On Dependency Analysis of NPM

A Survey

Guochang Wang
dz1933026@smail.nju.edu.cn

Institute of Computer Software
Nanjing University

I2EC Report, Oct 2020

Sections

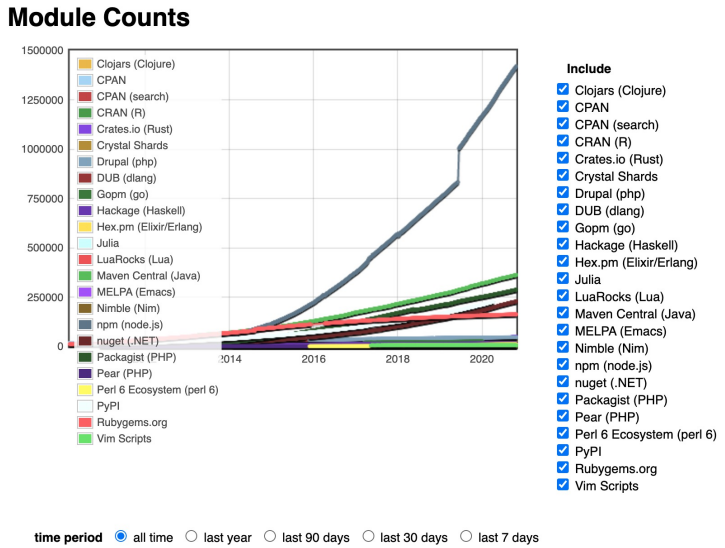
Sections

Node.js and Node Package Manager(NPM)

Node.js and Node Package Manager(NPM)

Developer publishes package

Other developers install the package

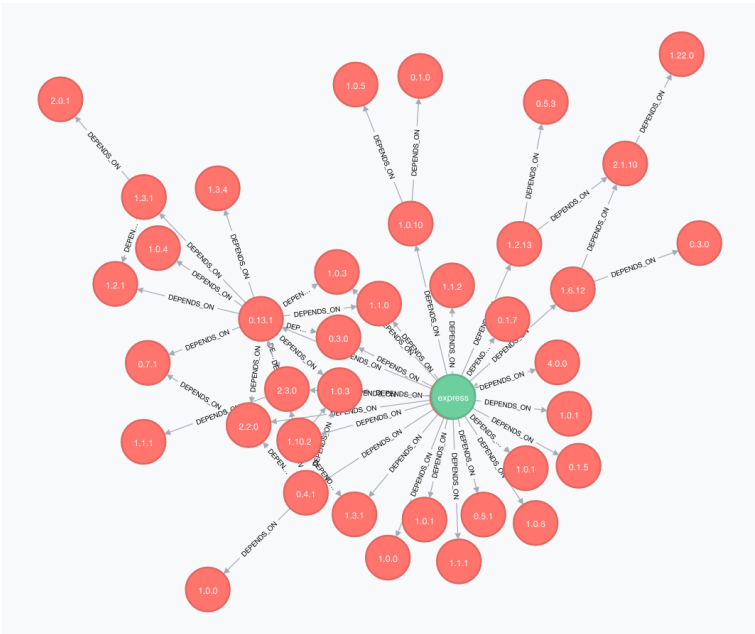


NPM Dependency Hell[1]

Definition

The frustrating problems that occur when **a software package depends on several other packages**.

- Many dependencies
- Long chains of dependencies
- Conflicting dependencies
- Circular dependencies
- Package manager dependencies
- Diamond dependency



[1] https://en.wikipedia.org/wiki/Dependency_hell

NPM Dependency Hell: Representative Incident[2]

Left-Pad Incident

Koçulu unpublished a NPM package named "Left-Pad", causing all its downstream packages to crash on installation, including Babel(over 16,000 Dependents by 2020.10.22) and Webpack(20,983 Dependents by 2020.10.22).

Now developers are forbidden from abandoning packages directly.
But vulnerable or malicious code in a single package may still affect thousands of others, e.g. through breaking API change.

[2] https://www.infoworld.com/article/3047177/javascript/how-one-yanked-javascript-package-wreaked-havoc.html

Sections

Small World with High Risks: A Study of Security Threats in the npm Ecosystem[3]

General Research Question

Are these security incidents unfortunate individual cases or first evidence of a more general problem?

Some Results

- Some highly popular packages reach more than 100,000 other packages, making them a prime target for attacks.
- Up to 40% of 5,386,239 versioned packages rely on code known to be vulnerable.

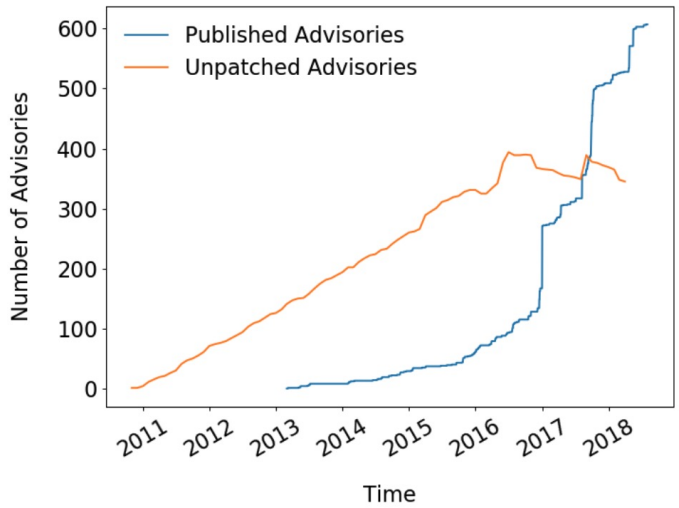


Figure 13: Evolution of the total and unpatched number of advisories.

