ○ | ○ |
| Figure 6 | ● | ● |
| Figure 7 | ● | ● |
| Figure 8 | ○ | ○ |
| vm2 issue #138 | ● | ○ |
| vm2 issue #175 | ● | ● |
| vm2 issue #177 | ● | ● |
| vm2 issue #179 | ● | ● |
| vm2 issue #184 | ● | ○ |
| vm2 issue #185 | ● | ● |
| vm2 issue #186 | ● | ● |
| vm2 issue #187 | ● | ● |
| vm2 issue #197 | ● | ○ |
| vm2 issue #199 | ● | ○ |
| vm2 issue #224 | ● | ● |
| vm2 issue #225 | ● | ● |
| vm2 issue #241 | ● | ● |
| vm2 issue #268 | ● | ● |
| vm2 issue #276 | ● | ○ |
| safe-eval issue #5 | ● | ○ |
| safe-eval issue #16 | ● | ○ |
| safe-eval issue #18 | ● | ○ |
| safe-eval issue #19 | ● | ● |
| safe-eval issue #24 (1) | ● | ● |
| safe-eval issue #24 (2) | ● | ● |
| Michał Bentkowski (1) | ● | ● |
| Michał Bentkowski (2) | ● | ● |
| Michał Bentkowski (3) | ● | ● |

**Table 6: Overview of the impact of sandbox breakouts from SandDriller [17] on NodeShield and ndg. A ● means the breakout fails and a ○ means it succeeds.**

the `file-system` capability. However, some of its transitive dependencies do need this capability. If the presented view was instead used for enforcement, `mobile-icon-resizer` would have received the capability and the exploit would not have been prevented.

## 5.3 RQ3: Robustness against Attack

To empirically evaluate the security of our enforcement strategy, as described in Section 4.3, we collect a benchmark of language-level JavaScript sandbox breakouts from SandDriller [17]. We test these on NodeShield, as well as ndg [45] for comparison. We consider snippets from figures as well as snippets referenced in Table 5 in SandDriller [17]. We create a separate program to execute each snippet and adapt them in a way that prevents the capability inference (Section 4.4) from granting the program the capability it may be trying to obtain. Some snippets were omitted because they are not applicable (e.g., they target the browser environment). In total, we create a benchmark of 27 snippets.

For NodeShield, we run the programs using an SBOM generated for the program and a statically inferred CBOM. For [45], node-dependency-guardian (ndg), we run the program under a policy that disallows everything except access to the global variables `Buffer`, `console`, `Error`, `Object`, `process`, `require`, and `setTimeout`. The results are summarized in Table 6.

**Results** In response to RQ3, we find that NodeShield can handle all sandbox breakout techniques in the benchmark except for those that only achieve prototype pollution (2/29, 6.90%), which is outside our threat model. In contrast, 11/29 (37.93%) sandbox breakouts work on ndg [45], allowing advanced malware to bypass its enforcement.

## 5.4 RQ4: Maintenance Effort

To get an indication of the maintenance required for using NodeShield we evaluate the CBOM size and frequency of CBOM changes. Our focus is on the CBOM because developers need to put in no (or little, see Section 5.6) manual effort to use NodeShield.

To evaluate size we consider CBOMs generated across all evaluations. For the evaluation of NodeShield we have generated 143 CBOMs. The average CBOM spans 78 dependencies with 64 capabilities total, or 0.82 capability per dependency. We note the relative size may be an overestimate as RQ1 and RQ2 target packages with capabilities.

To evaluate change frequency we use 6 git-based server projects, picked from the evaluations of related work [26, 61], and consider the 1,000 most recent commits. These are typically server-side Node.js project and are thus expected to accurately capture real-world capability change patterns. The experiment ignores commits for which npm dependency installation or SBOM generation fails and terminates early if there are fewer commits. The experiment

| Application | Total | Reviewable | Updated |
|---|---|---|---|
| connect | 2.42 (58) | 1.38 (33) | 0.04 (1) |
| express | 2.67 (8) | 0.67 (2) | 0.33 (1) |
| fastify | 2.22 (20) | 1.33 (12) | 0.00 (0) |
| json-server | 19.00 (285) | 8.53 (128) | 0.67 (10) |
| koa | 25.37 (1,598) | 12.81 (807) | 0.11 (7) |
| st | 3.40 (34) | 2.50 (25) | 0.50 (5) |

**Table 7: Overview of the number of capability changes (added, removed, or updated) per project. A cell reports the average per dependency-changing-commit and sum of changes across all commit (resp.).**

computes the total, "reviewable", and "updated" number of capability changes. *Reviewable* covers new capabilities of added or updated dependencies while *updated* covers only those of updated dependencies. The total is included only for completeness as removed capabilities entail no maintenance work. The results are in Table 7.

