# Defensive Capability Analysis for JavaScript Libraries

Wenyuan Xu
Aarhus University
Aarhus, Denmark
wenyuan.xu@cs.au.dk

Anders Møller
Aarhus University
Aarhus, Denmark
amoeller@cs.au.dk

## Abstract

Capability analysis is a key building block for JavaScript supply-chain security tasks such as malware detection, security auditing, and runtime policy construction. However, existing approaches do not provide guarantees in the presence of JavaScript's dynamic features.

We present a defensive capability analysis for JavaScript libraries that soundly reports every exercised capability for code executed under a lightweight protected runtime. The analysis tracks only objects that are relevant to capabilities. It reports capability usage either at the program locations where such objects are used, or conservatively when relevant objects escape through operations that the analysis does not model precisely. To address a small number of dynamic language features that would otherwise invalidate this analysis, we complement the static analysis with a lightweight runtime enforcement mechanism that blocks those patterns.

We implement the approach and evaluate it on large datasets of benign and malicious npm packages. The results show that the approach scales to real-world packages, pinpoints capability-usage locations with high accuracy, and remains compatible with 98.2% of benign libraries when using the protected runtime. Moreover, it detects capability usage in 99.8% of the malicious packages in our dataset, while the remaining cases are blocked by the runtime protection. Our study also shows that at least 72.9% of the npm packages use no security-sensitive capabilities at all, suggesting that capability analysis can substantially reduce auditing effort in practice.

## CCS Concepts

• **Security and privacy → Software security engineering**; • **Software and its engineering → Automated static analysis**; *Software libraries and repositories*; *Scripting languages.*

## Keywords

static analysis, software supply-chain security

**ACM Reference Format:**
Wenyuan Xu and Anders Møller. 2026. Defensive Capability Analysis for JavaScript Libraries. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837491


*ASE '26, Munich, Germany*

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837491

## 1 Introduction

JavaScript programs rely heavily on third-party libraries [4, 35, 39]. The npm registry now hosts more than 3 million packages, many of which are downloaded millions of times each week. In most cases, these packages are benign and provide useful functionality that helps accelerate development. However, thousands of npm packages are reported as malicious or vulnerable each year,[1] indicating that software supply-chain security has become a critical concern in the npm ecosystem.

This problem is particularly acute in Node.js. Unlike browser JavaScript, Node.js exposes native functions and objects that allow code to interact with the file system, the network, the process environment, and the module system. Examples include `fs.readFile`, `process.env`, and `module._cache`. In this paper, we refer to such security-relevant native functions and objects as *sensitive objects*. When code calls them, reads their values, or writes their properties, it gains the ability to perform security-sensitive actions such as reading files, sending data over the network, or executing code. We refer to such operations as *capabilities*.

*Capability analysis* reasons about which capabilities a package may exercise and where in the code they are used. It is central to software supply-chain security because executing malicious behavior in Node.js-based applications typically involves such capabilities. For example, most vulnerability and malware detection tools follow a common pipeline: they first extract capability-usage information from the code, then apply dataflow and control-flow analysis based on the capability usage location, and finally decide whether the information flow is malicious or vulnerable by predefined rules, supervised classification, or LLM-based techniques to [8, 15, 25, 28, 30, 36–38].

Regardless of how sophisticated the downstream analysis is, failing to identify code that uses capabilities causes the analysis to miss potentially malicious or vulnerable code. This problem is especially serious for malware detection, because malicious code is deliberately designed to evade detection tools.

Unfortunately, no existing malware detection tool guarantees that all capability uses are identified [8, 15, 25, 28, 30, 36–38]. JavaScript has many dynamic features—dynamic property reads and writes, prototype chains, and a large set of native objects—that make capability analysis challenging. To balance precision, recall, and scalability, existing tools handle these features incompletely, causing their approaches to miss capability uses. Moreover, some tools ignore dependency code altogether to achieve higher scalability [25, 28, 38].

Some approaches [16, 37] use LLMs to audit capability usage, thereby sidestepping the limitations of traditional program analysis.

[1] https://socket.dev/blog/2023-npm-retrospective and https://github.com/advisories.

However, LLMs provide no such guarantee either, and recent studies have demonstrated practical evasion techniques against such detectors [22, 32]. Moreover, these LLM-based approaches incur high analysis costs, which makes them impractical for analyzing the entire packages in npm. Another group of approaches provides runtime sandboxing and permission systems [1, 5, 12, 13, 31, 33, 34] that enforce policies specifying which capabilities to allow. However, constructing correct policies requires knowing what capabilities the code may exercise—which brings us back to the need for analysis that does not miss exercised capabilities. For these reasons, developers still cannot prevent potentially malicious code from compromising their systems without manually reviewing all dependency code.

Inspired by the defensive analysis technique by Smaragdakis and Kastrinis [29], we propose a *defensive capability analysis* for JavaScript libraries together with a lightweight runtime protection mechanism. The static analysis uses a pointer analysis that tracks only sensitive objects through assignments, property reads, and modeled native calls. When the points-to information loses track of sensitive objects at unsupported instructions, the analysis conservatively reports capability use; otherwise, it identifies the exact locations at which capabilities are exercised according to the points-to information. To handle a small number of dynamic language features that would otherwise invalidate the static analysis, we propose a lightweight runtime protection mechanism that blocks those patterns. Together, these two components guarantee that every exercised capability is reported for code executed under the protected runtime.

The analysis does not, by itself, determine whether a package is malicious or vulnerable; instead, it provides a reliable foundation that strengthens the malicious code detection and runtime protection techniques:

(1) For manual auditing, packages that do not use capabilities can be excluded from further review, whereas, for packages that do use capabilities, the analysis pinpoints the relevant locations and helps auditors focus on the corresponding code.
(2) For more expensive downstream detectors, such as LLM-based approaches, capability analysis can serve as a lightweight filtering step that forwards only potentially security-relevant packages.
(3) For runtime protection mechanisms, the inferred capability information provides a principled basis for reviewing or constructing capability policies.

We implement the approach and evaluate it on large datasets of benign and malicious npm packages. The results show that our approach can analyze 99.0% of the packages within 20 seconds, even including their dependencies. The approach also allows 72.9% of the benign packages to be excluded from further review because they do not exercise any capabilities, and for a collection of malicious packages it detects 99.8% with static analysis alone, while blocking the remaining cases with the runtime protection.

In summary, this paper makes the following contributions:

- We present a defensive capability analysis for JavaScript libraries that guarantees every exercised capability is reported for executed code under a lightweight protected runtime.
- We show that the approach is practical on a large npm dataset, with high precision, high location accuracy, and low analysis cost.
- We provide an empirical study of capability usage in benign npm packages, showing that many packages use no security-sensitive capabilities and characterizing how the remaining packages use them.
- We study capability usage in real-world malicious npm packages and show that our approach can capture all of their capability usages.

## 2 Motivation

Malware detection relies on capability analysis because malicious code ultimately requires such capabilities to achieve its objectives. Existing approaches are often effective when code directly imports and invokes sensitive APIs. However, they become unreliable when capability usage is mediated through dependencies, dynamic JavaScript features, or insufficient native objects modeling.

Figure 1 shows an example. Figure 1a is a code fragment from *@ungap/structured-clone*, which has many dependents on npm.[2] At line 2, the code defines a variable `env` that may resolve to the `global` object. The `deserialize` function, defined at line 3, reconstructs JavaScript objects from serialized data. At lines 15 and 18, the function performs dynamic property reads on `env`. In JavaScript, the `global` object exposes built-in objects, including the `Function` constructor, which can construct new functions from string arguments. Consequently, this code introduces a potential vulnerability that allows the creation of arbitrary functions, although the package itself is not malicious.

An attacker-controlled package such as Figure 1b, however, can exploit this vulnerability to achieve arbitrary code execution. Unlike a typical malicious package that directly invokes the `Function` constructor to create and execute malicious code, the malicious package instead places a crafted serialized string in the `"signature"` field of `package.json` and passes it to the deserialization function provided by *@ungap/structured-clone* (line 26). This serialized string causes the code at line 15 to construct a malicious function instance. As shown by the serialized string (line 27), this function calls `global.process.getBuiltinModule('fs').rm('/')`, which loads the `fs` module and invokes `fs.rm('/')` to delete all files in the root directory. The code later requires *@jest/diff-sequences* (line 28) to obtain the `diffSequenceModule` function. We omit the implementation of `diffSequenceModule` here. The key point is that its third parameter is a callback that is executed when the elements of `a` and `b` differ. By passing the malicious function at line 31 to `diffSequenceModule`, the malicious package ultimately triggers a call to `fs.rm('/')`.

