# From Noise to Signal: Precisely Identify Affected Packages of Known Vulnerabilities in npm Ecosystem

**Yingyuan Pu, Lingyun Ying, Yacong Gu**

**puyingyuan01@qianxin.com**

## The npm Supply Chain Security Challenge

**Background:** npm is The world's largest software ecosystem, 3+ million packages

**The Problem:** *Complex Dependency Chains*

- Deep nesting, intricate dependencies
- Single vulnerability → massive downstream impact
- Example: pac-resolver CVE → 285,000+ GitHub repos at risk

**Current Reality:**

- ~25% of package versions depend on vulnerable packages

## Current State - The Alert Fatigue Crisis

**Package-Level Analysis:**

- SCA tools (npm audit, Dependabot) report vulnerable dependencies
- Developers face critical question: *"Am I really affected?"*

**The Alert Fatigue Crisis:**

- 95% of vulnerabilities have fixes available
- Yet 80% of enterprise dependencies remain unpatched for >1 year
- Root cause: "Patching paralysis" from alert overload
- Uncertainty about true impact → costly update process

## Package-Level Analysis Limitation

**The Current Standard:** Package-Level Analysis

- SCA tools report vulnerable dependencies present in dependency graph

**Critical Limitation:** Coarse-grained analysis

- Presence ≠ Reachability
- Cannot determine if vulnerable code is actually called
- Leads to massive false positive rates

## Why Not Function-Level Analysis? (Scalability)

**The Logical Solution**: *Function-Level Reachability*

- Determine if vulnerable code is actually called
- This is the necessary first step before *exploitability*

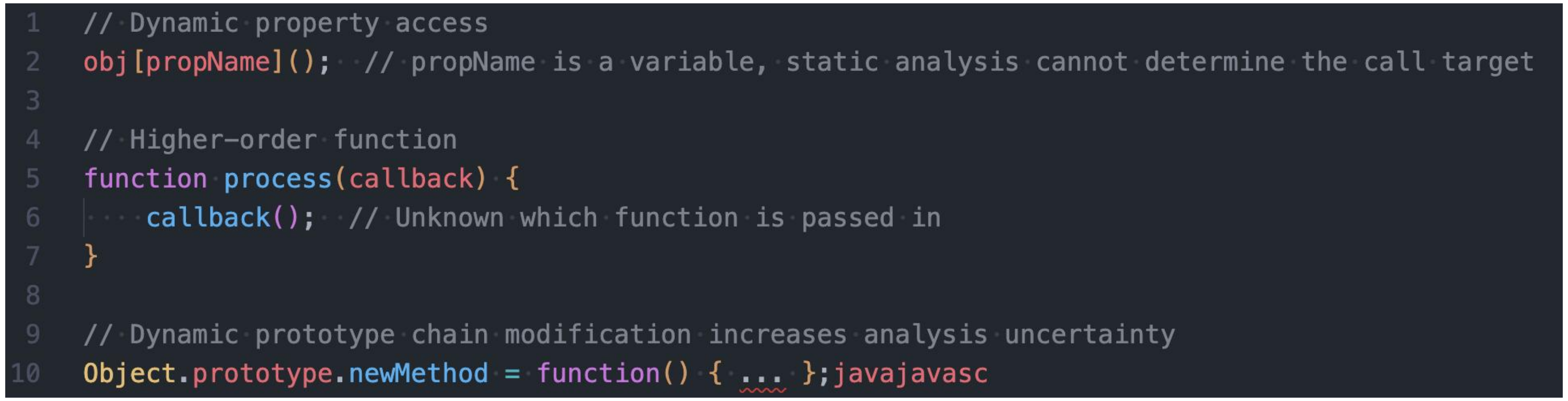
**Challenge 1: Scalability**

- Existing approaches face critical computational barriers
- Whole-program analysis is computationally prohibitive
- **Jelly/JAM:** Only 37% success rate under 4GB memory limit
- Must re-analyze entire dependency graph for each project

## Why Not Function-Level Analysis? (Dynamic Nature)

### Challenge 2: JavaScript's Dynamic Nature

- First-class functions: passed as arguments, returned from functions
- Callbacks and higher-order functions obscure call targets
- Dynamic property access: obj[variable]() → statically undecidable



```javascript
// Dynamic property access
obj[propName]();  // propName is a variable, static analysis cannot determine the call target

// Higher-order function
function process(callback) {
    callback();  // Unknown which function is passed in
}

// Dynamic prototype chain modification increases analysis uncertainty
Object.prototype.newMethod = function() { ... };javajavasc
```

**Impact:** Simple syntactic analysis is highly imprecise

## Why Not Function-Level Analysis? (Module Systems)

**Challenge 3**: JavaScript's Module Systems

- CommonJS (CJS): require(), module.exports
  - Mutable exports modified at runtime
  - Dynamic require expressions: require(variable)
- ECMAScript Modules (ESM): static imports
- **Challenge:** CJS + ESM interoperability

**Our Goal:** Precise function-level analysis at *ecosystem scale*

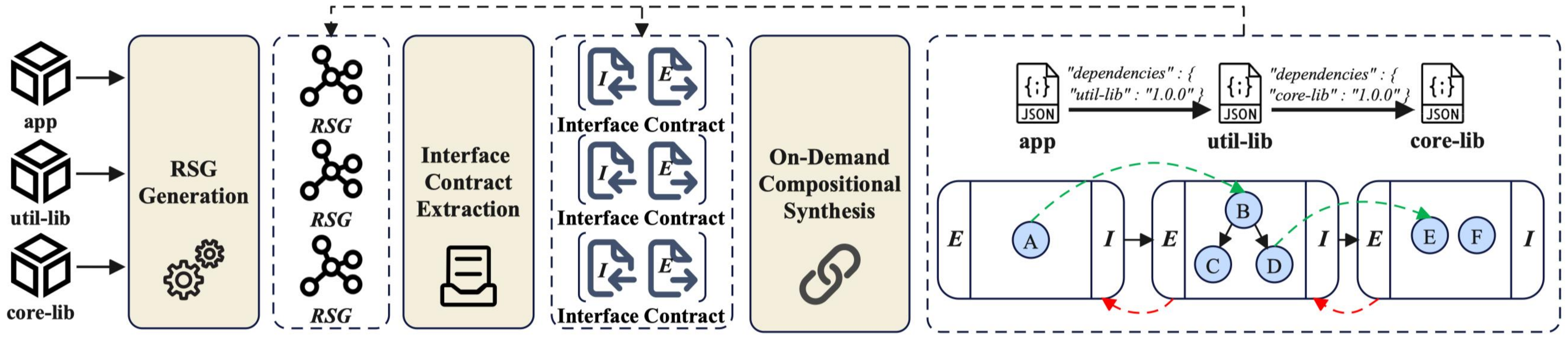
## Methodology - Key Insight

**Key Insight:** *npm package versions are immutable*

- Once published, (package, version) pair never changes
- **Opportunity:** Pre-compute once, reuse many times
- **Our Strategy:** "Analyze-once, reuse-many-times" model
- Enables *ecosystem-scale* analysis with practical performance

## VulTracer Overview

- **Key Insight:** *Pre-compute once, reuse many times*



app
util-lib
core-lib
RSG Generation
RSG
RSG
RSG
Interface Contract Extraction
I
E
Interface Contract
Interface Contract
Interface Contract
On-Demand Compositional Synthesis
JSON
"dependencies" : { "util-lib" : "1.0.0" }
JSON
"dependencies" : { "core-lib" : "1.0.0" }
JSON
app
util-lib
core-lib
E
A
I
E
B
C
D
I
E
E
F
I

Fig. 1: Overview of VulTracer.

# Phase 1 - Rich Semantic Graph (RSG)

**Rich Semantic Graph (RSG):** Beyond Standard Call Graphs

**Traditional CG Problem:**

- function A() → function B()
- Loses boundary information needed for composition
- Cannot model external dependencies or public APIs

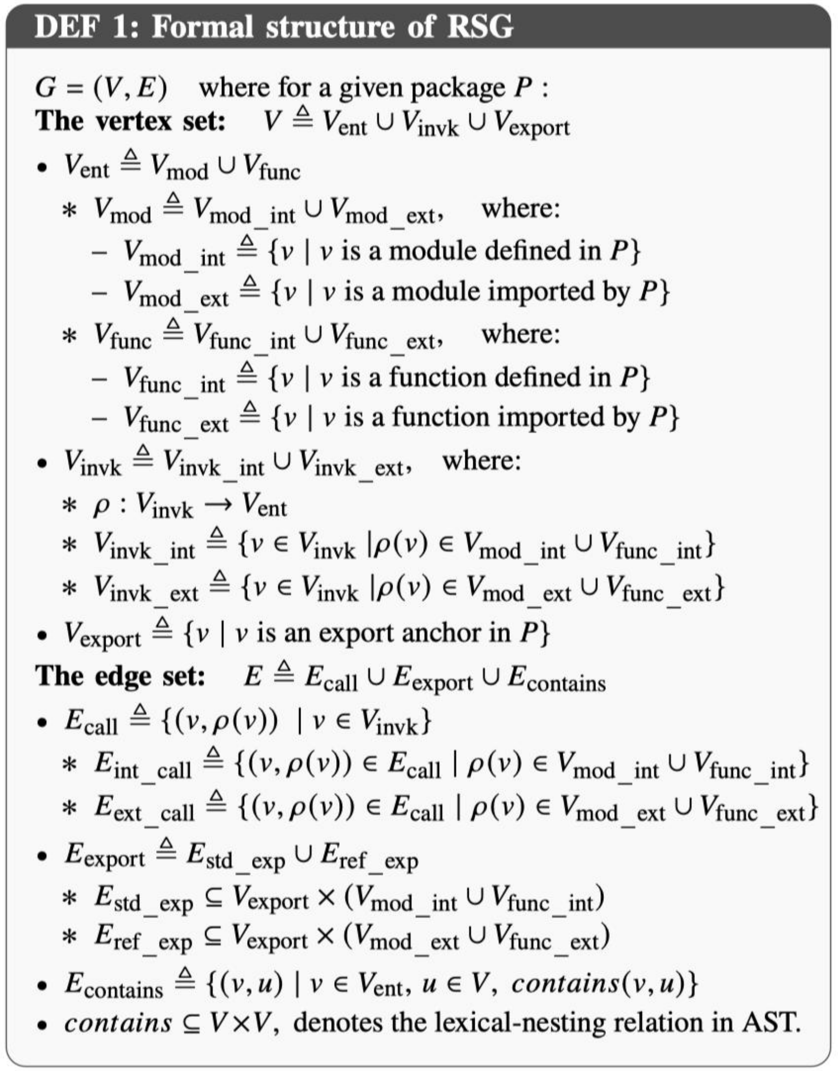
**RSG Solution:** *Reifies boundaries as vertices:*