**Results** In response to RQ4, we find that projects can expect fewer capabilities than dependencies and that capability updates are infrequent. For dependency updates, application developers need to review less than 1 capability per dependency-changing commit on average. When including new dependencies, this ranges from 0.67 and 12.81 capabilities per dependency-changing commit, highlighting the difference in maintenance work between new and existing dependencies. The variance suggests that careful selection of dependencies can reduce the review workload. In conclusion, maintaining a CBOM for an application can help protect against supply chain attacks at a low overhead. This is in line with findings of related work [31, 45].

## 5.5 RQ5: Performance and Compatibility

To estimate the cost of adopting NodeShield for protecting applications, we evaluate the performance overhead and compatibility compared to vanilla Node.js. For performance, we evaluate the performance impact in terms of the response overhead and throughput of long-lived server applications as well as the runtime overhead of short-lived CLI applications. For compatibility, we report on observed incompatibilities in NodeShield across all evaluated packages and applications as well as the false positive rate for violations.

**Performance** NodeShield's performance is measured in four experiments: the response time overhead, memory overhead, and throughput of long-lived server apps, and the runtime overhead of short-lived CLI apps. The first three scenarios target our use case, while the last, based on [31], measures startup overhead. We use inferred SBOMs and CBOMs and run NodeShield in `log` mode.

The first three experiments consider server frameworks used in the evaluations of related works [26, 61]. We instantiate server applications based on the descriptions of these papers and run these applications with Node.js and NodeShield. First, we measure the response time overhead as the difference between Node.js and NodeShield of the average of 5,000 requests. Second, we measure the memory overhead by comparing the memory usage of the server when running on Node.js against NodeShield. Third, we measure the throughput (requests per second, RPS) and compare between Node.js and NodeShield. Specifically, we start by sending $n$ parallel requests and increment if all responses have been received in less than 1 second. This is repeated until handling $n$ requests takes 1 second or more, 5 times in a row. The results are in Table 8.

For the third experiment we use short-lived CLI applications used in the evaluation of [31]. We run two separate experiments. First, the applications are run as is with Node.js and NodeShield and their runtime is compared. Second, the applications are run following the methodology of [31] with Node.js and NodeShield and their runtime is compared, providing a common baseline for comparison, in absence of their tool. The results are in Table 9.

**Results** The experiments show that NodeShield induces low runtime overhead but noticeable memory and startup overhead. We observe a response time overhead of less than 1 ms (0.31%-1.99%) and we find a throughput reduction of up to 360 requests per second (0.00%-11.84%). Memory overhead incurred by NodeShield can vary, ranging from 42.50% to 250.74%. This is due to the use of `vm`, linking it to dependency count rather than workload. At 120 dependencies, `json-server` suggests a few MBs of memory may be required for Node.js apps, acceptable for modern server hardware. The startup overhead may be up to 4× (experiment A), which is because the `vm` creation and policy enforcement primarily happen during this phase (supported by experiment B).

**Compatibility** In total, the evaluation spans 86 real-world packages and applications, covering a total of 3,443 transitive packages. Out of 86, 3 were not compatible, giving a 96.51% compatibility rate. In contrast, ndg [45] exhibits compatibility issues with 15 out 61 real-world packages and applications tested, giving a 75.41% compatibility rate. For NodeShield, one incompatibility is due to the use of an undocumented API (`module._compile`) and two due to the use of `instanceof Array`.

**Results** We find that NodeShield is broadly compatible with software written for Node.js. While incompatible coding patterns are used in practice, they are infrequent and generally easy to overcome. We provide a more in-depth discussion about incompatibility in Section 5.6.

**False positive rate** We evaluate the false positive rate of violations—i.e., how often developers can expect violations that are actually benign—by running the (assumed) benign applications of the performance evaluation. As ground truth we use the set of accesses requested at runtime. A false positive is any violation that occurs (e.g., there is a capability missing from the CBOM). Dually, a true negative is the lack of a violation when there should not be (e.g., a capability from the CBOM is being used).