When existing state-of-the-art tools based on traditional static analysis [8, 15, 25, 28, 30, 36–38] analyze the code in Figure 1b, the first challenge arises from dependency handling. Most malware-detection tools ignore dependencies for precision and scalability [25, 28, 38]. As a result, they may miss the code-execution capability entirely because it is defined in *@ungap/structured-clone*.

The second challenge is the lack of accurate modeling of JavaScript features and native functions. For example, existing tools typically

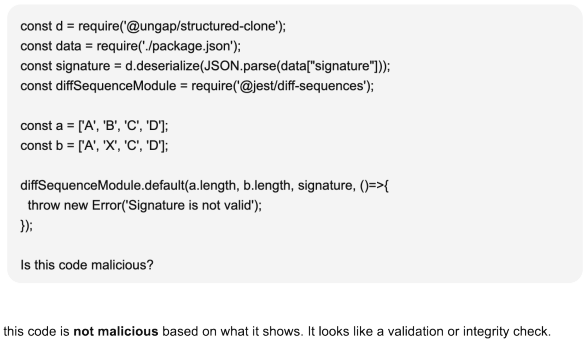
[2] https://www.npmjs.com/package/@ungap/structured-clone

```
const { VOID, PRIMITIVE, /*...*/ } = require('./types.js');
const env = typeof self === 'object' ? self : global;
const deserializer = ($,_) => {
  const as = (out, index) => {
    $.set(index, out);
    return out;
  };
  const unpair = index => {
    const [type, value] = _[index];
    switch (type) {
      case PRIMITIVE: // ...
      case VOID: // ...
      case ERROR: {
        const {name, message} = value;
        return as(new env[name](message), index);
      }
    }
    return as(new env[type](value), index);
  };
  return unpair;
};
const deserialize = serialized => deserializer(new Map,
      serialized)(0);
exports.deserialize = deserialize;
```

**(a) Code snippet from *@ungap/structured-clone* has a potential code execution vulnerability.**

```
const d = require('@ungap/structured-clone');
const data = require('./package.json');
const signature = d.deserialize(JSON.parse(data["signature"])
      );
// the signature in package.json is "[[7,{\"name\":\"
      Function\",\"message\":\"global.process.
      getBuiltinModule('fs').rm('/')\"}]]";
const diffSequenceModule = require('@jest/diff-sequences');
const a = ['A', 'B', 'C', 'D'];
const b = ['A', 'X', 'C', 'D'];
diffSequenceModule.default(a.length, b.length, signature, ()
      =>{
  throw new Error('Signature is not valid');
});
```

**(b) An evil package leverages the potential vulnerability in *@ungap/structured-clone* to execute arbitrary code. It also uses the *@jest/diff-sequences* package as a gadget to call the deserialized function.**




**(c) GPT-5.2(Auto) misses the malicious package.**

**Figure 1: Cross-package capability abuse example.**

do not handle the dynamic property reads used at ine 15 and line 18 in a way that guarantees complete capability coverage [17]. Modeling native objects is also difficult and frequently incomplete. Some tools [8] even ignore that the global object contains properties corresponding to built-in objects.

In recent years, AI-based approaches have shown great promise for malware detection. However, such approaches cannot guarantee that they identify all capability uses. For example, when we provide the JavaScript code to GPT-5.2, it responds as shown in Figure 1c, incorrectly claiming that the code exhibits no malicious behavior. Notably, we make the code short for simplicity of the example, which should make the task easier for the LLM. Even under this favorable setup, the model is still confused. Although an LLM might detect the behavior if we also provide the malicious `package.json` file, repeat the query multiple times (because the model has inherent randomness in each response), or use a more clever model than GPT-5.2, real-world attackers can conceal the payload in many other ways. Exploring such evasions is beyond the scope of this paper. The key point is that instead of obfuscating the malicious code for bypassing the traditional static analysis, attackers can leverage benign packages to acquire capabilities that camouflage themselves as benign packages, for bypassing the LLM-based detection. There are other advanced techniques to bypass the AI-based code analysis, which will be discussed in Section 7.

In summary, we seek a capability analysis that satisfies the following requirements: (R1) It must be sound, which means that in any possible execution, if a program may use a capability, the analysis must report that capability; (R2) It must be precise to avoid overwhelming users with false positives. (R3) It must be scalable to handle large codebases (including their dependencies);

In addition to R1–R3, security auditing also requires that (R4) the analysis can accurately report the location where a capability is exercised. Specifically, we expect the analysis to report each capability usage as a tuple $\langle l, c, d \rangle$, where $l$ denotes a code location, $c$ denotes a capability, and $d \in \{direct, indirect\}$ indicates whether the capability is exercised at location $l$ itself or may be exercised after the execution of code at location $l$. Under this definition, for each capability usage reported by the analysis, we prefer direct reports ($d = direct$) over indirect reports ($d = indirect$). For example, in Figure 1, the analysis could report $\langle line\ 2, CodeExec, indirect \rangle$ (*report1*), or it could report $\langle line\ 15, CodeExec, direct \rangle$ and $\langle line\ 18, CodeExec, direct \rangle$ (*report2*). We prefer *report2* because it shows the capability usage more explicitly.

In the subsequent sections, we present an approach that satisfies these four requirements. For this specific example, it generates *report2* as result.

## 3 Modeling Capabilities

Before describing the capability analysis, we define which operations on native objects correspond to which capabilities.

By reviewing prior work together with the ECMAScript and Node.js documentations,[3] we summarized the capabilities with their corresponding operation patterns in Table 1. In the following sections, we represent this mapping as a *capability mapping* $CM \subseteq P \times C$, where $P$ is the set of operations on sensitive objects and $C$ is the

[3] ECMAScript 2026 and Node.js v25.8.0.

**Table 1: Capability categories, capabilities, and a representative subset of the operations on sensitive objects.**

| Category | Capability | Operations |
|---|---|---|
| File Operation | *FileRead* | `call <fs>.readFile`, `call <fs>.read` |
| | *FileWrite* | `call <fs>.writeFile`, `call <fs>.write` |
| | *FileOther* | `call <fs>.access`, `call <fs>.exists` |
| Network Operation | *NetInfo* | `call <dns>.lookup`, `call <net>.connect` |
| | *NetIO* | `call <http>.get`, `call <https>.get` |
| Process Execution | *ProcExec* | `call <child_process>.exec`, `call process.execve` |
| Code Execution | *CodeStatic* | `call eval(1:StringLiteral)`, `call Function(1:StringLiteral)` |
| | *CodeDyn* | `call eval(1:!StringLiteral)`, `call Function(1:!StringLiteral)`, `call WebAssembly.compile` |
| Dynamic Require Modules | *DynRequire* | `call require(1:!Literal)`, `call <process>.getBuiltinModule`, `write <module>._cache.*` |
| Information | *Info* | `read process.env.SENSITIVE`, `call <os>.userInfo` |

set of capabilities. For example, $\langle$`call <fs>.readFile`, *FileRead*$\rangle \in$ *CM*, means that calling `<fs>.readFile` exercises the *FileRead* capability.

We use the pattern language by Møller et al. [24] to describe the operations on sensitive objects. Each pattern consists of an operation (`read`, `write`, and `call`) together with an access path that denotes the target object. For example, `<fs>.readFile` denotes the `readFile` function defined in export object of module `fs`. For call patterns, we optionally attach an argument filter of the form `(n:type)` to constrain the *n*-th argument's AST type. For object properties, we use `*` to denote all properties of the object, and `SENSITIVE` refers to a predefined set of sensitive properties, such as `PATH`, `GITHUB_TOKEN`, etc.

Most capability names are self-explanatory. For network operations and code execution, we further divide operations into finer-grained capabilities because they have different security implications. In network operations, *NetInfo* denotes the capability to establish or expose network connectivity that can reveal host information or enable side-channel behavior, whereas *NetIO* denotes the capability to directly transmit data over the network. In code execution, *CodeStatic* denotes the capability to execute dynamic code but the code is a static string. The remaining cases, classified as *CodeDyn*, involve dynamically computed arguments and are considered more dangerous. *DynRequire* represents cases where code requires a module whose name is determined at runtime. Although it does not directly exercise a concrete capability such as file or network access, it may load unanalyzed code, making it effectively similar to dynamic code execution.

## 4 Defensive Capability Analysis

The key idea of our approach is to use pointer analysis to track capability-relevant objects and identify program locations where capabilities are exercised. When the points-to information can no longer track such objects precisely, the analysis conservatively reports indirect capability usage.

The analysis of a given program is defined by a collection of constraints over points-to sets denoted $[\![\cdot]\!]$. A *program variable* $[\![x]\!]$ denotes the abstract values of a variable $x$ defined in the program, and an *expression variable* $[\![e]\!]$ similarly represents other kinds of syntactic expressions $e$. A *property variable* $[\![o.p]\!]$ represents the values stored in property $p$ of an abstract heap object $o$, and a *return variable* $[\![f_{return}]\!]$ denotes the return values of function $f$.