**Vertex Types:**

- Programmatic entities (modules, functions)
- Invocation points (reified as vertices)
- Export anchors (public API)

**Edge Types:**

- Lexical nesting (contains)
- Call resolution (internal/external)
- Export resolution (standard/reference)



**DEF 1: Formal structure of RSG**

$G = (V, E)$ where for a given package $P$ :

**The vertex set:** $V \triangleq V_{\text{ent}} \cup V_{\text{invk}} \cup V_{\text{export}}$

- $V_{\text{ent}} \triangleq V_{\text{mod}} \cup V_{\text{func}}$
  - ∗ $V_{\text{mod}} \triangleq V_{\text{mod\_int}} \cup V_{\text{mod\_ext}}$, where:
    - – $V_{\text{mod\_int}} \triangleq \{v \mid v \text{ is a module defined in } P\}$
    - – $V_{\text{mod\_ext}} \triangleq \{v \mid v \text{ is a module imported by } P\}$
  - ∗ $V_{\text{func}} \triangleq V_{\text{func\_int}} \cup V_{\text{func\_ext}}$, where:
    - – $V_{\text{func\_int}} \triangleq \{v \mid v \text{ is a function defined in } P\}$
    - – $V_{\text{func\_ext}} \triangleq \{v \mid v \text{ is a function imported by } P\}$
- $V_{\text{invk}} \triangleq V_{\text{invk\_int}} \cup V_{\text{invk\_ext}}$, where:
  - ∗ $\rho : V_{\text{invk}} \rightarrow V_{\text{ent}}$
  - ∗ $V_{\text{invk\_int}} \triangleq \{v \in V_{\text{invk}} \mid \rho(v) \in V_{\text{mod\_int}} \cup V_{\text{func\_int}}\}$
  - ∗ $V_{\text{invk\_ext}} \triangleq \{v \in V_{\text{invk}} \mid \rho(v) \in V_{\text{mod\_ext}} \cup V_{\text{func\_ext}}\}$
- $V_{\text{export}} \triangleq \{v \mid v \text{ is an export anchor in } P\}$

**The edge set:** $E \triangleq E_{\text{call}} \cup E_{\text{export}} \cup E_{\text{contains}}$

- $E_{\text{call}} \triangleq \{(v, \rho(v)) \mid v \in V_{\text{invk}}\}$
  - ∗ $E_{\text{int\_call}} \triangleq \{(v, \rho(v)) \in E_{\text{call}} \mid \rho(v) \in V_{\text{mod\_int}} \cup V_{\text{func\_int}}\}$
  - ∗ $E_{\text{ext\_call}} \triangleq \{(v, \rho(v)) \in E_{\text{call}} \mid \rho(v) \in V_{\text{mod\_ext}} \cup V_{\text{func\_ext}}\}$
- $E_{\text{export}} \triangleq E_{\text{std\_exp}} \cup E_{\text{ref\_exp}}$
  - ∗ $E_{\text{std\_exp}} \subseteq V_{\text{export}} \times (V_{\text{mod\_int}} \cup V_{\text{func\_int}})$
  - ∗ $E_{\text{ref\_exp}} \subseteq V_{\text{export}} \times (V_{\text{mod\_ext}} \cup V_{\text{func\_ext}})$
- $E_{\text{contains}} \triangleq \{(v, u) \mid v \in V_{\text{ent}},\ u \in V,\ contains(v, u)\}$
- $contains \subseteq V \times V$, denotes the lexical-nesting relation in AST.

**Result:** Self-contained, composable package representations

## Phase 2 - Interface Contracts

**Formal Abstraction for Composition:** Interface Contracts

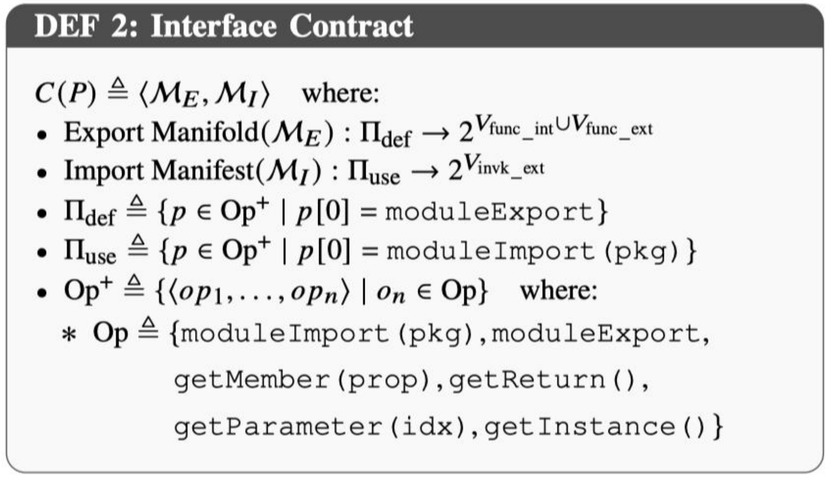
**Contract = ⟨Export Manifold, Import Manifest⟩**

**Export Manifold (ME):**

- API Path → Set of Functions
- Example: ⟨moduleExport, getMember('process')⟩ → {process_func}
- Example: ⟨moduleExport, getMember('parser')⟩ → {external_parse_func}

**Import Manifest (MI):**

- Use Path → Set of Invocation Points
- Example: ⟨moduleImport('util-lib'), getMember('process')⟩ → {invk_node_1}
- Catalogs external dependencies



**DEF 2: Interface Contract**

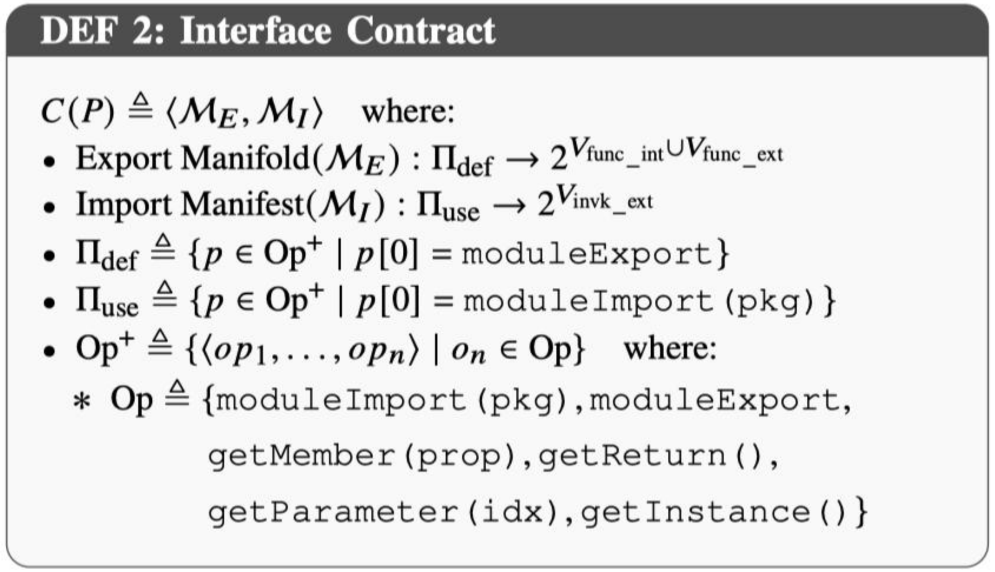
$C(P) \triangleq \langle \mathcal{M}_E, \mathcal{M}_I \rangle$ where:

- Export Manifold($\mathcal{M}_E$) : $\Pi_{\text{def}} \rightarrow 2^{V_{\text{func\_int}} \cup V_{\text{func\_ext}}}$
- Import Manifest($\mathcal{M}_I$) : $\Pi_{\text{use}} \rightarrow 2^{V_{\text{invk\_ext}}}$
- $\Pi_{\text{def}} \triangleq \{p \in \text{Op}^+ \mid p[0] = \texttt{moduleExport}\}$
- $\Pi_{\text{use}} \triangleq \{p \in \text{Op}^+ \mid p[0] = \texttt{moduleImport(pkg)}\}$
- $\text{Op}^+ \triangleq \{\langle op_1, \ldots, op_n \rangle \mid o_n \in \text{Op}\}$ where:
  - $\text{Op} \triangleq \{\texttt{moduleImport(pkg)}, \texttt{moduleExport}, \texttt{getMember(prop)}, \texttt{getReturn()}, \texttt{getParameter(idx)}, \texttt{getInstance()}\}$

# Phase 2 - Interface Contracts

**Benefits:**

- Machine-readable API specification
- Enables semantic matching during composition
- Decouples implementation from interface



**DEF 2: Interface Contract**

$C(P) \triangleq \langle \mathcal{M}_E, \mathcal{M}_I \rangle$ where:

- Export Manifold$(\mathcal{M}_E)$ : $\Pi_{\text{def}} \rightarrow 2^{V_{\text{func\_int}} \cup V_{\text{func\_ext}}}$
- Import Manifest$(\mathcal{M}_I)$ : $\Pi_{\text{use}} \rightarrow 2^{V_{\text{invk\_ext}}}$
- $\Pi_{\text{def}} \triangleq \{p \in \text{Op}^+ \mid p[0] = \texttt{moduleExport}\}$
- $\Pi_{\text{use}} \triangleq \{p \in \text{Op}^+ \mid p[0] = \texttt{moduleImport(pkg)}\}$
- $\text{Op}^+ \triangleq \{\langle op_1, \ldots, op_n \rangle \mid o_n \in \text{Op}\}$ where:
  - $*$ $\text{Op} \triangleq \{\texttt{moduleImport(pkg)}, \texttt{moduleExport}, \texttt{getMember(prop)}, \texttt{getReturn()}, \texttt{getParameter(idx)}, \texttt{getInstance()}\}$

## Phase 3 - Compositional Synthesis

**Building Ecosystem-Scale Call Graphs:** *Compositional Synthesis*

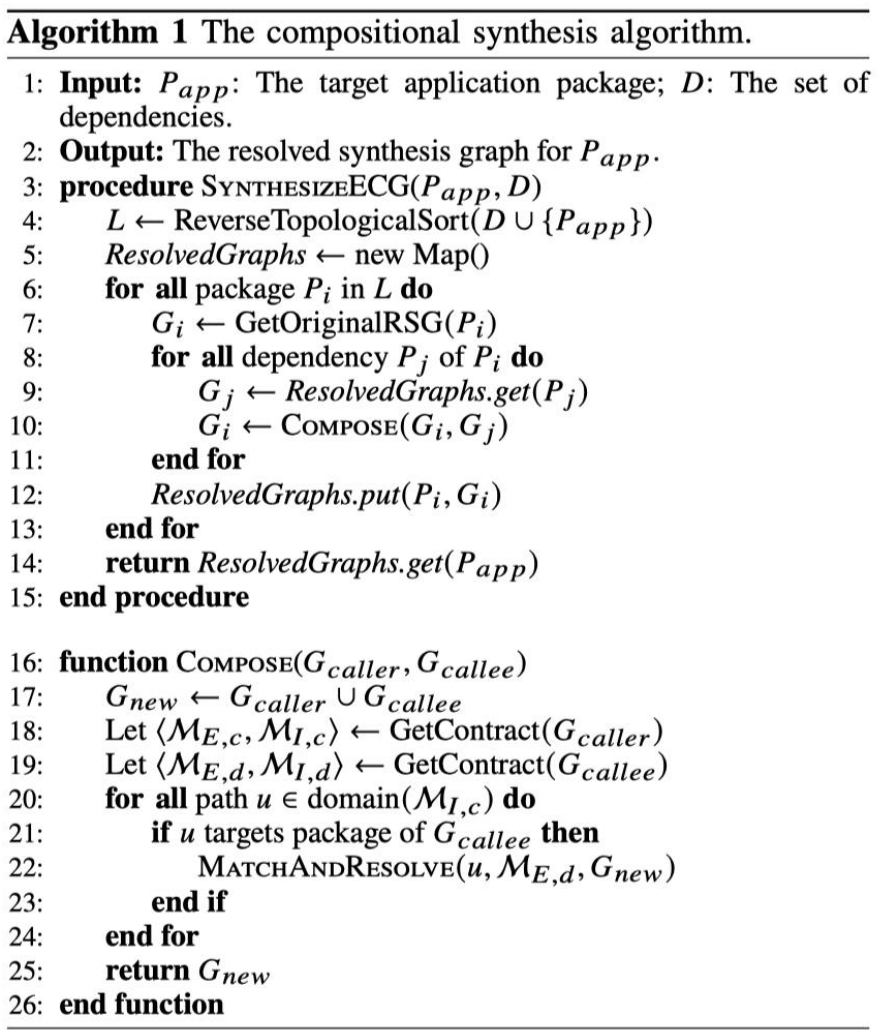
**Strategy:**

1. **Topological sort** of dependency graph
2. **Bottom-up processing:** leaves first
3. **For each package:**
    - Load pre-computed RSG from cache
    - Compose with already-resolved dependency graphs

**Interface stitching:**

- Match use paths (Import Manifest) to definition paths (Export Manifold)
- Create cross-package call edges
- Handle both direct calls and transitive resolution

**Result:** Precise, on-demand call graphs at ecosystem scale



**Algorithm 1** The compositional synthesis algorithm.

```
Input: $P_{app}$: The target application package; $D$: The set of dependencies.
Output: The resolved synthesis graph for $P_{app}$.
procedure SynthesizeECG($P_{app}, D$)
    $L \leftarrow$ ReverseTopologicalSort($D \cup \{P_{app}\}$)
    $ResolvedGraphs \leftarrow$ new Map()
    for all package $P_i$ in $L$ do
        $G_i \leftarrow$ GetOriginalRSG($P_i$)
        for all dependency $P_j$ of $P_i$ do
            $G_j \leftarrow ResolvedGraphs.get(P_j)$
            $G_i \leftarrow$ Compose($G_i, G_j$)
        end for
        $ResolvedGraphs.put(P_i, G_i)$
    end for
    return $ResolvedGraphs.get(P_{app})$
end procedure

function Compose($G_{caller}, G_{callee}$)
    $G_{new} \leftarrow G_{caller} \cup G_{callee}$
    Let $\langle \mathcal{M}_{E,c}, \mathcal{M}_{I,c} \rangle \leftarrow$ GetContract($G_{caller}$)
    Let $\langle \mathcal{M}_{E,d}, \mathcal{M}_{I,d} \rangle \leftarrow$ GetContract($G_{callee}$)
    for all path $u \in$ domain($\mathcal{M}_{I,c}$) do
        if $u$ targets package of $G_{callee}$ then
            MatchAndResolve($u, \mathcal{M}_{E,d}, G_{new}$)
        end if
    end for
    return $G_{new}$
end function
```

# RQ1 - Call Graph Accuracy

**Evaluation Setup:**

- 7 projects with 100% test coverage
- Dynamic ground truth (NodeProf + GraalVM)
- Compare against Jelly (state-of-the-art)

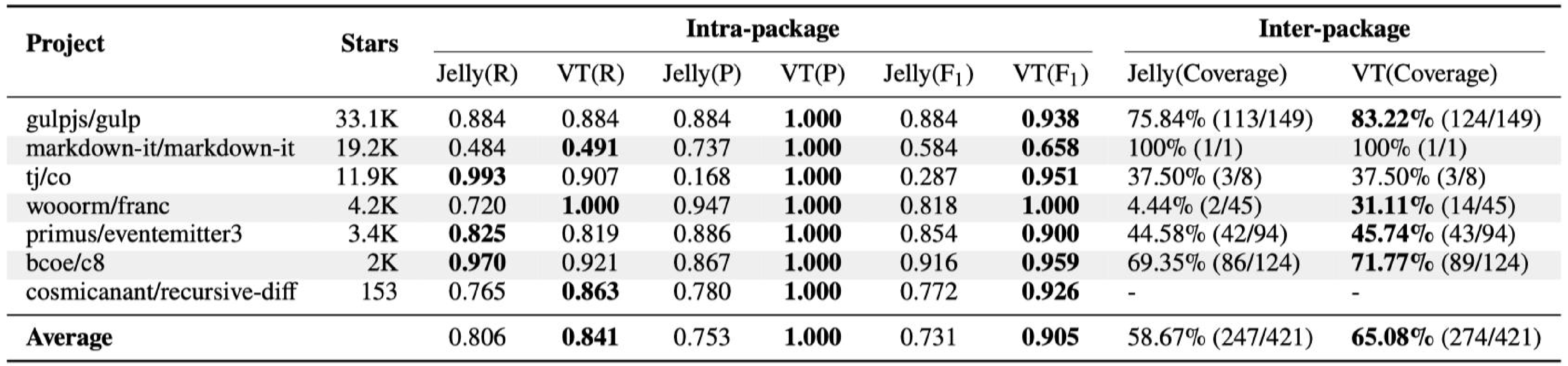
**Key Results:**

- VulTracer: F1 score of 0.905
- Perfect precision: 1.000 (zero false positives)
- Superior recall: 0.841 vs Jelly's 0.806
- Inter-package coverage: 65.08% vs Jelly's 58.67%

# RQ1 - Call Graph Accuracy

**Evaluation Setup:**

TABLE I: Evaluation of intra- and inter-package call resolution accuracy across different tools. VT denotes VULTRACER.



| Project | Stars | Intra-package | | | | | | Inter-package | |
|---|---|---|---|---|---|---|---|---|---|
| | | Jelly(R) | VT(R) | Jelly(P) | VT(P) | Jelly($F_1$) | VT($F_1$) | Jelly(Coverage) | VT(Coverage) |
| gulpjs/gulp | 33.1K | 0.884 | 0.884 | 0.884 | **1.000** | 0.884 | **0.938** | 75.84% (113/149) | **83.22%** (124/149) |
| markdown-it/markdown-it | 19.2K | 0.484 | **0.491** | 0.737 | **1.000** | 0.584 | **0.658** | 100% (1/1) | 100% (1/1) |
| tj/co | 11.9K | **0.993** | 0.907 | 0.168 | **1.000** | 0.287 | **0.951** | 37.50% (3/8) | 37.50% (3/8) |
| wooorm/franc | 4.2K | 0.720 | **1.000** | 0.947 | **1.000** | 0.818 | **1.000** | 4.44% (2/45) | **31.11%** (14/45) |
| primus/eventemitter3 | 3.4K | **0.825** | 0.819 | 0.886 | **1.000** | 0.854 | **0.900** | 44.58% (42/94) | **45.74%** (43/94) |
| bcoe/c8 | 2K | **0.970** | 0.921 | 0.867 | **1.000** | 0.916 | **0.959** | 69.35% (86/124) | **71.77%** (89/124) |
| cosmicanant/recursive-diff | 153 | 0.765 | **0.863** | 0.780 | **1.000** | 0.772 | **0.926** | - | - |
| **Average** | | 0.806 | **0.841** | 0.753 | **1.000** | 0.731 | **0.905** | 58.67% (247/421) | **65.08%** (274/421) |

**Bold** values indicate the superior result in each comparison pair. **R** denotes Recall, and **P** denotes Precision. Coverage is shown as (covered items/total items) percentage.