For this evaluation we use generated SBOMs and statically inferred CBOMs. To get the false positives count we run the apps—for servers sending one request—and count violations reported. Repeated violations by the same dependency are ignored. For true negatives we use an empty CBOM, count the violations reported by NodeShield, and subtract the number of false positives. Thus, true negatives (CBOM violation) are an underestimate w.r.t. to false positives (all violations), leading to a lower bound on the false positive rate.

The results of this experiment depend on the code covered when running the applications. To give a sense of the completeness of these experiments, we measure the code coverage of all applications in the evaluation. We find an average line coverage of 15.29% (min. 2.95%, max. 38.93%).

| | **Response Overhead** (ms) | | | **Throughput** (RPS) | | | **Memory Overhead** (kB) | | |
|---|---|---|---|---|---|---|---|---|---|
| **Server** | Node.js | NodeShield | Overhead | Node.js | NodeShield | Reduction | Node.js | NodeShield | Overhead |
| connect | 5.267 | 5.305 | 0.72% | 8,310 | 8,150 | -1.93% | 219,344 | 334,512 | 52.51% |
| express | 5.726 | 5.839 | 1.99% | 3,210 | 2,830 | -11.84% | 175,584 | 363,056 | 106.77% |
| fastify | 5.364 | 5.387 | 0.44% | 7,820 | 7,610 | -2.69% | 257,168 | 715,232 | 178.11% |
| json-server | 8.208 | 8.314 | 1.29% | 3,410 | 3,050 | -10.56% | 175,104 | 614,160 | 250.74% |
| koa | 5.217 | 5.234 | 0.31% | 8,110 | 7,830 | -3.45% | 221,696 | 375,184 | 69.23% |
| st | 5.392 | 5.464 | 1.34% | 4,380 | 4,380 | -0.00% | 217,440 | 309,856 | 42.50% |

**Table 8: Overview of long-lived server application performance test results. Response overhead is the response-time overhead from the perspective of the client. Throughput is the number of concurrent requests the server can handle in 1 second. Memory overhead is the additional memory used by NodeShield.**

| | **Experiment A** (ms) | | | | **Experiment B** (ms) | | | |
|---|---|---|---|---|---|---|---|---|
| **Application** | Node.js | NodeShield | Overhead | | Node.js | NodeShield | Overhead | |
| d3-dsv | 114.62 | 212.16 | 97.54 | 85.10% | 5,052.22 | 5,126.35 | 74.13 | 1.47% |
| docco | 898.86 | 1,201.85 | 302.89 | 33.69% | 5,043.12 | 5,097.70 | 54.58 | 1.09% |
| dot-object | 39.90 | 120.19 | 80.29 | 201.25% | 5,047.07 | 5,126.09 | 79.02 | 1.57% |
| dox | 52.50 | 235.45 | 182.95 | 348.48% | 5,060.85 | 5.239.37 | 178.53 | 3.53% |
| findup | 37.09 | 88.83 | 51.74 | 139.49% | 5,041.12 | 5,093.43 | 52.31 | 1.04% |
| html-minifier | 105.51 | 446.45 | 340.94 | 323.13% | 5,100.42 | 5,443.12 | 342.70 | 6.72% |
| js-cfb | 73.55 | 130.16 | 56.61 | 76.96% | 5,086.08 | 5,136.81 | 50.73 | 1.00% |
| json-refs | 202.15 | 981.94 | 779.79 | 385.75% | 5,121.57 | 5,905.51 | 783.93 | 15.31% |
| json2csv | 51.36 | 123.92 | 72.57 | 141.30% | 5,042.24 | 5,119.25 | 77.00 | 1.53% |
| juice | 381.43 | 875.99 | 494.56 | 129.66% | 5,146.21 | 5,667.72 | 521.52 | 10.13% |
| metalsmith | 144.31 | 193.65 | 49.34 | 34.19% | 5,114.55 | 5,163.68 | 49.14 | 0.96% |
| mocha | 140.42 | 193.78 | 53.36 | 38.00% | 5,038.53 | 5,085.31 | 46.78 | 0.92% |
| mockjs | 47.96 | 100.86 | 52.90 | 110.31% | 1,015,73 | 3,101.05 | 2,085.32 | 205.30% |
| sails | 255.85 | 1,366.53 | 1,110.67 | 434.11% | 5,234.75 | 6,302.20 | 1,067.45 | 20.39% |
| svgicons2svgfont | 158.56 | 280.19 | 121.63 | 76.71% | 5,055.82 | 5,161.54 | 105.71 | 2.09% |
| traceur | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| uglify-js | 3,406.22 | 4,500.16 | 1,093.94 | 32.12% | 5,153.41 | 5,222.16 | 68.75 | 1.33% |
| xss | 55.83 | 127.71 | 71.88 | 128.74% | 5,043.40 | 5,113.82 | 70.42 | 1.40% |
| yaml-front-matter | 59.56 | 167.53 | 107.97 | 181.29% | 5,038.85 | 5,091.74 | 52.89 | 1.05% |