*Preliminary step.* As a preliminary step, we model native objects. For all sensitive objects used in access patterns, we create their corresponding abstract heap objects, such as $o_{\texttt{require}}$ representing the `require` object and $o_{\texttt{eval}}$ representing the `eval` object. For sensitive objects that are available through predeclared identifiers, we introduce the corresponding constraint variables and initialize their points-to sets, such as $o_{\texttt{require}} \in [\![\texttt{require}]\!]$ and $o_{\texttt{eval}} \in [\![\texttt{eval}]\!]$.

A sensitive object may be obtained through other native objects; we call such native objects *carrier objects*. For example, `global` is a carrier object because the sensitive object `eval` can be obtained by reading `global.eval`. This notion is transitive: a native object that exposes a carrier object is itself a carrier object. For the remainder of this section, we collectively refer to sensitive objects and carrier objects as *relevant objects*.

For carrier objects, we first prepare the corresponding abstract heap objects and constraint variables, as we do for sensitive objects. A carrier object that is the export object of a native module is denoted by $export_{id}$, where *id* is the module identifier. Carrier objects may expose relevant objects either through property accesses or through the return values of function invocations. For property accesses, we add a property constraint $o \in [\![o'.p]\!]$, where $o$ and $o'$ are relevant objects and property $p$ of $o'$ evaluates to $o$. Taking `eval` and `global` as an example, we add $o_{\texttt{eval}} \in [\![o_{\texttt{global}}.\texttt{eval}]\!]$. Property constraints also account for prototype-chain lookup. That is, if $o \in [\![o'.p]\!]$ and $o'$ is the prototype of $o''$, then we add $o \in [\![o''.p]\!]$. For each property constraint, we record the modeled property in the property set of the base object, which will be used when the analysis needs to enumerate modeled properties, i.e., $o \in [\![o'.p]\!] \Rightarrow p \in Props(o')$. Furthermore, for any carrier function object $f$ that returns a relevant object $o$, we add a return constraint $o \in [\![f_{return}]\!]$.

When the analysis can no longer track a carrier object precisely, it must conservatively report every capability reachable through it. We introduce a carrier mapping $KM \subseteq O \times C$ for this purpose, where $O$ is the set of carrier objects and $C$ is the set of capabilities. *KM* is constructed from the property and return constraints described above. We write $exposes(o', o)$ to mean that $o'$ exposes $o$ through either a property access or a return value: $exposes(o', o) \equiv o \in [\![o'.p]\!] \vee o \in [\![o'_{return}]\!]$. Then *KM* is the least set closed under the following two rules: $exposes(o', o) \wedge (op\, o, c) \in CM \Rightarrow (o', c) \in KM$ and $exposes(o', o) \wedge (o, c) \in KM \Rightarrow (o', c) \in KM$ where $op \in \{\texttt{call}, \texttt{read}, \texttt{write}\}$ and *CM* is the capability mapping defined in Section 3. The first rule associates carrier objects with capabilities based on the capabilities of the sensitive objects, whereas the second rule propagates such associations transitively.

Figure 2 presents representative rules of the main analysis, which consists of three parts: basic pointer-analysis rules (prefixed with BASIC), sensitive-operation rules (prefixed with SENSITIVE), and escaping rules (prefixed with ESC).

*Pointer analysis rules.* These rules propagate points-to information through the program. Because JavaScript's dynamic features

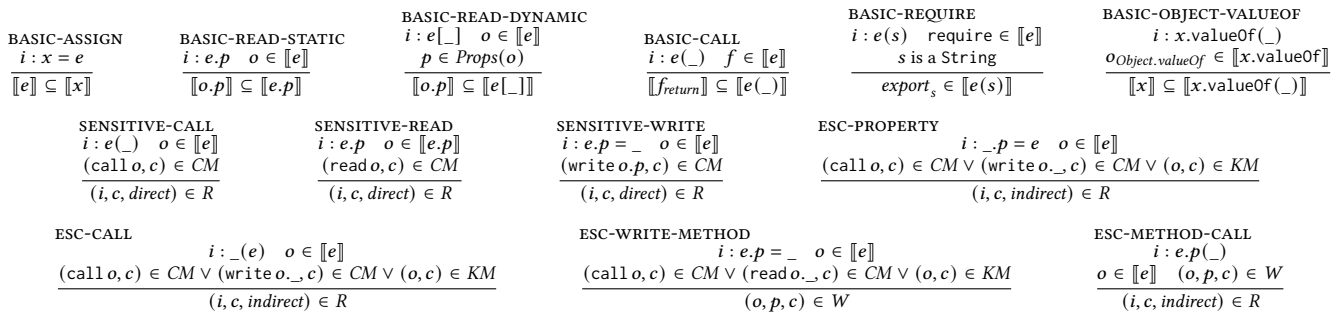



**Figure 2: the representative rules for defensive capability analysis, where $i : \ldots$ represents the instruction; $R$ is the set of reported capability usages.**

make fully sound and scalable tracking of complete points-to information extremely difficult, these rules only care about relevant objects. Moreover, the analysis supports only assignments, property reads, and calls whose callees resolve to modeled native functions. Including additional operations would substantially increase the complexity of the analysis. For example, in a property write such as `x.p = eval`, determining which property variables should receive `eval` requires computing the full points-to set of `x`, not only its relevant objects. This requirement conflicts with our design choice to track only relevant objects. We explain below how property reads can nevertheless be handled.

Rules BASIC-ASSIGN and BASIC-READ-STATIC are classical Andersen-style propagation rules for assignments and static property reads [2]. For property reads $e.p$ to soundly propagate relevant objects it suffices to know the relevant objects of the base expression $e$, unlike the situation for property write operations.

Rule BASIC-READ-DYNAMIC handles dynamic property accesses by conservatively reading all modeled properties of the base object.

Rule BASIC-CALL models function calls. Because the analysis tracks only relevant objects, callee expressions can resolve only to native functions introduced in the preliminary step, not to user-defined functions. Since the preliminary step models only return values of native functions and does not model argument-to-parameter dataflow, this rule propagates only the return value.

Rule BASIC-REQUIRE models the `require` function. When the program calls `require` with a string literal, the analysis resolves the target module and propagates the corresponding exported object to the call result expression variable. This special handling is necessary because programs use `require` to load modules that export sensitive objects, such as *fs* and *http*.

Rule BASIC-OBJECT-VALUEOF models the `Object.prototype.valueOf` function. We will discuss the motivation for this rule in the context of escaping rules.

*Sensitive operation rules.* These rules report direct capability usage based on the AST together with the computed points-to information. For example, rule SENSITIVE-CALL matches a call expression and checks whether the callee may point to a sensitive object whose corresponding call operation is recorded in *CM*. If so, it reports a direct capability usage. Rules SENSITIVE-READ and SENSITIVE-WRITE are analogous for property reads and writes, respectively.

*Escaping rules.* These rules handle data flows that the basic rules do not track. Along such flows, the BASIC rules cannot determine where relevant objects may flow. To preserve the guarantee that every exercised capability is reported, without explicitly tracking those flows, the escaping rules report *indirect* capability usages.

Rule ESC-PROPERTY handles property writes. Since the pointer analysis rules ignore property writes, as discussed above, the rule conservatively emits an *indirect* report if the value being written may point to a relevant object.

Rule ESC-CALL handles call instructions whose argument may contain a relevant object. Similar to the discussion of property write instructions, the analysis cannot determine where such an argument may flow after the call. Therefore, if an argument may point to a relevant object, the rule emits an *indirect* report. The dataflow from receiver object to `this` in method calls is not handled by this rule; we discuss it separately at the end of this section.

Other similar cases, including return statements, thrown exceptions, and export statements, are handled in the same way: the rules check whether the source expression may point to a relevant object and, if so, emit an *indirect* report.

The dataflow from base object to `this` in method calls requires a different treatment from ordinary arguments. Otherwise the rule become too imprecise. For example, `global` is a carrier object, but a call such as `global.isNaN()` should not trigger an *indirect* report merely because the receiver flows to `this`. We handle receiver-to-`this` dataflow by classifying method calls along two dimensions: whether the invoked method is native or user-defined, and whether it is defined directly on the relevant object or reached through its prototype chain. This classification gives four cases, which we handle separately:

(1) **Native methods defined on relevant objects.** In this case, we treat the method itself is a sensitive object and record it in *CM*.
(2) **User-defined methods written to properties of relevant objects.** For example, in `global.f=function(){this.eval(...)}; global.f()`, the call eventually invokes `eval`. Rules ESC-WRITE-METHOD and ESC-METHOD-CALL handle this case. Rule ESC-WRITE-METHOD matches the property write and records triples $W$ consisting of the base object, the defined property name, and the capability associated with the base object. Rule

ESC-METHOD-CALL then reports an *indirect* usage when a later call may invoke the defined method.