## RQ2 - Scalability Wins

**Performance Comparison on 99 Real Projects (CVE-2023-32314):**

**Jelly (Monolithic):**

- Success rate: **37.37%** (37/99 packages)
- Majority fail: out-of-memory errors
- Must re-analyze **26,653 dependencies** for each project
- Average time: 31.34 minutes (for successful cases only)

**VulTracer (Compositional):**

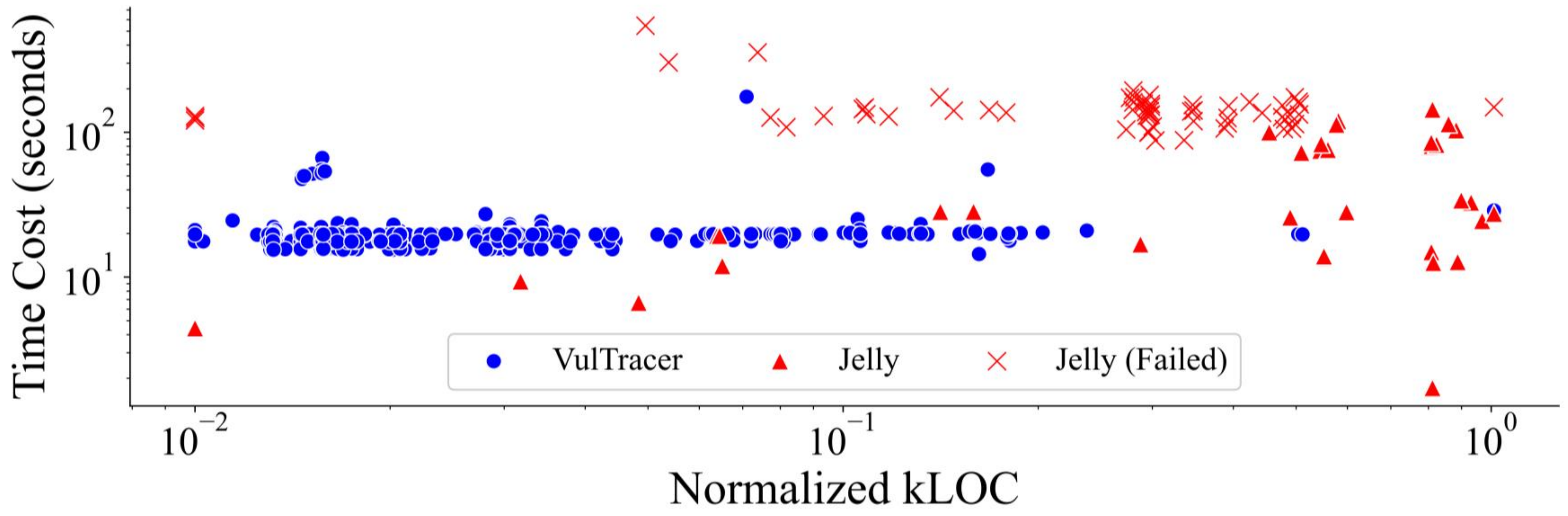
- **Success rate:** *99.41%* (503/506 packages)
- *98% reduction* in analysis scope (26,653 → 506 unique packages)
- One-time pre-computation: 174 minutes (cached for reuse)
- On-demand synthesis: **41.87 seconds** per project

**Key Advantage:** Time scales with unique packages, not project complexity

# RQ2 - Scalability Wins

**Performance Comparison on 99 Real Projects (CVE-2023-32314):**

**J**



**V**

Fig. 3: Time consumption comparison of different tools.

- On-demand synthesis: **41.87 seconds** per project

**Key Advantage:** Time scales with unique packages, not project complexity

# RQ3 - False Positive Reduction

**Real-World Vulnerability Auditing**

**Dataset:** 12 applications (from JAM benchmark)

**npm audit baseline:**

- 532 vulnerable packages reported
- 75 vulnerability propagation paths

**VulTracer Results:**

- 532 packages → 53 alarms (49 unique vulnerabilities)
- Manual verification: Only **21 True Positives**
- **94%** false positive reduction  VS  npm audit
- **Better than prior work:** JAM achieved 81% reduction

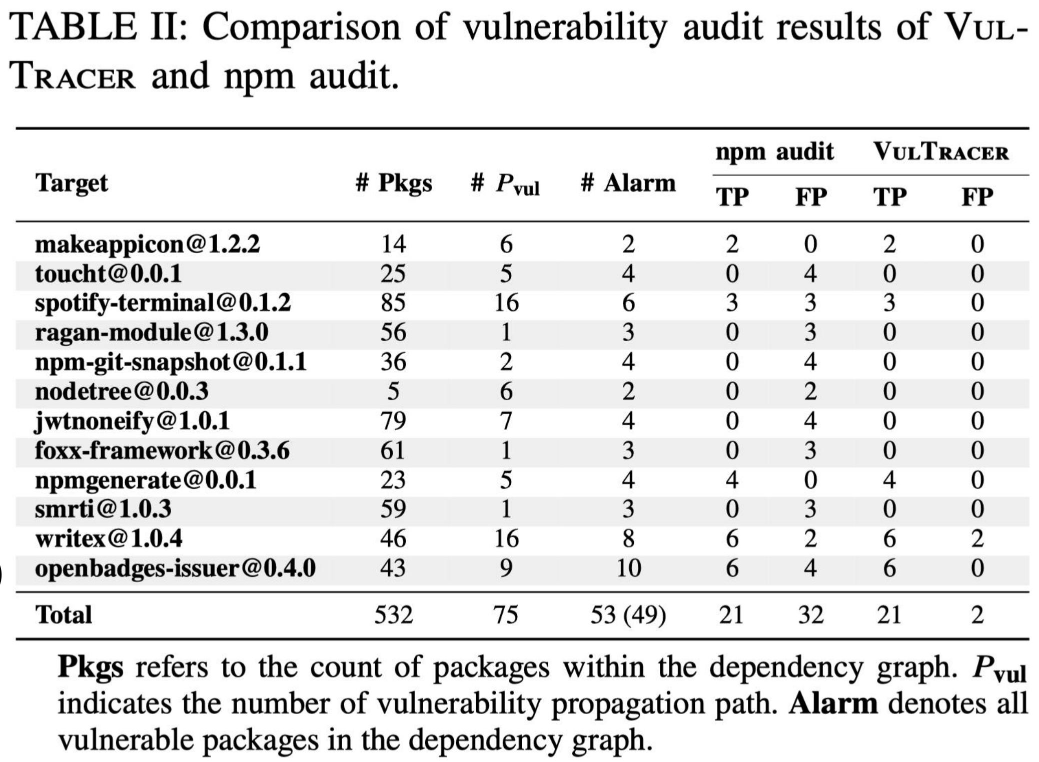


TABLE II: Comparison of vulnerability audit results of VulTracer and npm audit.

| Target | # Pkgs | # $P_{vul}$ | # Alarm | npm audit | | VulTracer | |
|---|---|---|---|---|---|---|---|
| | | | | TP | FP | TP | FP |
| makeappicon@1.2.2 | 14 | 6 | 2 | 2 | 0 | 2 | 0 |
| toucht@0.0.1 | 25 | 5 | 4 | 0 | 4 | 0 | 0 |
| spotify-terminal@0.1.2 | 85 | 16 | 6 | 3 | 3 | 3 | 0 |
| ragan-module@1.3.0 | 56 | 1 | 3 | 0 | 3 | 0 | 0 |
| npm-git-snapshot@0.1.1 | 36 | 2 | 4 | 0 | 4 | 0 | 0 |
| nodetree@0.0.3 | 5 | 6 | 2 | 0 | 2 | 0 | 0 |
| jwtnoneify@1.0.1 | 79 | 7 | 4 | 0 | 4 | 0 | 0 |
| foxx-framework@0.3.6 | 61 | 1 | 3 | 0 | 3 | 0 | 0 |
| npmgenerate@0.0.1 | 23 | 5 | 4 | 4 | 0 | 4 | 0 |
| smrti@1.0.3 | 59 | 1 | 3 | 0 | 3 | 0 | 0 |
| writex@1.0.4 | 46 | 16 | 8 | 6 | 2 | 6 | 2 |
| openbadges-issuer@0.4.0 | 43 | 9 | 10 | 6 | 4 | 6 | 0 |
| Total | 532 | 75 | 53 (49) | 21 | 32 | 21 | 2 |

**Pkgs** refers to the count of packages within the dependency graph. $P_{vul}$ indicates the number of vulnerability propagation path. **Alarm** denotes all vulnerable packages in the dependency graph.