**Table 9: Overview of short-lived CLI application performance test results. In experiment A, the application is run as is. In experiment B, the application is run following the approach of [31]. For both experiments the average of 10 runs is reported. `metalsmith` and `svgicons2svgfont` were modified to use `Array.isArray` (instead of `instanceof Array`) for compatibility with NodeShield. A ✗ means there is a compatibility issue.**

**Results** Across the 24 benign applications (covering 994 transitive dependencies) we observe 387 non-violations (true negatives) and 18 violations (false positives)—7 SBOM and 11 cross-package import violations. Thus, the false positive rate ($FP \div (FP + TN)$) of NodeShield in our evaluation is 4.65%.

## 5.6 Limitations

**Language support** NodeShield support most of the JavaScript language and Node.js environment, empirically supported by the evaluation in Section 5.5. However, there are three language aspect with partial support: 1) dynamic type checking, 2) overriding certain global variables, and 3) undocumented Node.js APIs.

First, the usage of `vm` creates separate V8 contexts, each with unique built-in constructors. As a result the `instanceof` operator does not work as expected for types instantiable through syntax (e.g., arrays). This limitation can be overcome by using, e.g., `Array.isArray` instead, which is recommended practice [21].

Second, our approach prevents overriding privileged global variables (from Table 1) in packages with the corresponding capability (e.g., `fetch=42`). This is due to the local binding of those variables in those packages (per Section 4.3). We believe this is uncommon yet can be overcome by overriding as `globalThis.fetch=42`.

Third, Node.js provides some APIs that are not documented. If such APIs are not implemented in NodeShield, code using these APIs breaks. We only know about `module._compile`. Covering such APIs is challenging as their intended functionality is unknown.

**Incompleteness** Despite our systematic efforts to map all Node.js APIs to capabilities (see Table 1), we cannot prove that our mapping is complete. We cover all documented APIs and some undocumented APIs. We argue this limitation is not fundamental but rather incidental. In particular, our enforcement approach outlined in Section 4.2 supports enforcing policies on such APIs, but might be insufficiently configured.

**Capabilities** While our evaluation shows that the enhancement of SBOM with CBOM is a simple and effective abstraction, there may

be challenges. First, capability overlap, most notably with `addon` and `command`, may not be obvious and potentially lead to unexpected use of resources. Second, capability transfer between components may not always be desirable. Third, as seen in Section 5.1, when a privileged malicious packages (e.g., `node-ipc`) is compromised the attack can freely use its capabilities. We argue that many packages will not be privileged, thus NodeShield significantly reduce the attack surface. Lastly, the need for manual review of capabilities can be a limiting factor for the security gained from using NodeShield.

**SBOMs** There are known issues with the accuracy of SBOMs, especially in the form of missing components. For NodeShield, this results in legitimate imports not being allowed, thus causing usability rather than security problems. Additionally, not all SBOM generators capture the dependency hierarchy, eliminating much of the benefit of SBOM enforcement from NodeShield.

## 6 MALWARE SPOTLIGHT: COPAY WALLET

To further illustrate the practical benefits of NodeShield we apply it in a case study of the attack on Copay through `event-stream` [41]. Copay is a cryptocurrency wallet application build using Node.js and Electron. In 2018, it was targeted by a supply chain attack. A dependency of Copay, `event-stream`, was compromised to include the malicious dependency `flatmap-stream`, which contained a payload that was designed to trigger only when used by Copay.