(3) **Native methods defined in prototype objects of relevant objects.** In this case, we only need to model native methods whose implementations may either exercise capabilities through `this` or propagate the receiver object to another constraint variable that remains accessible after the call. By inspecting the methods defined on the prototype objects of all relevant objects, we identify four native methods that require special handling: `Object.prototype.valueOf`, `Function.prototype.apply`, `Function.prototype.call`, and `Function.prototype.bind`. We add dedicated rules for these four native methods. For example, malicious code could call `eval.valueOf()`, which invokes `Object.prototype.valueOf` and returns the `eval` function itself. Rule BASIC-OBJECT-VALUEOF models this behavior by propagating the receiver into the points-to set of the call expression when the receiver is a relevant object. To enable the rule to handle such calls, the preliminary step has the property constraint $o_{\texttt{Object.valueOf}} \in [\![o.\texttt{valueOf}]\!]$ for each relevant object $o$.

(4) **User-defined methods defined in prototype objects of relevant objects.** The protected runtime described in Section 5 prevents code from defining methods on the prototypes of relevant objects, so the static analysis can ignore this pattern.

The representative rules in Figure 2 show only part of our handling of JavaScript features. The implementation covers additional instructions using similar rules, and the protected runtime rejects the remaining patterns, which we discuss in Section 5. For dynamic property accesses, BASIC-READ-DYNAMIC handle dynamic property reads. The implementation also includes dynamic-property variants of SENSITIVE-READ, SENSITIVE-WRITE, ESC-PROPERTY, ESC-WRITE-METHOD and ESC-METHOD-CALL, which we omit for brevity. In addition, ESC-METHOD-CALL has variants for implicit method calls, such as `for-of` loops that invoke an iterator method. Destructuring assignments such as `{a:b}=x` are handled by rules analogous to BASIC-READ-STATIC. When destructuring involves computed properties or rest properties, as in `let {[a]:y, ...z}=x`, the implementation falls back to BASIC-READ-DYNAMIC: it over-approximates the behavior by allowing `y` and `z` to receive all modeled properties of `x`. Function default parameters are desugared into assignments and are then handled by BASIC-ASSIGN. We do not need to model unmodeled native functions such as `Object.getOwnPropertyDescriptor` and `Reflect.defineProperty` specially: whenever such a call involves a relevant object, that object must appear among the arguments, which are therefore handled by ESC-CALL. If the analysis encounters an unsupported statement such as a `with` statement or `import.meta`, it reports *Unsupported* immediately.

*Correctness.* The capability analysis is correct in the following sense. Assume that a given JavaScript program does not cause the analysis to report *Unsupported* and is executed using the modified runtime presented in Section 5. If the execution exercises a capability $c$ at some program instruction $i$, then either $(i, c, \textit{direct}) \in R$ or $(j, c, \textit{indirect}) \in R$, where $j$ is an instruction that occurs before $i$ in the execution.

To see why this property holds, consider any execution accepted by the protected runtime. We maintain the following invariant for every execution prefix: whenever a relevant object occurs at a concrete program location represented by a constraint variable, either that variable contains the object in the computed points-to solution, or, for every capability $c$ reachable through that object, there exists an instruction $j$ that has occurred in the execution prefix such that $(j, c, \textit{indirect}) \in R$.

The invariant holds initially by construction of the native-object model. The BASIC rules preserve it for the flows that the analysis tracks explicitly: assignments, property reads, including prototype-chain lookup and dynamic reads, `require`, and calls to modeled native functions. The ESC rules preserve it for the remaining instructions that may propagate a relevant object without being tracked explicitly, including property writes, arguments to unmodeled calls, returns, throws, and exports. For each such instruction, the rules add an *indirect* report to $R$ for every capability that may be reachable through the propagated object, corresponding to the second alternative of the invariant for that object. The construction of the capability and carrier mappings ensures that this includes capabilities that may subsequently be obtained through an escaped carrier object. The protected runtime described in Section 5 rules out the exceptional language behaviors that would otherwise invalidate the invariant.

Now suppose instruction $i$ exercises capability $c$. By the native model, the corresponding operation is recorded in $CM$. If the sensitive object used by the operation is represented in the computed points-to solution, the appropriate SENSITIVE rule adds $(i, c, \textit{direct})$ to $R$. Otherwise, by applying the invariant immediately before the operation at $i$, there exists an instruction $j$ that has already occurred in the execution and for which $(j, c, \textit{indirect}) \in R$. Therefore, no capability use can occur without either a direct report at its use site or an earlier indirect report.

## 5 Runtime Enforcement

The correctness guarantee in Section 4 depends on a preservation condition: every concrete operation allowed by the protected runtime must be covered by analysis rules that preserve the analysis invariant. By comparing the specification of ECMAScript with the rules in Figure 2 for each instruction, we identify three operations that break the invariant unless treated specially. Modeling these operations statically would substantially reduce precision, and benign programs seldom use them; we therefore use the protected runtime to block them instead.

First, property reads can break the invariant through the inherited `constructor` property. In JavaScript, function objects inherit a `constructor` property from `Function.prototype`, and this property refers to the `Function` constructor which is a sensitive object. For example, `(function(){})[E]` returns the `Function` when `E` evaluates to `"constructor"`. Because the receiver is not a relevant object tracked by the analysis, the rules neither propagate the sensitive object $o_{\texttt{Function}}$ to the result nor report indirect code execution capability, breaking the invariant.

Second, function calls can break the invariant through the implicit `this` binding in non-strict mode. In JavaScript's non-strict mode, when a function is called without an explicit receiver, `this` inside the callee is bound to the `global` object.[4] For example,

[4] See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this#examples for additional explanation.

`a=function(){this.eval(...)}; a();` makes the carrier object `global` available as `this` inside the function body. The rules neither propagate `global` to `this` nor report the indirect capability usage, which breaks the invariant. Modeling this behavior statically would require treating many ordinary uses of `this` as potential accesses to the global object, which would significantly reduce precision.

Third, method calls can break the invariant through user-defined methods inherited by relevant objects. In JavaScript, method lookup follows the prototype chain, and a method call binds `this` inside the method to the receiver object. Thus, code such as `Object.prototype.m=function(){this.eval(...)}; global.m();` defines a method inherited by `global`; when `global.m()` executes, `this` is bound to the carrier object `global`, allowing the method to access `global.eval`. This behavior corresponds to the fourth method-call case in the discussion of escaping rules in Section 4.

To prevent the above three situations, we implement the runtime protection as a preload module that blocks these three mechanisms before application code executes. Users can enable the protection by passing our module to Node.js through the `--require` command-line option, which loads the module before the program entry point and places the subsequent execution under the protected runtime. Alternatively, users may import the module manually after trusted initialization code has run, allowing trusted code to use these mechanisms before protection is enabled.

To block the first case, the preload module replaces `Function.prototype.constructor` with a function that throws an error when invoked. We apply analogous protection to async functions, generator functions, and async generator functions.

To block the second case, the preload module intercepts Node.js module loading. Before a module is compiled, the preload module parses its source code and transforms function definitions whose bodies contain the `this` keyword. For each such function, it applies one of two strategies:

- If the function body contains the AST pattern `Function("return␣this")`, the `"use␣strict"` directive is inserted at the beginning of the function body. This pattern is commonly used to detect whether the current context is strict or non-strict, and the subsequent code typically contains logic that handles strict mode.
- Otherwise, the transformer adds code that tests whether `this` refers to `global`. If so, the code replaces `this` with a proxied global object. This proxy behaves like `global` except that it does not expose sensitive objects through property reads such as `global.eval`.

To block the third case, the preload module freezes the prototype objects of relevant objects. However, when a prototype object is a plain object rather than a function, the module instead clones it, freezes the properties of the clone, and replaces the prototype of the relevant object with the cloned version. In this way, the user code can still modify properties of the original prototype object while the prototype of the relevant object remains unaffected.

## 6 Evaluation

We implemented our approach in the Jelly[5] framework for JavaScript, reusing its parser and BASIC rules while adding our own rules for tracking relevant objects and reporting capability use. Since our approach, referred to as **PTR** hereafter, is the first capability analysis for JavaScript that provides capability guarantees, there are no existing tools against which we can directly compare. Existing AST-based capability-extraction techniques used in runtime protection approaches [5, 33, 34] are the closest alternatives, so we implement **AST** as a strengthened version of these techniques that provides the same capability guarantee as **PTR** under our protected runtime. **AST** reports capability usage by pattern matching statements against the AST, without computing dataflow or points-to information. **AST** emits a *direct* report at a call, property read, or property write when the sensitive object being operated on is identifiable syntactically, as in `eval()` or `require("fs").readFile()`. Since **AST** cannot track values, whenever a relevant object flows into a position where its subsequent uses can no longer be identified syntactically, as in `x.f = eval; x.f(...)`, it conservatively emits an *indirect* report for every capability that may become reachable from that object. In both analyses, we exclude modules that directly or indirectly require development dependencies. Because npm does not install a package's development dependencies for downstream users, the excluded modules are relevant only to development or testing workflows rather than production use.