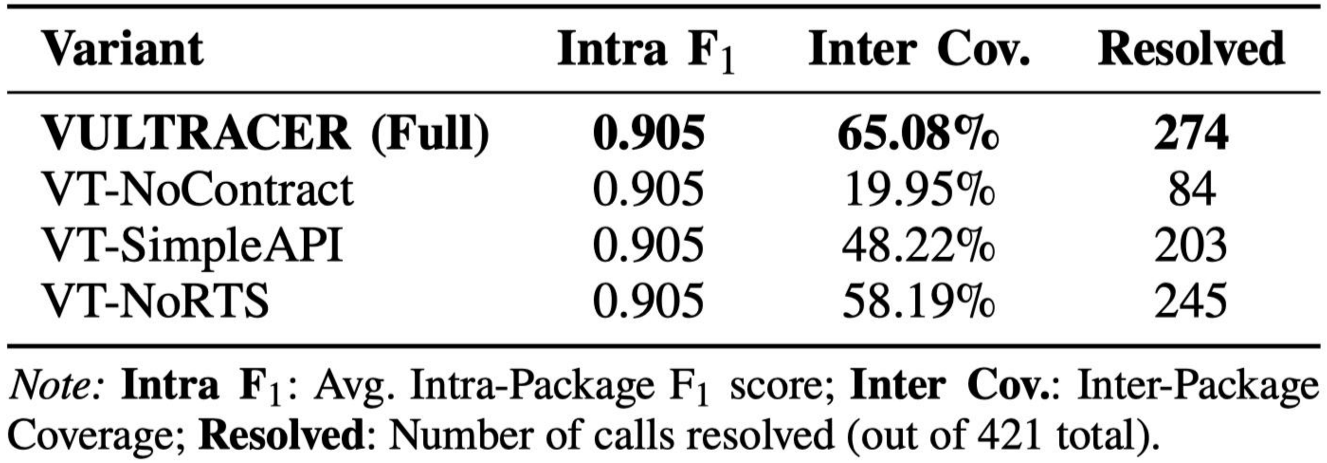
**Impact:** Dramatically reduces alert fatigue while maintaining accuracy

## RQ4 - Ablation Study

**Variants:**

- **VulTracer-Full:** Complete system
- **VT-NoContract:** Removes formal contracts (uses name matching)
- **VT-SimpleAPI:** Restricts API vocabulary (removes getReturn, getParameter, getInstance)
- **VT-NoRTS:** Disables Reverse Topological Sort

TABLE III: Ablation study results for VULTRACER.



| Variant | Intra $F_1$ | Inter Cov. | Resolved |
|---|---|---|---|
| **VULTRACER (Full)** | **0.905** | **65.08%** | **274** |
| VT-NoContract | 0.905 | 19.95% | 84 |
| VT-SimpleAPI | 0.905 | 48.22% | 203 |
| VT-NoRTS | 0.905 | 58.19% | 245 |

*Note:* **Intra $F_1$**: Avg. Intra-Package $F_1$ score; **Inter Cov.**: Inter-Package Coverage; **Resolved**: Number of calls resolved (out of 421 total).

# Ecosystem-Scale Study Datasets

**DS$_{\text{npm}}$:** *Complete npm ecosystem*

- 3,267,273 unique packages
- 34,685,976 distinct versions
- 900+ million dependency links
- Data collected through December 31, 2024

**DS$_{\text{CVE}}$:** *Two-dimensional vulnerability selection*

1. High-impact: 6 CVEs from top 10 most downloaded packages
   - lodash, debug, semver, minimatch
2. Diversity: 21 CVEs aligned with 2024 CWE Top-25
   - Covering injection, prototype pollution, path traversal, etc.

**Total:** 27 unique CVEs with precisely identified vulnerable functions

## Ecosystem-Scale Study Datase

**DS$_{npm}$:** *Complete npm ecosystem*

- 3,267,273 unique packages

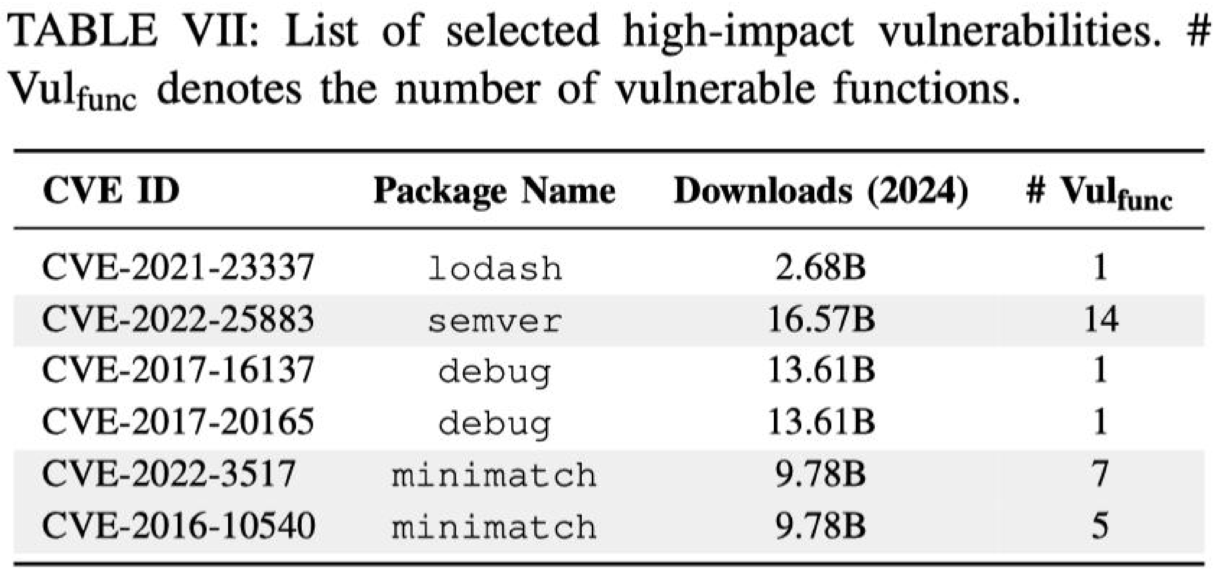


TABLE VII: List of selected high-impact vulnerabilities. # Vul$_{func}$ denotes the number of vulnerable functions.

| CVE ID | Package Name | Downloads (2024) | # Vul$_{func}$ |
|---|---|---|---|
| CVE-2021-23337 | `lodash` | 2.68B | 1 |
| CVE-2022-25883 | `semver` | 16.57B | 14 |
| CVE-2017-16137 | `debug` | 13.61B | 1 |
| CVE-2017-20165 | `debug` | 13.61B | 1 |
| CVE-2022-3517 | `minimatch` | 9.78B | 7 |
| CVE-2016-10540 | `minimatch` | 9.78B | 5 |

2. Diversity: 21 CVEs aligned with 2024 CWE Top-

   - Covering injection, prototype pollution, path tr

**Total:** 27 unique CVEs with precisely identifie

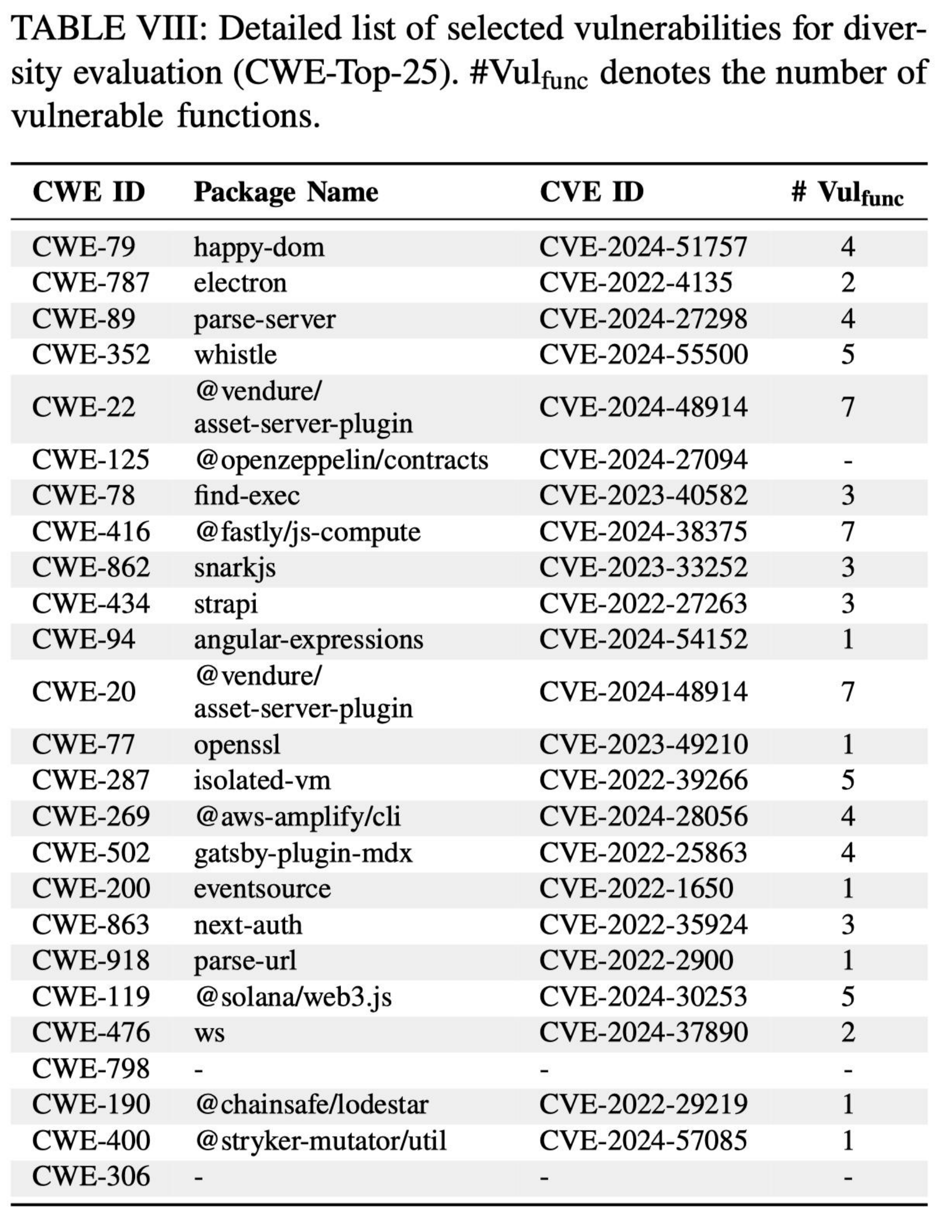


TABLE VIII: Detailed list of selected vulnerabilities for diversity evaluation (CWE-Top-25). #Vul$_{func}$ denotes the number of vulnerable functions.