Copay used `event-stream` as part of its development processes. It was present transitively through the use of `npm-run-all`—a utility for running multiple npm scripts in a conveniently and concisely.

```
copay > npm-run-all > ps-tree > event-stream
```

In the attack, `event-stream` imports `flatmap-stream` which in turn alters the files of a runtime dependency of Copay. This alteration causes Copay to leak account data and private keys of users with sufficient balance at runtime. If `npm-run-all` ran on NodeShield, the attack would be detected during the build of Copay and no backdoored versions would have been released. As such, we describe how this attack would have played out if this was the case.

For this purpose we use `npm-run-all` v4.1.2 (commit `ec4d56c`) which, due to version ranges in `ps-tree`, can depend on both a benign and malicious version of `event-stream`. We generate an SBOM and statically infer a CBOM for both cases. When using the benign version of `event-stream` (v3.3.4) the SBOM contains the above dependency relation and a CBOM with:

```
1 { "npm-run-all@4.1.2": ["system", "file-system", "
      command"],
2   "ps-tree@1.2.0": ["system", "command"],
3   "event-stream@3.3.4": ["system", "file-system"] }
```

The malicious version of `event-stream` introduces a new dependency on `flatmap-stream` through the relation:

```
npm-run-all > ps-tree > event-stream > flatmap-stream
```

and, consequently, the CBOM will also have an entry for it (but is otherwise unchanged):

```
1 { "flatmap-stream@0.1.1": ["system"] }
```

Hence, the difference between the two version comes down to the addition of the `flatmap-stream` package—as a dependency of `event-stream`—requiring access to the `system` capability. We run `npm-run-all` as a NodeShield-based project, once with v3.3.4 of `event-stream` and once with v3.3.6 of `event-stream`, showing full compatibility with the original project and preventing the attack (resp.). This is because at runtime `flatmap-stream` uses extra capabilities (`crypto`, `file-system`, and `network`) Interestingly, none of these are present in the CBOM from static capability inference because `flatmap-stream` is obfuscated. If the code was not obfuscated, these extra capabilities would appear in the CBOM, signaling the maintainers of `npm-run-all` about the change in the capabilities used by its dependencies. Considering the presented view of its direct dependencies, the update of `event-stream` adds the `network` and `crypto` capabilities to `ps-tree`. This is unexpected for a dependency that (by its own description) "[gets] all children of a pid", thus providing a strong indicator to review the update.

In summary, the case study shows that NodeShield would have prevented the `event-stream` incident if it was used by `npm-run-all` with no manual effort from the maintainers. Moreover, the hypothetical alternative attack (i.e., not obfuscated) would likely have been caught due to the introduction of two suspicious capabilities for an existing dependency.

## 7 RELATED WORK

We discuss closely-related works and place our contributions in the broader area of web application security. NodeShield contributes with a practical open-source system with direct focus on the supply chain of Node.js applications, while ensuring Node.js compatibility, automation, minimal overhead, and policy conciseness. To the best of our knowledge, there is no system that meets these goals.

**Permission systems** We are not the first to propose a system that enforces permissions on a dependency-granular level to protect against supply chain attacks. Prior work by Ferreira et al. [31] and concurrent (unpublished) work by Ohm et al. [45] propose similar systems. Our shared goal is to protect against supply chain-based malware through a system that is broadly compatible with Node.js.

Both Ferreira et al. and Ohm et al. opt to modify the Node.js runtime to trap on imports and the `globalThis` object. The former also uses source code rewriting to add dynamic property access checks. This hinders adoption because maintaining a security-enhanced fork is expensive, and while Node.js (and Deno [27]) has introduced a permission system of its own [8], there is still a gap between academia and practice (e.g., these permission systems are not dependency aware). In contrast, NodeShield is built without modifying Node.js thus simplifying adoption and maintenance. Neither related work supports ESModules, significantly hindering adoption for modern JavaScript applications. Lastly, neither consider an attacker that attempts bypass the enforcement, which we address through lexical scoping of sensitive global variables (see Section 4.3), thus preventing such attacks as shown in Section 5.3.

The permission system of Ferreira et al. covers a subset of our capabilities, missing the `system` and `crypto` capability we find used in practice as well as attacks utilizing undeclared dependencies such as the `fast-requests` package in Section 5.1. Ohm et al. offer a more granular permission system that gives control over all imports and global variables. The resulting policy is overly verbose according to our evaluation.