Our evaluation addresses the following research questions:

**RQ1 (precision).** How precise are the capability analysis techniques? Here, *precision* measures whether a reported capability usage is actually exercised during program execution. A *direct* report is precise if the capability is indeed exercised at the reported program location. An *indirect* report is precise if the capability is indeed exercised later during program execution.
**RQ2 (location accuracy).** How accurate are the capability analysis techniques in pinpointing where capabilities are exercised? In particular, how many reports are *indirect* rather than *direct*?
**RQ3 (efficiency).** How efficient and scalable are the PTR and AST capability analysis techniques?
**RQ4 (runtime protection).** Do the runtime protection mechanisms break the execution of real-world libraries, or is it reasonable to assume that benign libraries do not rely on the three blocked features?
**RQ5 (capability usage in benign packages).** What is the distribution of capability usage among npm packages? In particular, how many packages use no capabilities and therefore require no review in malware detection scenarios? For the remaining packages, which capabilities do they use, and for what purposes?
**RQ6 (capability usage in malicious packages).** Which capabilities are used by real-world malicious packages? How many can our static analysis detect? How does capability usage in malicious packages differ from that in benign ones?

To answer RQ1, RQ2, RQ3, and RQ5, we collect 5685 non-frontend npm packages from the 10000 most-downloaded packages on the npm registry; we refer to this set as the *npm dataset*. To answer RQ4, we collect 3682 corresponding GitHub repositories from the npm dataset; we refer to this set as the *GitHub dataset*. Note that a single GitHub repository may contain multiple npm packages. To answer RQ6, we use 1311 non-frontend malicious npm packages provided by Socket; we refer to this set as the *malicious dataset*.

[5] https://github.com/cs-au-dk/jelly

Before presenting the results, we explain the terminology used in the following tables and figures:

- The analysis reports capability usage at source locations. However, multiple npm packages may depend on the same library, and identical code may therefore appear multiple times in the dataset. To account for such dependency-induced duplication, we report two statistics throughout the evaluation. **Raw** counts each analyzed npm package independently, meaning that the same dependency location may be counted multiple times if it appears in different packages. In contrast, **Dedup** removes such duplication by counting each unique library location only once. This distinction is applied consistently across different aggregation levels, including locations, modules, and packages.
- The reports are classified as either *direct* or *indirect*. For *direct* reports, we use the corresponding capability names in the tables. For *indirect* reports, however, the code often exposes an entire capability category (the first column of Table 1) rather than a single capability. For example, `exports.fs = __importStar(require("fs"))` exposes all file-system-related capabilities. We therefore aggregate indirect results at the capability-category level and denote them using the format *Category*Ind., where *Category* is the abbreviation of the capability category, such as FileInd., NetInd., and ProcExecInd.
- For each top-level capability category, we also report the total number of locations, modules, or packages (depending on the aggregation level in a given table) that are reported as using that category. We denote these totals using the format *Category*Total, such as FileTotal, NetTotal, and ProcExecTotal.

## 6.1 Precision

Table 2 presents the results of two analyses on the npm dataset. To estimate precision, for each capability we randomly sampled 20 reports from both analyses (after deduplicating locations) and manually checked whether the reported capability usage is indeed exercised during program execution. In total, we audited 540 reports, which took approximately 8 hours. We first used GPT-5.3-Codex to examine the reports and produce preliminary classifications, after which we manually verified each case. When a case was too complex to classify with confidence, we conservatively labeled it as a false positive to avoid overestimating precision. For the overall precision (Total), we compute a weighted average in which the weight of each capability is proportional to the number of entries in the **Dedup** column. The numbers shown in Table 2 result from one co-author performing the auditing. A second independent audit performed by the other co-author led to agreement in 523 of the 540 reports, and most disagreements arose from different conservative choices for indirect reports.

Both analyses achieve high precision for the file-system and process-execution categories, but lower precision for dynamic require and code execution, especially in indirect reports. This difference stems from how Node.js exposes different capabilities through distinct access patterns. To access file-system or process-execution capabilities, code must first load the corresponding module (e.g., `require("fs")` and `require("child_process")`). In most cases, code that loads these modules subsequently uses the corresponding capabilities. Therefore, even **AST** achieves high precision for these categories. In contrast, other capabilities can be obtained through more flexible patterns. For example, code may access the `eval` object through `global.eval`. As a result, any operation that stores `global` in another object property or passes it as a function argument may trigger an indirect code execution report. However, in most cases such uses are unrelated to capability acquisition. More generally, JavaScript aggregates many objects with different functionalities and sensitivities as properties of a single "super object," which makes it difficult for static analysis to infer the intent of code that introduces that object.

In addition, many such references to `global` follow the UMD pattern, where the top-level code is wrapped in an IIFE (Immediately Invoked Function Expression) and the global object is passed as an argument, even though the module itself never accesses any capability from it. We believe this case could be handled in future work by introducing special treatment for the UMD pattern.

Furthermore, **AST** analysis has lower precision because it detects capabilities at the introduction points. Many false positives occur when code imports a capability-related module only for type information or for accessing non-sensitive APIs. For example, a module requires `http` just for enumerating methods, which does not actually use the underlying capability, because the code does not import any sensitive API from the `http` module. The **PTR** analysis avoids this issue by tracking property-read operations. This also explains why **AST** produces more indirect reports than **PTR**, as discussed further in Section 6.2.

Overall, the two analyses exhibit different precision characteristics due to their distinct designs. **AST** tends to produce more false positives for indirect reports, whereas **PTR** achieves higher precision by tracking pointer flows. Overall, **PTR** achieves 70.2% precision, which we consider promising for large-scale capability auditing.

**Table 2: Capability usage locations detected by AST and PTR analysis (RQ1). Statistics are reported as counts of source locations.**

| Capability | Raw | | Dedup | | Precision | |
|---|---|---|---|---|---|---|
| | AST | PTR | AST | PTR | AST | PTR |
| *FileRead* | N/A | 6190 | N/A | 2941 | - | 100.0% |
| *FileWrite* | N/A | 2647 | N/A | 1251 | - | 100.0% |
| *FileOther* | N/A | 7591 | N/A | 3484 | - | 100.0% |
| FileInd. | 9629 | 5907 | 3850 | 1532 | 85.0% | 90.0% |
| FileTotal | 9629 | 22335 | 3850 | 9208 | 85.0% | 98.3% |
| *NetInfo* | N/A | 228 | N/A | 113 | - | 100.0% |
| *NetIO* | 432 | 1913 | 253 | 909 | 100.0% | 100.0% |
| NetInd. | 10879 | 5655 | 3639 | 1796 | 20.0% | 20.0% |
| NetTotal | 11311 | 7796 | 3892 | 2818 | 25.2% | 49.0% |
| *ProcExec* | N/A | 2126 | N/A | 1489 | - | 100.0% |
| ProcExecInd. | 1288 | 323 | 777 | 155 | 90.0% | 90.0% |
| ProcExecTotal | 1288 | 2449 | 777 | 1644 | 90.0% | 99.1% |
| *CodeDyn* | 1798 | 1906 | 759 | 868 | 100.0% | 100.0% |
| *CodeStatic* | 956 | 1781 | 440 | 588 | 100.0% | 100.0% |
| CodeInd. | 11139 | 7105 | 3270 | 2083 | 5.0% | 10.0% |
| CodeTotal | 13893 | 10792 | 4469 | 3539 | 30.5% | 47.0% |
| *DynRequire* | 2624 | 3849 | 1123 | 1420 | 100.0% | 95.0% |
| DynRequireInd. | 15549 | 9129 | 4050 | 2385 | 5.0% | 0.0% |
| DynRequireTotal | 18173 | 12978 | 5173 | 3805 | 25.6% | 35.5% |
| *Info* | 3172 | 7592 | 1316 | 3030 | 100.0% | 100.0% |
| InfoInd. | 18168 | 8495 | 5664 | 2731 | 30.0% | 25.0% |
| InfoTotal | 21340 | 16087 | 6980 | 5761 | 43.2% | 64.4% |
| Total | 50565 | 53573 | 17735 | 20887 | 42.4% | 70.2% |