| CWE ID | Package Name | CVE ID | # Vul$_{func}$ |
|---|---|---|---|
| CWE-79 | happy-dom | CVE-2024-51757 | 4 |
| CWE-787 | electron | CVE-2022-4135 | 2 |
| CWE-89 | parse-server | CVE-2024-27298 | 4 |
| CWE-352 | whistle | CVE-2024-55500 | 5 |
| CWE-22 | @vendure/ asset-server-plugin | CVE-2024-48914 | 7 |
| CWE-125 | @openzeppelin/contracts | CVE-2024-27094 | - |
| CWE-78 | find-exec | CVE-2023-40582 | 3 |
| CWE-416 | @fastly/js-compute | CVE-2024-38375 | 7 |
| CWE-862 | snarkjs | CVE-2023-33252 | 3 |
| CWE-434 | strapi | CVE-2022-27263 | 3 |
| CWE-94 | angular-expressions | CVE-2024-54152 | 1 |
| CWE-20 | @vendure/ asset-server-plugin | CVE-2024-48914 | 7 |
| CWE-77 | openssl | CVE-2023-49210 | 1 |
| CWE-287 | isolated-vm | CVE-2022-39266 | 5 |
| CWE-269 | @aws-amplify/cli | CVE-2024-28056 | 4 |
| CWE-502 | gatsby-plugin-mdx | CVE-2022-25863 | 4 |
| CWE-200 | eventsource | CVE-2022-1650 | 1 |
| CWE-863 | next-auth | CVE-2022-35924 | 3 |
| CWE-918 | parse-url | CVE-2022-2900 | 1 |
| CWE-119 | @solana/web3.js | CVE-2024-30253 | 5 |
| CWE-476 | ws | CVE-2024-37890 | 2 |
| CWE-798 | - | - | - |
| CWE-190 | @chainsafe/lodestar | CVE-2022-29219 | 1 |
| CWE-400 | @stryker-mutator/util | CVE-2024-57085 | 1 |
| CWE-306 | - | - | - |

## RQ5 - Over-Approximation

**RQ5: How Much Do Package-Level Alerts Over-Approximate?**

**Analysis:** 27 CVEs, 703,896 Direct Dependents (d1)

**Key Finding:**

- **Single-hop:** *67.51%* global over-approximation
- **Multi-hop (transitive):** *68.28%* of package-level alerts are false positives
- Only **32.49%** of flagged packages actually reach vulnerable functions

**Implication:** *Package-level tools create massive noise*

- 2 out of 3 alerts don't represent real threats
- Alert fatigue is a solvable technical problem

TABLE IX: Comprehensive single-hop reachability analysis merging High-Impact and Diversity datasets. The global average is weighted based on the number of CVEs in each dataset.

| **Dimension** | **CVE ID** | **Package** | **#$d_0$** | **#$d_1$** | **#$C_{\text{mod}}$** | **#$C_{\text{func}}$** | **#$C_{\text{vuln\_func}}$** |
|---|---|---|---|---|---|---|---|
| High-Impact | CVE-2021-23337 | `lodash` | 100 | 396,112 | 264,179 (66.69%) | 244,130 (61.63%) | 11,574 (2.92%) |
| | CVE-2022-3517 | `minimatch` | 26 | 38,112 | 28,667 (75.22%) | 15,791 (41.43%) | 15,791 (41.43%) |
| | CVE-2016-10540 | `minimatch` | 23 | 10,341 | 9,211 (89.07%) | 3,528 (34.12%) | 3,528 (34.12%) |
| | CVE-2022-25883 | `semver` | 74 | 139,257 | 111,138 (79.81%) | 102,209 (73.40%) | 73,314 (52.65%) |
| | CVE-2017-16137 | `debug` | 55 | 70,297 | 54,098 (76.96%) | 51,425 (73.15%) | 50,454 (71.77%) |
| | CVE-2017-20165 | `debug` | 42 | 39,365 | 29,702 (75.45%) | 29,583 (75.15%) | 29,576 (75.13%) |
| Diversity | CVE-2022-1650 | `eventsource` | 17 | 167 | 109 (65.27%) | 100 (59.88%) | 100 (59.88%) |
| | CVE-2022-25863 | `gatsby-plugin-mdx` | 125 | 610 | 286 (46.89%) | 0 (0.00%) | 0 (0.00%) |
| | CVE-2022-27263 | `strapi` | 16 | 30 | 3 (10.00%) | 3 (10.00%) | 3 (10.00%) |
| | CVE-2022-2900 | `parse-url` | 11 | 204 | 67 (32.84%) | 63 (30.88%) | 63 (30.88%) |
| | CVE-2022-29219 | `@chainsafe/lodestar` | 23 | 23 | 17 (73.91%) | 11 (47.83%) | 0 (0.00%) |
| | CVE-2022-35924 | `next-auth` | 17 | 58 | 34 (58.62%) | 10 (17.24%) | 9 (15.52%) |
| | CVE-2022-39266 | `isolated-vm` | 15 | 38 | 25 (65.79%) | 25 (65.79%) | 25 (65.79%) |
| | CVE-2022-4135 | `electron` | 504 | 2,453 | 1,978 (80.64%) | 1,816 (74.03%) | 1,816 (74.03%) |
| | CVE-2023-33252 | `snarkjs` | 27 | 309 | 243 (78.64%) | 220 (71.20%) | 148 (47.90%) |
| | CVE-2023-40582 | `find-exec` | 8 | 11 | 11 (100.00%) | 11 (100.00%) | 11 (100.00%) |
| | CVE-2023-49210 | `openssl` | 2 | 56 | 0 (0.00%) | 0 (0.00%) | 0 (0.00%) |
| | CVE-2024-27298 | `parse-server` | 9 | 27 | 13 (48.15%) | 8 (29.63%) | 8 (29.63%) |
| | CVE-2024-28056 | `@aws-amplify/cli` | 5 | 13 | 0 (0.00%) | 0 (0.00%) | 0 (0.00%) |
| | CVE-2024-30253 | `@solana/web3.js` | 109 | 428 | 400 (93.46%) | 381 (89.02%) | 168 (39.25%) |
| | CVE-2024-37890 | `ws` | 86 | 4,080 | 3,163 (77.52%) | 2,389 (58.55%) | 1,561 (38.26%) |
| | CVE-2024-38375 | `@fastly/js-compute` | 24 | 24 | 0 (0.00%) | 0 (0.00%) | 0 (0.00%) |
| | CVE-2024-48914 | `@vendure/asset-server-plugin` | 38 | 41 | 0 (0.00%) | 5 (12.20%) | 5 (12.20%) |
| | CVE-2024-51757 | `happy-dom` | 150 | 452 | 252 (55.75%) | 257 (56.86%) | 13 (2.88%) |
| | CVE-2024-54152 | `angular-expressions` | 8 | 65 | 44 (67.69%) | 42 (64.62%) | 41 (63.08%) |
| | CVE-2024-55500 | `whistle` | 7 | 27 | 13 (48.15%) | 13 (48.15%) | 0 (0.00%) |
| | CVE-2024-57085 | `@stryker-mutator/util` | 80 | 640 | 588 (91.88%) | 546 (85.31%) | 64 (10.00%) |
| **Average** | | | - | - | **57.72%** | **45.71%** | **32.49%** |

# Understanding Attenuation - API Surface

## Why Such Different Propagation Rates?

### Factor 1: API Breadth

**- Broad API (lodash):**

- 242 functions available
- template (vulnerable): rank #49, only 0.30% of calls
- Top functions: forEach (10.66%), isFunction (9.99%) are safe
- **Result:** 2.92% propagation (shallow usage)
- Long-tail distribution limits impact

**- Narrow API (debug):**

- Focused on core debugging functionality
- 98% of importers use the vulnerable function
- **Result:** 71.77% propagation (deep usage)
- Limited alternative functions drive high usage