From industry, LavaMoat [7] has emerged as a prominent tool to protect against JavaScript supply chain attacks. It leverage Secure ECMAScript (SES) compartments. For Node.js, it can be used as a application framework to protect against attacks from or on third-party packages. It provides stronger security guarantees (e.g., protecting against prototype pollution) at the cost of more restrictions, requiring code to be written using a subset of JavaScript.

**Sandboxes** More broadly, various works have looked at sandboxing for Node.js. Trading of compatibility and usability—in terms of API and policy conciseness—for security. De Groef et al. [26] present NodeSentry, which uses membranes [12] for policy enforcement through a modified `require` implementation. Later Vasilakis et al. [59] introduce BreakApp, separating components using three isolation tiers (language, process, container) aiming to reduce vulnerability impact. They iterate on this with a language-level Read-Write-Execute-based permission model [60] offering fine-grained control over all object properties through rewriting-based context-rebinding. Ahmadpanah et al. [15] present a sandbox library designed for running untrusted code in Trigger-Action platforms.

The above focus primarily on isolation within the Node.js process. This leaves native extensions and subprocesses vulnerable for exploitation. Addressing these gaps, Christou et al. [20] present BinWrap as a system to sandbox native code extensions along with the JavaScript itself. Similarly, Abbadini et al. [13] present NatiSand and Cage4Deno to sandbox native code extensions and subprocesses (resp.) for JavaScript runtimes using Landlock LSM [6], eBPF [4], and seccomp [9]. Wang et al. [61] present HODOR, a unified system for enforcing least privilege of system call usage in Node.js applications, including native extensions, using seccomp.

Besides runtime, the npm ecosystem can be subject to install-time attacks. As shown in Section 5.1, NodeShield can be used if the script is written in JavaScript, yet installation scripts can be arbitrary scripts or programs. Wyss et al. [63] propose Latch to enforce a policy on any install script by leveraging AppArmor [1]. LavaMoat [7] offers tooling to manage installation scripts.

**Web** Besides server-side JavaScript, there has also been work on isolating client-side JavaScript. Early attempts, such as Caja, ADsafe, and FBJs, often relied on filtering or rewriting [39]. Terrace et al. [58] present js.js, a JavaScript based interpreter to interpret untrusted JavaScript safely from JavaScript to achieve isolation. Agten et al. [14] present JSand, an SES-based sandbox that uses membranes for cross-component object sharing. Stefan et al. [55] suggest COWL as an extension of Browser security policies with label-based mandatory access control per script. Mickens [40] present the Pivot framework for building web applications using `iframes` as isolation containers, leveraging post messages as RPC between components.

**Malware detection** Many recent works analyze or detect malicious intent in the supply chain to prevent its spread. Ohm et al. [44] analyze known malicious packages for patterns and attack vectors. Ladisa et al. [35] expand with a more extensive literature review and construct an attack tree covering 107 unique vectors. Similarly, [36] evaluate the features of package managers to uncover attack vectors for arbitrary code execution on developer machines.

Fass et al. [30] propose using multiple static analyses combined with random forest classification to detect malicious JavaScript samples. Duan et al [28] propose using metadata, static, and dynamic analysis to detect malicious packages on PyPI, npm, and RubyGems. Ntousakis et al. [43] extend the work of Vasilakis et al. [60] to detect malicious packages at runtime. Sejfia and Schäfter [48] use feature-driven machine learning to detect malicious npm packages. Li et al. [37] leverage inter-procedural source-to-sink analysis to reduce false positives in detecting malicious npm and PyPI packages. Liang et al. [38] focus on detecting malicious install scripts in PyPI packages, identifying behavioral outliers. Huang et al. [34] proposes the use of behavior sequences observed in known malicious packages to detect new malicious packages. Sofaer et al. [54] focuses on detecting malicious updates of packages based on changes in the use of external APIs, aligning in principle with our CBOM proposal.

## 8 CONCLUSION

We presented a runtime protection mechanism, NodeShield, against supply chain attacks on Node.js that is able to defend against 98.51% of tested real-world supply chain attacks and 87.50% of tested vulnerability exploits. NodeShield requires little effort from developers and incurs low overhead on long-lived applications. Driven by a novel application of lexical scoping, NodeShield can protect against more sandbox breakouts than related works. We applied NodeShield to a case study of the 2018 attack on the Copay application, showing its potential in practice. In future work NodeShield and CBOM can be applied in different settings with more granular policies, including other JavaScript runtimes, browsers, or languages.