**Table 3: Indirect capability reports aggregated at the module level (RQ2).**

| Capability | Raw | | Dedup | |
|---|---|---|---|---|
| | AST | PTR | AST | PTR |
| FileInd. | 8725 | 2788 | 3387 | 861 |
| NetInd. | 7219 | 2562 | 2245 | 706 |
| ProcExecInd. | 1200 | 239 | 715 | 122 |
| CodeInd. | 7224 | 3003 | 1954 | 780 |
| DynRequireInd. | 8989 | 4206 | 2406 | 982 |
| InfoInd. | 10844 | 4266 | 3201 | 1212 |
| Total | 23149 | 9101 | 7539 | 2437 |

**Table 4: Distribution of packages by the number of *indirect* reports in PTR (RQ2).**

| Capability | Raw | | | Dedup | | |
|---|---|---|---|---|---|---|
| | None | [1,10] | >10 | None | [1,10] | >10 |
| FileInd. | 4487 | 1159 | 39 | 8612 | 465 | 5 |
| NetInd. | 4421 | 1208 | 56 | 8512 | 564 | 6 |
| ProcExecInd. | 5530 | 155 | 0 | 8986 | 96 | 0 |
| CodeInd. | 4993 | 650 | 42 | 8645 | 426 | 11 |
| DynRequireInd. | 4379 | 1245 | 61 | 8414 | 659 | 9 |
| InfoInd. | 4659 | 994 | 32 | 8650 | 431 | 1 |
| Total | 3971 | 1481 | 233 | 7880 | 1178 | 24 |

## 6.2 Location Accuracy

As shown in Table 2, **PTR** is more accurate than **AST** in both the **Raw** and **Dedup** results: **PTR** identifies more direct locations and fewer indirect locations. However, counting *indirect* reports by location is biased. **AST** marks a capability as indirect at its introduction point, yielding one report per import, whereas **PTR** tracks each individual use and reports an indirect location only when object tracking fails. As a result, **PTR** could generate more location-level indirect reports even while being more accurate overall. Thus, for fairness, we will evaluate accuracy at the module level.

Table 3 presents the number of modules that contain at least one indirect report in each analysis. In both **Raw** and **Dedup**, **AST** has substantially more indirect reports than **PTR**, especially for file, network, and process capabilities, due to the capability import mechanism discussed in Section 6.1.

Table 4 further shows how these indirect modules are distributed across packages in **PTR**. The **None** column shows the number of packages with no indirect reports. The **[1,10]** column gives the number of packages with between 1 and 10 modules containing indirect reports, and **>10** gives the number of packages with more than 10 such modules. After deduplication, 86.8% of packages have no indirect reports. Among the remaining packages, 98.0% have no more than 10 such modules, while only 24 packages have more than 10. These numbers are encouraging from a security-auditing perspective. Auditing each indirect report in principle requires inspecting the entire package codebase, which takes considerably greater effort than auditing direct reports. For reference, in the sampling study for RQ1, excluding the 2 hours spent writing LLM prompts, we spent 6 hours auditing 540 sampled reports, of which more than 4 hours were devoted to investigating indirect behaviors. The small number of packages with indirect reports suggests that auditing is manageable for most packages.

In conclusion, **PTR** is more accurate than **AST** in pinpointing where capabilities are used. In the deduplicated results, 86.8% of packages have no indirect reports, and among those that do, 98.0% have at most 10 modules with indirect reports, making manual auditing feasible.

## 6.3 Performance

We evaluated the analysis time of all packages under both **AST** and **PTR**. Both analyses are highly scalable: 99.0% of packages are analyzed within 20 seconds in both **AST** and **PTR**, and all packages are analyzed within 2 minutes. Given that the largest package in our dataset comprises 5889 modules and 86 MB of code, a 2-minute analysis time appears reasonable.

Although **PTR** is more precise, there is no significant difference in running time between the two analyses. This is because, unlike conventional pointer analysis, **PTR** tracks only a limited number of objects and omits the rules for property writes and call expressions, so the generated constraint set remains small and the analysis scales well.

In summary, both **AST** and **PTR** finish analyzing most of the packages in a few seconds, and **PTR** has no significant additional runtime cost.

## 6.4 Runtime Protection

For each repository in the GitHub dataset, we ran the unit tests with our runtime protection mechanism enabled. Only 67 packages had tests rejected by the protection mechanism.

We manually inspected the rejected packages and found that most cases were caused by testing or development tooling. For example, 29 packages failed because of *istanbul-lib-instrument*, which injects `let _Function=(function(){}).constructor` into every required module. This behavior appears to obtain the real `Function` object from a unit-test environment. It is unlikely to appear in production environments. We observed similar patterns in *webpack*, where the relevant code is documented as unit-test code, and in *@angular/core@9.0.x*, whose version is outdated. Another package *mdx* uses `Object.getPrototypeOf(run).constructor` to obtain the `AsyncFunction` constructor for asynchronous JavaScript evaluation and exports this functionality. The functionality of *mdx* compiles `.mdx`, a format that supports Markdown and JSX syntax, into JavaScript and is typically used in build-time workflows rather than at runtime. We also found a few test files that use `Function.constructor` for reasons similar to those in *istanbul-lib-instrument*, and these files are likewise not executed in production environments. We omit other similar problematic packages for brevity.

We further compared the number of passed tests with and without the protection mechanism to determine whether any tests failed silently without explicit rejection. Apart from a few additional failures caused by AST-transformation timeouts, the remaining minor failures were due to test oracles that are sensitive to the presence of the protection mechanism, for example by checking whether `this` is the native global object or by observing stack-trace mismatches caused by the injected function frame. Notably, these timeouts arose from repeated AST transformation in the unit-test setting and therefore do not affect real runtime efficiency.

In conclusion, our runtime protection mechanism affects very few packages (1.8%), and our manual inspection suggests that these cases mostly arise outside production environments.

**Table 5: Capability counts for raw vs. deduplicated packages (RQ5).**

| Capability | Raw | Dedup | Capability | Raw | Dedup |
|---|---|---|---|---|---|
| *FileRead* | 1050 | 890 | CodeInd. | 1198 | 470 |
| *FileWrite* | 474 | 352 | *CodeStatic* | 649 | 203 |
| *FileOther* | 745 | 592 | CodeTotal | 1433 | 697 |
| FileInd. | 692 | 437 | *DynRequire* | 685 | 520 |
| FileTotal | 1278 | 1305 | DynRequireInd. | 1264 | 570 |
| *NetInfo* | 127 | 67 | DynRequireTotal | 1430 | 907 |
| *NetIO* | 470 | 323 | *Info* | 932 | 651 |
| NetInd. | 1026 | 432 | InfoInd. | 1306 | 668 |
| NetTotal | 1182 | 690 | InfoTotal | 1507 | 1067 |
| *ProcExec* | 367 | 307 | *CodeDyn* | 450 | 228 |
| ProcExecInd. | 155 | 96 | NoCapability | 3376 | 6625 |
| ProcExecTotal | 428 | 359 | No or Only *Info* | 3463 | 6824 |
| | | | **Total** | **5685** | **9082** |

**Table 6: Capability usage in malicious packages (RQ6).**

| Capability | Entry | Full | Capability | Entry | Full |
|---|---|---|---|---|---|
| *FileRead* | 725 | 771 | CodeInd. | 17 | 105 |
| *FileWrite* | 46 | 54 | CodeTotal | 65 | 134 |
| *FileOther* | 56 | 84 | *CodeStatic* | 10 | 77 |
| FileInd. | 12 | 20 | CodeStaticTotal | 10 | 77 |
| FileTotal | 753 | 798 | *DynRequire* | 53 | 62 |
| *NetInfo* | 683 | 686 | DynRequireInd. | 34 | 114 |
| *NetIO* | 1096 | 1156 | DynRequireTotal | 64 | 142 |
| NetInd. | 11 | 93 | *Info* | 1130 | 1173 |
| NetTotal | 1126 | 1188 | InfoInd. | 93 | 173 |
| *ProcExec* | 234 | 237 | InfoTotal | 1146 | 1200 |
| ProcExecInd. | N/A | 7 | AnyCap | 1297 | 1309 |
| ProcExecTotal | 234 | 242 | NoCapability | 14 | 2 |
| *CodeDyn* | 51 | 60 | No or Only *Info* | 20 | 5 |
| | | | **Total** | **1311** | **1311** |

## 6.5 Capability Usage in Benign Packages

Table 5 shows which capabilities are used in the npm dataset.

The most notable finding is that a large fraction of packages use no capabilities at all. The **NoCapability** row shows that, before deduplication, 59.4% of packages use no capabilities, and after deduplication this figure rises to 72.9%. If we assume that packages do not collude with one another to perform malicious actions (i.e., each package may call only native libraries, its own code, and the dependencies declared in `package.json`), then packages that use only *Info* capabilities can also be treated as benign. Under this assumption, the **No or Only *Info*** row shows that, after deduplication, 75.1% of packages are benign.