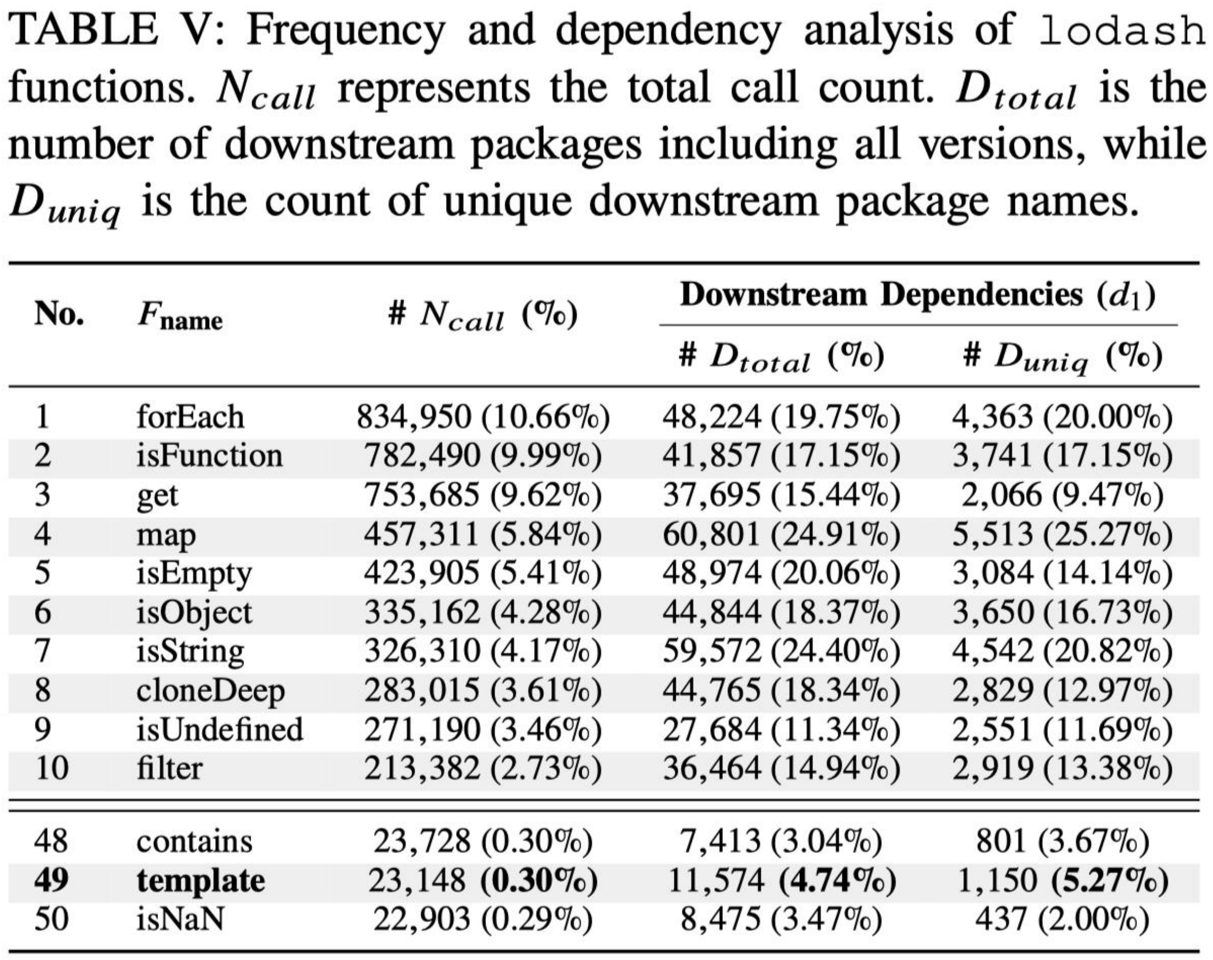


TABLE V: Frequency and dependency analysis of `lodash` functions. $N_{call}$ represents the total call count. $D_{total}$ is the number of downstream packages including all versions, while $D_{uniq}$ is the count of unique downstream package names.

| No. | $F_{\text{name}}$ | # $N_{call}$ (%) | Downstream Dependencies ($d_1$) # $D_{total}$ (%) | Downstream Dependencies ($d_1$) # $D_{uniq}$ (%) |
|---|---|---|---|---|
| 1 | forEach | 834,950 (10.66%) | 48,224 (19.75%) | 4,363 (20.00%) |
| 2 | isFunction | 782,490 (9.99%) | 41,857 (17.15%) | 3,741 (17.15%) |
| 3 | get | 753,685 (9.62%) | 37,695 (15.44%) | 2,066 (9.47%) |
| 4 | map | 457,311 (5.84%) | 60,801 (24.91%) | 5,513 (25.27%) |
| 5 | isEmpty | 423,905 (5.41%) | 48,974 (20.06%) | 3,084 (14.14%) |
| 6 | isObject | 335,162 (4.28%) | 44,844 (18.37%) | 3,650 (16.73%) |
| 7 | isString | 326,310 (4.17%) | 59,572 (24.40%) | 4,542 (20.82%) |
| 8 | cloneDeep | 283,015 (3.61%) | 44,765 (18.34%) | 2,829 (12.97%) |
| 9 | isUndefined | 271,190 (3.46%) | 27,684 (11.34%) | 2,551 (11.69%) |
| 10 | filter | 213,382 (2.73%) | 36,464 (14.94%) | 2,919 (13.38%) |
| 48 | contains | 23,728 (0.30%) | 7,413 (3.04%) | 801 (3.67%) |
| **49** | **template** | 23,148 **(0.30%)** | 11,574 **(4.74%)** | 1,150 **(5.27%)** |
| 50 | isNaN | 22,903 (0.29%) | 8,475 (3.47%) | 437 (2.00%) |

**Insight:** Vulnerability impact correlates with API specificity

# Understanding Attenuation - Unused Dependencies

## Factor 2: Unused Dependencies

**Critical Finding:**

- 22.80% of high-impact direct dependents (d1), 42.28% globally
- Never import the dependency at all (fail C_mod condition)

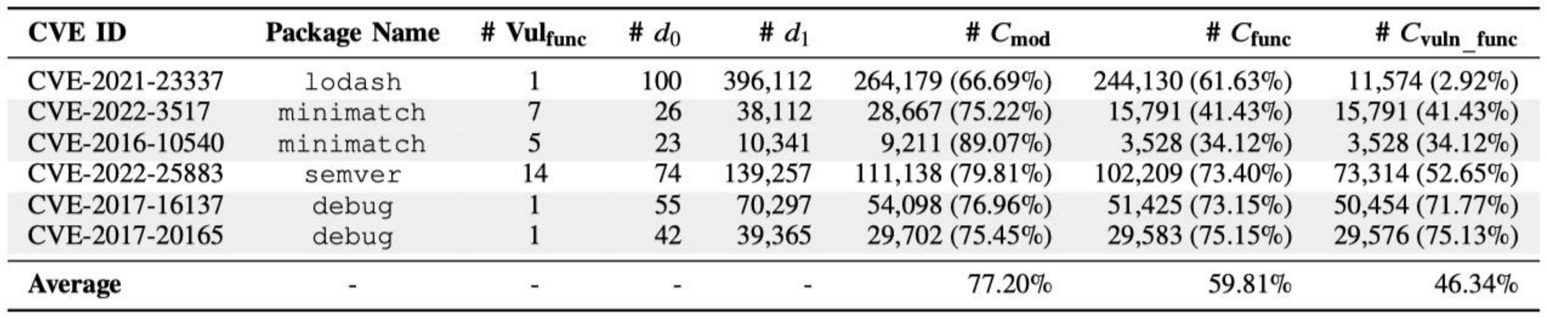
**Examples:**

- CVE-2021-23337 (lodash): 131,933 packages declare but never use
- Declared in package.json but no actual import statements

**Implication:** Instant false positive

- Presence in dependency graph ≠ actual usage

TABLE IV: Single-hop reachability analysis and attenuation for High-impact vulnerabilities. The full detailed results for all 27 CVEs are provided in Table IX in the Appendix.



| CVE ID | Package Name | # $\text{Vul}_{\text{func}}$ | # $d_0$ | # $d_1$ | # $C_{\text{mod}}$ | # $C_{\text{func}}$ | # $C_{\text{vuln\_func}}$ |
|---|---|---|---|---|---|---|---|
| CVE-2021-23337 | `lodash` | 1 | 100 | 396,112 | 264,179 (66.69%) | 244,130 (61.63%) | 11,574 (2.92%) |
| CVE-2022-3517 | `minimatch` | 7 | 26 | 38,112 | 28,667 (75.22%) | 15,791 (41.43%) | 15,791 (41.43%) |
| CVE-2016-10540 | `minimatch` | 5 | 23 | 10,341 | 9,211 (89.07%) | 3,528 (34.12%) | 3,528 (34.12%) |
| CVE-2022-25883 | `semver` | 14 | 74 | 139,257 | 111,138 (79.81%) | 102,209 (73.40%) | 73,314 (52.65%) |
| CVE-2017-16137 | `debug` | 1 | 55 | 70,297 | 54,098 (76.96%) | 51,425 (73.15%) | 50,454 (71.77%) |
| CVE-2017-20165 | `debug` | 1 | 42 | 39,365 | 29,702 (75.45%) | 29,583 (75.15%) | 29,576 (75.13%) |
| **Average** | - | - | - | - | 77.20% | 59.81% | 46.34% |

# Multi-Hop Propagation Decay

## RQ6: How Does Impact Attenuate Transitively?

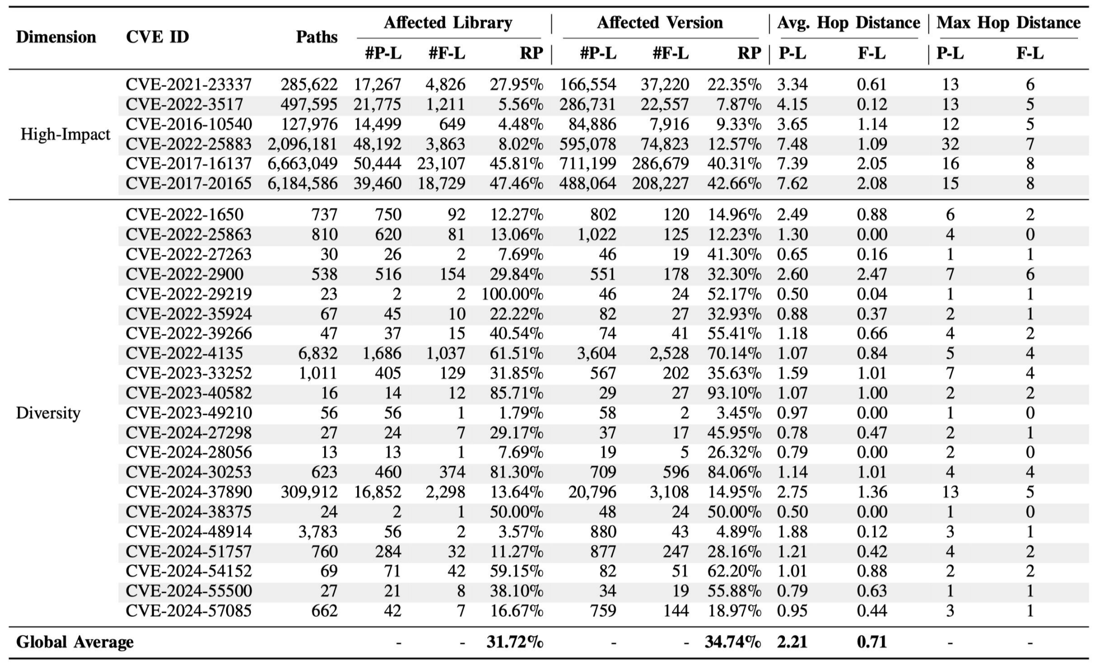
## Key Findings:

- Average: *68.28%* of transitive alerts are false positives
- True impact is shallow and localized

## Path Depth Comparison:

- **Package-level:**
  - Average: 2.21 hops
  - Maximum: 32 hops
  - Appears to propagate deep
- **Function-level:**
  - Average: 0.71 hops
  - Maximum: 8 hops
  - Actually very shallow

TABLE X: Comparative analysis of vulnerability propagation at package and function levels (Multi-hop). The table merges High-Impact and Diversity datasets. **RP** denotes the relative proportion of Function-level to Package-level.



| Dimension | CVE ID | Paths | Affected Library | | | Affected Version | | | Avg. Hop Distance | | Max Hop Distance | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | #P-L | #F-L | RP | #P-L | #F-L | RP | P-L | F-L | P-L | F-L |
| High-Impact | CVE-2021-23337 | 285,622 | 17,267 | 4,826 | 27.95% | 166,554 | 37,220 | 22.35% | 3.34 | 0.61 | 13 | 6 |
| | CVE-2022-3517 | 497,595 | 21,775 | 1,211 | 5.56% | 286,731 | 22,557 | 7.87% | 4.15 | 0.12 | 13 | 5 |
| | CVE-2016-10540 | 127,976 | 14,499 | 649 | 4.48% | 84,886 | 7,916 | 9.33% | 3.65 | 1.14 | 12 | 5 |
| | CVE-2022-25883 | 2,096,181 | 48,192 | 3,863 | 8.02% | 595,078 | 74,823 | 12.57% | 7.48 | 1.09 | 32 | 7 |
| | CVE-2017-16137 | 6,663,049 | 50,444 | 23,107 | 45.81% | 711,199 | 286,679 | 40.31% | 7.39 | 2.05 | 16 | 8 |
| | CVE-2017-20165 | 6,184,586 | 39,460 | 18,729 | 47.46% | 488,064 | 208,227 | 42.66% | 7.62 | 2.08 | 15 | 8 |
| Diversity | CVE-2022-1650 | 737 | 750 | 92 | 12.27% | 802 | 120 | 14.96% | 2.49 | 0.88 | 6 | 2 |
| | CVE-2022-25863 | 810 | 620 | 81 | 13.06% | 1,022 | 125 | 12.23% | 1.30 | 0.00 | 4 | 0 |
| | CVE-2022-27263 | 30 | 26 | 2 | 7.69% | 46 | 19 | 41.30% | 0.65 | 0.16 | 1 | 1 |
| | CVE-2022-2900 | 538 | 516 | 154 | 29.84% | 551 | 178 | 32.30% | 2.60 | 2.47 | 7 | 6 |
| | CVE-2022-29219 | 23 | 2 | 2 | 100.00% | 46 | 24 | 52.17% | 0.50 | 0.04 | 1 | 1 |
| | CVE-2022-35924 | 67 | 45 | 10 | 22.22% | 82 | 27 | 32.93% | 0.88 | 0.37 | 2 | 1 |
| | CVE-2022-39266 | 47 | 37 | 15 | 40.54% | 74 | 41 | 55.41% | 1.18 | 0.66 | 4 | 2 |
| | CVE-2022-4135 | 6,832 | 1,686 | 1,037 | 61.51% | 3,604 | 2,528 | 70.14% | 1.07 | 0.84 | 5 | 4 |
| | CVE-2023-33252 | 1,011 | 405 | 129 | 31.85% | 567 | 202 | 35.63% | 1.59 | 1.01 | 7 | 4 |
| | CVE-2023-40582 | 16 | 14 | 12 | 85.71% | 29 | 27 | 93.10% | 1.07 | 1.00 | 2 | 2 |
| | CVE-2023-49210 | 56 | 56 | 1 | 1.79% | 58 | 2 | 3.45% | 0.97 | 0.00 | 1 | 0 |
| | CVE-2024-27298 | 27 | 24 | 7 | 29.17% | 37 | 17 | 45.95% | 0.78 | 0.47 | 2 | 1 |
| | CVE-2024-28056 | 13 | 13 | 1 | 7.69% | 19 | 5 | 26.32% | 0.79 | 0.00 | 2 | 0 |
| | CVE-2024-30253 | 623 | 460 | 374 | 81.30% | 709 | 596 | 84.06% | 1.14 | 1.01 | 4 | 4 |
| | CVE-2024-37890 | 309,912 | 16,852 | 2,298 | 13.64% | 20,796 | 3,108 | 14.95% | 2.75 | 1.36 | 13 | 5 |
| | CVE-2024-38375 | 24 | 2 | 1 | 50.00% | 48 | 24 | 50.00% | 0.50 | 0.00 | 1 | 0 |
| | CVE-2024-48914 | 3,783 | 56 | 2 | 3.57% | 880 | 43 | 4.89% | 1.88 | 0.12 | 3 | 1 |
| | CVE-2024-51757 | 760 | 284 | 32 | 11.27% | 877 | 247 | 28.16% | 1.21 | 0.42 | 4 | 2 |
| | CVE-2024-54152 | 69 | 71 | 42 | 59.15% | 82 | 51 | 62.20% | 1.01 | 0.88 | 2 | 2 |
| | CVE-2024-55500 | 27 | 21 | 8 | 38.10% | 34 | 19 | 55.88% | 0.79 | 0.63 | 1 | 1 |
| | CVE-2024-57085 | 662 | 42 | 7 | 16.67% | 759 | 144 | 18.97% | 0.95 | 0.44 | 3 | 1 |
| **Global Average** | | | - | - | **31.72%** | - | - | **34.74%** | **2.21** | **0.71** | - | - |

**P-L** indicates package-level analysis, **F-L** indicates function-level analysis, and **RP** represents the relative proportion of **F-L** to **P-L**.

**Critical Result: 96.59%** of true impact identified **within 4 hops**

## Interpretation

**Interpretation:** *True vulnerability impact is shallow and localized*

**Key Insights:**

- Initial attenuation factors (unused deps, shallow API usage) compound at each hop
- Real vulnerabilities rarely propagate deep through dependency chains
- Most transitive propagation paths are noise, not signal
- Function-level analysis reveals true attack surface

**Practical Impact:**

- Focus remediation on nearby dependencies (≤4 hops)
- Prioritize direct and close transitive dependencies
- Long dependency chains rarely represent real threats

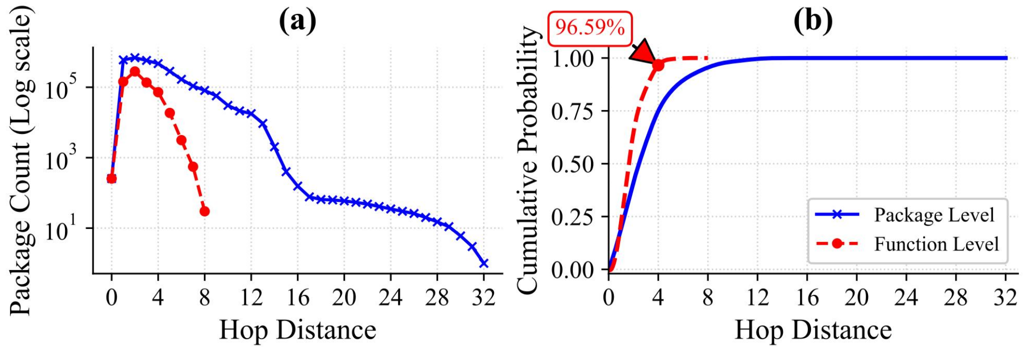


Fig. 4: Package- vs. Function-level propagation decay: (a) Count per hop; (b) CDF.

## Practical Implications

### For Developers and Security Teams:

- **Reachability-driven triage:** Prioritize by evidence of reachability, not just presence
- **Direct dependencies first:** Reachable vulnerabilities in direct deps warrant immediate action
- **Dependency hygiene:** Remove unused dependencies to reduce attack surface and noise
- **Strategic allocation:** Focus resources on genuine, reachable threats

### For SCA Tool Vendors:

- **Move beyond package-level:** Integrate function-level call-graph analysis
- **Reduce alert fatigue:** Distinguish presence from reachability
- **Actionable intelligence:** Provide precise, prioritized threat information
- **Transform value proposition:** From overwhelming reports to targeted insights

## Conclusions

***VulTracer*** *bridges the gap between precision and scalability:*

- **Addresses critical problem:** Alert fatigue from imprecise tools
- **Enables practical solution:** Function-level analysis at CI/CD speed
- **Provides actionable insights:** Focus on reachable, genuine threats
- **Transforms understanding:** Vulnerability propagation is shallower than believed

**From Noise to Signal:**

- Package-level analysis: ***68.28%*** noise
- Function-level analysis: Reveals true signal
- **Impact:** Developers can finally answer ***"Am I really affected?"***

**Future:** Path to more effective software supply chain security

# From Noise to Signal: Precisely Identify Affected Packages of Known Vulnerabilities in npm Ecosystem

Yingyuan Pu, Lingyun Ying, Yacong Gu

**puyingyuan01@qianxin.com**