## ACKNOWLEDGMENTS

We thank Daniel Hedin and the anonymous reviewers for their feedback. This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

## REFERENCES

[1] AppArmor. https://gitlab.com/apparmor/apparmor. Accessed: `8de2ff3`.
[2] Cackle (cargo acl). https://github.com/cackle-rs/cackle. Accessed: 6857d88.
[3] Capslock. https://github.com/google/capslock. Accessed: 93953b6.
[4] eBPF. https://ebpf.io/. Accessed: 2024-09-13.
[5] ECMAScript® 2026 specification. https://tc39.es/ecma262/. Accessed: 2025-04-03.
[6] Landlock. https://docs.kernel.org/security/landlock.html. Accessed: 2024-09-13.
[7] LavaMoat. https://github.com/LavaMoat/LavaMoat. Accessed: `a859f9f`.
[8] Permissions | Node.js v20.x documentation. https://nodejs.org/docs/latest-v20.x/api/permissions.html. Accessed: 2025-04-03.
[9] seccomp. https://github.com/seccomp. Accessed: 2024-09-13.
[10] Socket.dev alerts. https://socket.dev/alerts. Accessed: 2024-09-13.
[11] Threats, risks, and mitigations in the open source ecosystem. https://github.com/ossf/wg-metrics-and-metadata. Accessed: `45ff44b`.
[12] Isolating application sub-components with membranes. https://tvcutsem.github.io/membranes, 2018. Accessed: 2025-03-17.
[13] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. NatiSand: Native code sandboxing for JavaScript runtimes. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 639–653, 2023.
[14] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 1–10, 2012.
[15] Mohammad M Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2899–2916, 2021.
[16] Mark W. Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. Augur: Dynamic taint analysis for asynchronous javascript. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE'22, 2023.

[17] Abdullah Alhamdan and Cristian-Alexandru Staicu. SandDriller: A fully-automated approach for testing language-based JavaScript sandboxes. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3457–3474, 2023.

[18] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. SecBench.js: An executable security benchmark suite for server-side JavaScript. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1059–1070. IEEE, 2023.

[19] Darion Cassel, Wai Tuck Wong, and Limin Jia. Nodemedic: End-to-end analysis of node. js vulnerabilities with provenance graphs. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 1101–1127. IEEE, 2023.

[20] George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P Kemerlis, and Nikos Vasilakis. BinWrap: Hybrid protection against native Node.js add-ons. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, pages 429–442, 2023.

[21] MDN Contributors. Array.isArray. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray. Accessed: 2025-03-27.

[22] Node.js Contributors. Node.js v20.19.3 documentation. https://nodejs.org/docs/latest-v20.x/api/index.html. Accessed: 2025-07-14.

[23] Node.js Contributors. Usage of primordials in core. https://github.com/nodejs/node/blob/main/doc/contributing/primordials.md. Accessed: `038d829`.

[24] Eric Cornelissen and Musard Balliu. NodeShield: Runtime enforcement of security-enhanced SBOMs for Node.js - artifact. https://zenodo.org/records/16873448.

[25] Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu. Ghunter: Universal prototype pollution gadgets in javascript runtimes. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3693–3710, 2024.

[26] Willem De Groef, Fabio Massacci, and Frank Piessens. NodeSentry: Least-privilege library integration for server-side JavaScript. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 446–455, 2014.

[27] Fernando Doglio. Introducing Deno.

[28] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. 2021.

[29] ESLint. Postmortem for malicious packages published on july 12th, 2018. https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes/. Accessed: 2025-04-13.

[30] Aurore Fass, Michael Backes, and Ben Stock. Jstap: A static pre-filter for malicious javascript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 257–269, 2019.

[31] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1334–1346. IEEE, 2021.

[32] Harry Garrood. Malicious code in the purescript npm installer. https://harry.garrood.me/blog/malicious-code-in-purescript-npm-installer/. Accessed: 2025-04-02.

[33] Dan Geer, Bentz Tozer, and John Speed Meyers. For good measure: Counting broken links: A quant's view of software supply chain security. *USENIX; Login*, 45(4), 2020.