For packages that use capabilities, those capabilities are often needed to implement the functionality of the package itself, while some modules intentionally re-export capabilities to their consumers. For example, *tsconfig-paths* re-exports a decorated `require` function. Unlike *@ungap/structured-clone* in our motivating example, this case does not involve a vulnerability; the module is designed to expose this capability. These packages further imply the necessity of including dependency code when analyzing capability usage or detecting malicious code. We also found more complex cases, such as *fengari*, where a package implements an interpreter for another language in JavaScript, effectively exposing all capabilities. Detecting such packages and applying cross-language analysis is an interesting direction for future work.

Among individual capability categories, *FileRead*, *FileWrite*, *FileOther*, and *Info* capabilities are the most frequently used because the npm dataset—and, more broadly, the npm package ecosystem—contains numerous tools for compiling, scaffolding, testing, etc.

Process execution and code execution are the least frequently used capabilities, which is encouraging from a security perspective, as they are higher-risk than other capabilities. Packages use process execution to inspect or manage system processes, run `git` commands, and check the versions of their dependencies. For code execution, a large fraction of packages execute code from constant strings, e.g., `Function("return this")()`; packages use this pattern to detect the current runtime environment, which appears to pose little security risk. In the future, such constant-string cases could be analyzed statically with little difficulty. Another common use involves the WebAssembly API, where packages offload computation-intensive operations to WebAssembly for performance. This observation also suggests that future malware analysis may require cross-language analysis.

Among the *indirect* reports, we found that some cases result from intentional design choices rather than imprecision in our analysis. For example, *node-pn* defines all promisified versions of file system APIs in its module export fields, which our analysis therefore reports as *indirect* capabilities.

In summary, 75.1% of packages use no capabilities or only *Info* capabilities, meaning that they are benign and require no further review. The remaining packages in the npm ecosystem use capabilities mainly for compiling, scaffolding, testing, etc., which is expected given their intended functionality.

## 6.6 Malicious Capability Usage

Table 6 shows capability usage among malicious packages. In this table we do not distinguish between **Raw** and **Dedup** because we are more interested in the capability usage of individual malicious packages, and their dependencies do not heavily overlap. We distinguish **Entry** and **Full**—**Entry** counts only the capabilities reported in the entry-point package, i.e., the malicious package itself, while **Full** also includes capability usage from all of its dependencies.

The **Full** column shows that, among 1311 malicious packages, our static analysis approach detects 1309 packages that use at least one sensitive capability. The remaining 2 malicious packages evade our static analysis by using the function constructor feature, as discussed in Section 5. However, all of them are blocked by our runtime protection mechanism.

Moreover, most of the malicious packages use multiple capabilities, which is consistent with observations from previous work [15, 16]. In addition, the malicious code tends to focus on information collection and data exfiltration rather than on higher-risk capabilities such as code execution or process execution.

We inspected three packages that **PTR** classifies as using only *Info* capabilities. Two of them send information by calling a custom function stored in a `global` property; the attack may rely on another malicious package registering a function with network capability on the global object and then requiring these two packages to invoke it for data exfiltration. The third package exports an encoded buffer containing a sensitive payload; we suspect that it is intended to

be imported and executed by another malicious package. All three cases suggest that malicious packages can cooperate to carry out attacks. In practice, this means that our tool is most effective when all dependencies are analyzed together.

The **Entry** and **Full** columns show that capabilities are typically used directly in the entry-point package. Specifically, 99.1% (1297/1309) of malicious packages use at least one capability directly in the entry-point package. By comparison, the corresponding percentage in the npm dataset is 57.1% (1329/2329), suggesting benign packages are more likely to access capabilities through other libraries. This suggests that many malicious packages in our dataset use capabilities relatively directly, but it also leaves open the possibility that more sophisticated ones remain undetected.

In summary, our static analysis detects 99.8% of malicious packages; the remaining packages are prevented by our runtime protection mechanism. Compared to benign packages, the detected malicious packages tend to use capabilities more directly.

## 7 Related Work

*Capability concepts.* Capabilities are unforgeable references that designate objects and confer authority to invoke operations on them [7]. Our model adopts this perspective by associating selected operations on security-relevant native objects with capabilities (Table 1). Guha et al. [14] study fine-grained capability policies for browser extensions in a specialized language, and Melicher et al. [23] present a language with a type system for statically checking capability use. Unlike these language-specific policy or type-system approaches, our work guarantees that every exercised capability in general JavaScript libraries is reported under a lightweight protected runtime.

*Malicious package detection.* Existing malicious-package detectors use security-relevant API usage, information flow, version differences, syntactic or lexical features, and LLM-based review to identify malicious packages [6, 8, 10, 11, 15–17, 19, 25, 28, 30, 36–38]. This information may serve as a feature, rule trigger, or seed for static or dynamic behavioral analysis. These approaches do not guarantee complete capability coverage for general Node.js code: some omit dependency code [25, 28, 38], and they do not exhaustively model JavaScript's dynamic features, native objects, or data flows across packages.

Prior work also demonstrates evasion of syntactic detectors through AST rewriting [9] or WebAssembly [27], and limitations of LLM-based analysis through unreliable reasoning [32] or attention-diversion attacks [22]. These results motivate capability analysis that does not miss exercised capabilities.

*Node.js vulnerability detection.* Several Node.js analyses target particular vulnerability classes rather than general capability usage. Li et al. [21] construct object dependence graphs using static analysis and apply vulnerability-specific queries, whereas Li et al. [20] use static taint analysis specialized for detecting prototype pollution. Unlike our analysis, these approaches determine whether particular vulnerability conditions hold rather than report all uses of security-sensitive capabilities. Cassel et al. [3] use dynamic taint tracking, fuzzing, and exploit synthesis to detect arbitrary command injection and code execution. Their approach complements ours: it can establish exploitability for flows reached by generated inputs by synthesizing exploits, whereas our analysis identifies capability uses that can guide subsequent exploitability analysis.

*Runtime protection and sandboxing.* Runtime protection and sandboxing approaches restrict the authority of untrusted code by enforcing capability policies [1, 5, 12, 13, 31, 33, 34]. A central challenge is constructing policies that are restrictive enough to limit attacks while preserving intended behavior. Existing mechanisms include static and import-time analysis for inferring RWX permissions [33], call-graph-based derivation of system-call allowlists [34], and enforcement of capability-enhanced SBOM policies [5].

Pohl et al. [26] derive package policies from JavaScript ASTs by extracting imported modules, global objects, and explicitly accessed members, and enforce the policies by modifying the Node.js runtime. Their policy inference is therefore closest to our AST baseline, rather than our pointer-based analysis, and does not guarantee complete capability coverage. Our approach is complementary: it reports every exercised capability under the protected runtime, which can support policy generation and review.

*Defensive points-to analysis.* Defensive points-to analysis and related work aim to provide analysis guarantees in the presence of opaque or external code in Java or C programs, without focus on capabilities. Smaragdakis and Kastrinis [29] retain points-to information only when the corresponding points-to set is known to be bounded. Krogstie et al. [18] model locations accessible from external modules while preserving precision for locations that have not escaped. Inspired by these approaches, our analysis tracks the capability-relevant objects only to the extent it can be done soundly and efficiently. When such an object flows through an operation that we do not model precisely, we conservatively report every capability reachable through that object.

## 8 Conclusion

We have presented a new approach to capability analysis for JavaScript packages that ensures that every exercised capability is reported under a protected runtime. Our approach combines a defensive capability analysis that tracks only capability-relevant objects and reports capability usage whenever the points-to information can no longer track them precisely, with a lightweight runtime enforcement mechanism that blocks the small set of JavaScript mechanisms that would otherwise invalidate this guarantee. The experimental results show that our approach achieves high precision and good accuracy in identifying capability usage locations, while also being scalable for large codebases.

We hope this work encourages further research on capability analysis with strong coverage guarantees. One direction is to make JavaScript runtimes like Node.js more amenable to static analysis, for example by addressing the threats discussed in Section 5 and by avoiding designs that expose sensitive functionality through readily accessible objects. Another direction is to develop new analysis techniques that address the remaining challenges involving `global` and similar objects, while pinpointing capability-usage locations even more accurately.

## Acknowledgments

This work was supported by the Danish Research Council for Technology and Production, grant ID 10.46540/3105-00037B. We also thank Socket for providing us with the malicious dataset.

## Data Availability

The implementation source code is available at https://zenodo.org/records/21760897. Datasets other than the malicious dataset are publicly available; the malicious dataset cannot be released because of company data policy.

## References

[1] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, Robert H'obbes' Zakon (Ed.). ACM, 1–10. doi:10.1145/2420950.2420952

[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph. D. Dissertation. University of Copenhagen.