[34] Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, Zhen Yang, and Lei Shi. DONAPI: Malicious npm packages detector using behavior sequence knowledge mapping. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3765–3782, 2024.

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

[36] Piergiorgio Ladisa, Merve Sahin, Serena Elisa Ponta, Marco Rosa, Matias Martinez, and Olivier Barais. The hitchhiker's guide to malicious third-party dependencies. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 65–74, 2023.

[37] Ningke Li, Shenao Wang, Mingxi Feng, Kailong Wang, Meizhen Wang, and Haoyu Wang. Malwukong: Towards fast, accurate, and multilingual detection of malicious code poisoning in oss supply chains. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1993–2005. IEEE, 2023.

[38] Wentao Liang, Xiang Ling, Jingzheng Wu, Tianyue Luo, and Yanjun Wu. A needle is an outlier in a haystack: Hunting malicious pypi packages with code clustering. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 307–318. IEEE, 2023.

[39] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted javascript. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 77–91. IEEE, 2009.

[40] James Mickens. Pivot: Fast, synchronous mashup isolation using generator chains. In *2014 IEEE Symposium on Security and Privacy*, pages 261–275. IEEE, 2014.

[41] npm. Details about the event-stream incident. https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident. Accessed: 2025-03-27.

[42] npm. Reported malicious module: getcookies. https://blog.npmjs.org/post/173526807575/reported-malicious-module-getcookies. Accessed: 2025-04-13.

[43] Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. Detecting third-party library problems with combined program analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2429–2431, 2021.

[44] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pages 23–43. Springer, 2020.

[45] Marc Ohm, Timo Pohl, and Felix Boes. You can run but you can't hide: Runtime protection against malicious package updates for Node.js. *arXiv preprint arXiv:2305.19760*, 2023.

[46] Hamed Okhravi, Nathan Burow, and Fred B. Schneider. Software bill of materials as a proactive defense. *IEEE Security & Privacy*, 23(2):101–106, 2025.

[47] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. Explicit secrecy: A policy for taint tracking. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 15–30, 2016.

[48] Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1681–1692, 2022.

[49] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, 2013.

[50] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent spring: Prototype pollution leads to remote code execution in node.js. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 2023.

[51] Mikhail Shcherbakov, Paul Moosbrugger, and Musard Balliu. Unveiling the invisible: Detection and evaluation of prototype pollution gadgets with dynamic taint analysis. In *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*, pages 1800–1811, 2024.

[52] Snyk. electron-native-notify malicious package. https://security.snyk.io/vuln/SNYK-JS-ELECTRONNATIVENOTIFY-174928. Accessed: 2025-04-13.

[53] Snyk. rate-map malicious package. https://security.snyk.io/vuln/SNYK-JS-RATEMAP-451649. Accessed: 2025-04-02.

[54] Raphael J Sofaer, Yaniv David, Mingqing Kang, Jianjia Yu, Yinzhi Cao, Junfeng Yang, and Jason Nieh. Rogueone: Detecting rogue updates via differential dataflow analysis using trust domains. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[55] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. Protecting users by confining JavaScript with COWL. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 131–146, 2014.

[56] Marius Steffens and Ben Stock. Pmforce: Systematically analyzing postmessage handlers at scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 493–505, 2020.

[57] CNCF Security TAG. Catalog of supply chain compromises. https://github.com/cncf/tag-security/blob/06147d5/supply-chain-security/compromises.

[58] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. JavaScript in JavaScript (js.js): Sandboxing third-party scripts. In *3rd USENIX Conference on Web Application Development (WebApps 12)*, pages 95–100, 2012.

[59] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. BreakApp: Automated, flexible application compartmentalization. In *NDSS*, 2018.

[60] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 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*, pages 1821–1838, 2021.

[61] Wenya Wang, Xingwei Lin, Jingyi Wang, Wang Gao, Dawu Gu, Wei Lv, and Jiashui Wang. Hodor: Shrinking attack surface on node. js via system call limitation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2800–2814, 2023.

[62] OpenSSF Package Analysis WG. OpenSSF package analysis case studies. https://github.com/ossf/package-analysis/blob/c4af43d/docs/case_studies.md.

[63] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. Wolf at the door: Preventing install-time attacks in npm with Latch. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 1139–1153, 2022.

[64] 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 *28th USENIX Security Symposium (USENIX Security 19)*, pages 995–1010. USENIX Association, 2019.