[3] Darion Cassel, Nuno Sabino, Min-Chien Hsu, Ruben Martins, and Limin Jia. 2025. NodeMedic-FINE: Automatic Detection and Exploit Synthesis for Node.js Vulnerabilities. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nodemedic-fine-automatic-detection-and-exploit-synthesis-for-node-js-vulnerabilities/

[4] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. 2022. On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages. *IEEE Trans. Software Eng.* 48, 8 (2022), 2695–2708. doi:10.1109/TSE.2021.3068901

[5] Eric Cornelissen and Musard Balliu. 2025. NodeShield: Runtime Enforcement of Security-Enhanced SBOMs for Node.js. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*, Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier (Eds.). ACM, 1844–1858. doi:10.1145/3719027.3765136

[6] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association. http://static.usenix.org/events/sec11/tech/full_papers/Curtsinger.pdf

[7] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155. doi:10.1145/365230.365252

[8] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/towards-measuring-supply-chain-attacks-on-package-managers-for-interpreted-languages/

[9] Aurore Fass, Michael Backes, and Ben Stock. 2019. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1899–1913. doi:10.1145/3319535.3345656

[10] Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: a static pre-filter for malicious JavaScript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, David M. Balenson (Ed.). ACM, 257–269. doi:10.1145/3359789.3359813

[11] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. 2018. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10885)*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.). Springer, 303–325. doi:10.1007/978-3-319-93411-2_14

[12] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1334–1346. doi:10.1109/ICSE43902.2021.00121

[13] Willem De Groef, Fabio Massacci, and Frank Piessens. 2014. NodeSentry: least-privilege library integration for server-side JavaScript. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, Charles N. Payne Jr., Adam Hahn, Kevin R. B. Butler, and Micah Sherr (Eds.). ACM, 446–455. doi:10.1145/2664243.2664276

[14] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. 2011. Verified Security for Browser Extensions. In *32nd IEEE Symposium on Security and Privacy, SP 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 115–130. doi:10.1109/SP.2011.36

[15] Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, Zhen Yang, and Lei Shi. 2024. DONAPI: Malicious NPM Packages Detector using Behavior Sequence Knowledge Mapping. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/huang-cheng

[16] Yiheng Huang, Ruisi Wang, Wen Zheng, Zhuotong Zhou, Susheng Wu, Shulin Ke, Bihuan Chen, Shan Gao, and Xin Peng. 2024. SpiderScan: Practical Detection of Malicious NPM Packages Based on Graph-Based Behavior Modeling and Matching. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1146–1158. doi:10.1145/3691620.3695492

[17] Yiheng Huang, Wen Zheng, Susheng Wu, Bihuan Chen, You Lu, Zhuotong Zhou, Yiheng Cao, Xiaoyu Li, and Xin Peng. 2025. ProfMal: Detecting Malicious NPM Packages by the Synergy between Static and Dynamic Analysis. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE, 419–431. doi:10.1109/ASE63991.2025.00042

[18] Håvard Rognebakke Krogstie, Helge Bahmann, Magnus Själander, and Nico Reissmann. 2026. PIP: Making Andersen's Points-to Analysis Sound and Practical for Incomplete C Programs. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2026, Sydney, Australia, January 31 - Feb. 4, 2026*, Stephen M. Blackburn, Albert Cohen, and Timothy M. Jones (Eds.). IEEE, 535–547. doi:10.1109/CGO68049.2026.11395202

[19] Piergiorgio Ladisa, Serena Elisa Ponta, Nicola Ronzoni, Matias Martinez, and Olivier Barais. 2023. On the Feasibility of Cross-Language Detection of Malicious Packages in npm and PyPI. In *Annual Computer Security Applications Conference, ACSAC 2023, Austin, TX, USA, December 4-8, 2023*. ACM, 71–82. doi:10.1145/3627106.3627138

[20] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting Node.js prototype pollution vulnerabilities via object lookup analysis. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 268–279. doi:10.1145/3468264.3468542

[21] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 143–160. https://www.usenix.org/conference/usenixsecurity22/presentation/li-song

[22] Xiao Li, Yue Li, Hao Wu, Yue Zhang, Kaidi Xu, Xiuzhen Cheng, Sheng Zhong, and Fengyuan Xu. 2025. Make a Feint to the East While Attacking in the West: Blinding LLM-Based Code Auditors with Flashboom Attacks. In *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, Marina Blanton, William Enck, and Cristina Nita-Rotaru (Eds.). IEEE, 576–594. doi:10.1109/SP61157.2025.00125

[23] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, Barcelona, Spain, June 19-23, 2017 (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:27. doi:10.4230/LIPICS.ECOOP.2017.20

[24] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting locations in JavaScript programs affected by breaking library changes. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 187:1–187:25. doi:10.1145/3428255

[25] Marc Ohm, Felix Boes, Christian Bungartz, and Michael Meier. 2022. On the Feasibility of Supervised Machine Learning for the Detection of Malicious Software Packages. In *ARES 2022: The 17th International Conference on Availability, Reliability and Security, Vienna,Austria, August 23 - 26, 2022*. ACM, 127:1–127:10. doi:10.1145/3538969.3544415

[26] Timo Pohl, Marc Ohm, Felix Boes, and Michael Meier. 2024. You Can Run But You Can't Hide: Runtime Protection Against Malicious Package Updates For Node.js. In *Sicherheit, Schutz und Zuverlässigkeit: Konferenzband der 12. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), Sicherheit 2024, Worms, Germany, April 9-11, 2024 (LNI, Vol. P-345)*, Steffen Wendzel, Christian Wressnegger, Laura Hartmann, Felix C. Freiling, Frederik Armknecht, and Lena Reinfelder (Eds.). Gesellschaft für Informatik e.V., 231–241. doi:10.18420/SICHERHEIT2024_015

[27] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. 2022. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1574–1589. doi:10.1109/SP46214.2022.9833626

[28] Adriana Sejfia and Max Schäfer. 2022. Practical Automated Detection of Malicious npm Packages. In *44th IEEE/ACM 44th International Conference on Software*

*Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1681–1692. doi:10.1145/3510003.3510104

[29] Yannis Smaragdakis and George Kastrinis. 2018. Defensive Points-To Analysis: Effective Soundness via Laziness. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, Amsterdam, The Netherlands, July 16-21, 2018 (LIPIcs, Vol. 109)*, Todd D. Millstein (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:28. doi:10.4230/LIPICS.ECOOP.2018.23

[30] Raphael J. Sofaer, Yaniv David, Mingqing Kang, Jianjia Yu, Yinzhi Cao, Junfeng Yang, and Jason Nieh. 2024. RogueOne: Detecting Rogue Updates via Differential Data-flow Analysis Using Trust Domains. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 101:1–101:13. doi:10.1145/3597503.3639199

[31] Tung Tran, Riccardo Pelizzi, and R. Sekar. 2015. JaTE: Transparent and Efficient JavaScript Confinement. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*. ACM, 151–160. doi:10.1145/2818000.2818019

[32] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse K. Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 862–880. doi:10.1109/SP54263.2024.00210

[33] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1821–1838. doi:10.1145/3460120.3484535

[34] Wenya Wang, Xingwei Lin, Jingyi Wang, Wang Gao, Dawu Gu, Wei Lv, and Jiashui Wang. 2023. HODOR: Shrinking Attack Surface on Node.js via System Call Limitation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2800–2814. doi:10.1145/3576915.3616609

[35] Yingyuan Pu, Lingyun Ying, and Yacong Gu. 2026. From Noise to Signal: Precisely Identify Affected Packages of Known Vulnerabilities in npm Ecosystem. In *Proceedings of the 33rd Network and Distributed System Security Symposium (NDSS 2026)*. Internet Society, San Diego, CA, USA. doi:10.14722/ndss.2026.241902

[36] Zeliang Yu, Ming Wen, Xiaochen Guo, and Hai Jin. 2024. Maltracker: A Fine-Grained NPM Malware Tracker Copiloted by LLM-Enhanced Dataset. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1759–1771. doi:10.1145/3650212.3680397

[37] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie A. Williams. 2025. Leveraging Large Language Models to Detect NPM Malicious Packages. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2625–2637. doi:10.1109/ICSE55347.2025.00146

[38] Junan Zhang, Kaifeng Huang, Yiheng Huang, Bihuan Chen, Ruisi Wang, Chong Wang, and Xin Peng. 2025. Killing Two Birds with One Stone: Malicious Package Detection in NPM and PyPI using a Single Model of Malicious Behavior Sequence. *ACM Trans. Softw. Eng. Methodol.* 34, 4 (2025), 104:1–104:28. doi:10.1145/3705304

[39] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman

Received 2026-03-26; accepted 2026-06-18