[3]Small World with High Risks: A Study of Security Threats in the npm Ecosystem. USENIX Security 2019, Markus Zimmermann et al.

Small World with **High** Risks?

Mahmoud et al. conducted an empirical study involving 6,673 real-world, active, and mature open source Node.js applications.[4]

Core Insight

Our experience leads us to believe that, in the grand scheme of things, these software vulnerabilities may have less impact than what is reported.

Conclusion The findings show that although 67.93% of the examined applications depend on at least one vulnerable package, 94.91% of the vulnerable packages in those affected applications are classified as having low threat.

[4] On the Threat of npm Vulnerable Dependencies in Node.js Applications, arxiv, Mahmoud Alfadel et al.

Sections

Which Study is the Truth?

Threat to Validity

Both of the studies conducted their analysis at package level. Based on hosted vulnerability databases, e.g. Snyk[a], Rapid7[b], whose data is basically an aggregation of GitHub issues and bug reports from different package managers.

[a] https://snyk.io/
[b] https://www.rapid7.com/

Which Study is the Truth?

Threat to Validity

Both of the studies conducted their analysis at package level. Based on hosted vulnerability databases, e.g. Snyk[a], Rapid7[b], whose data is basically an aggregation of GitHub issues and bug reports from different package managers.

[a] https://snyk.io/
[b] https://www.rapid7.com/

Essentially, the studies give same results based on different **assumptions**.

- High Risk: if a package is vulnerable, then all it **dependents** are vulnerable.
- Low Threat: if a package is **not reported** as high threat, then it's **not vulnerable**.

Function Call Graph: Concrete Methodology

Vulnerability Comes from Functions: A package is vulnerable because of its **function(s)**, which you may **not** even called.

Research Problem

How to judge the correlation between vulnerability of a NPM project and its dependencies?

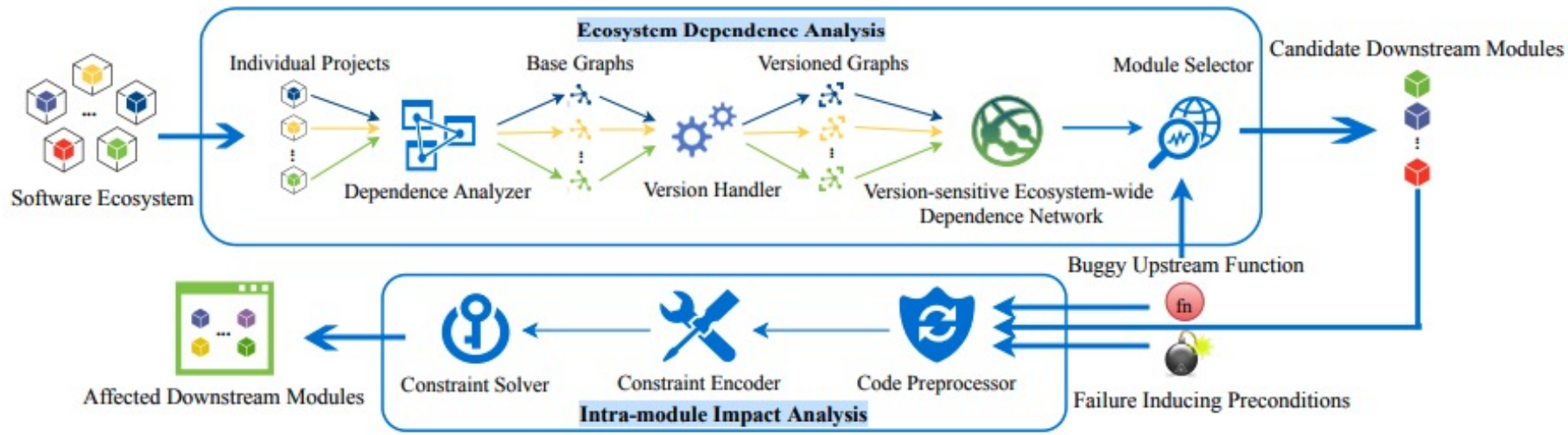
A fine-grained dependency network that goes beyond packages and into call graphs is needed.

Function Call Graph: Concrete Methodology

Actually, various analysis methodologies based on JavaScript function call graph was proposed before.

- Software Ecosystem Call Graph for Dependency Management, ICSE-NIER 18, Joseph et al.
- Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages, ICSME 18, Rodrigo et al.
- Static analysis of event-driven Node.js JavaScript applications, SIGPLAN Notices 15, Magnus et al.
- Efficient construction of approximate call graphs for JavaScript IDE services, ICSE 13, Asger et al.

Function Call Graph: Workflow[5]



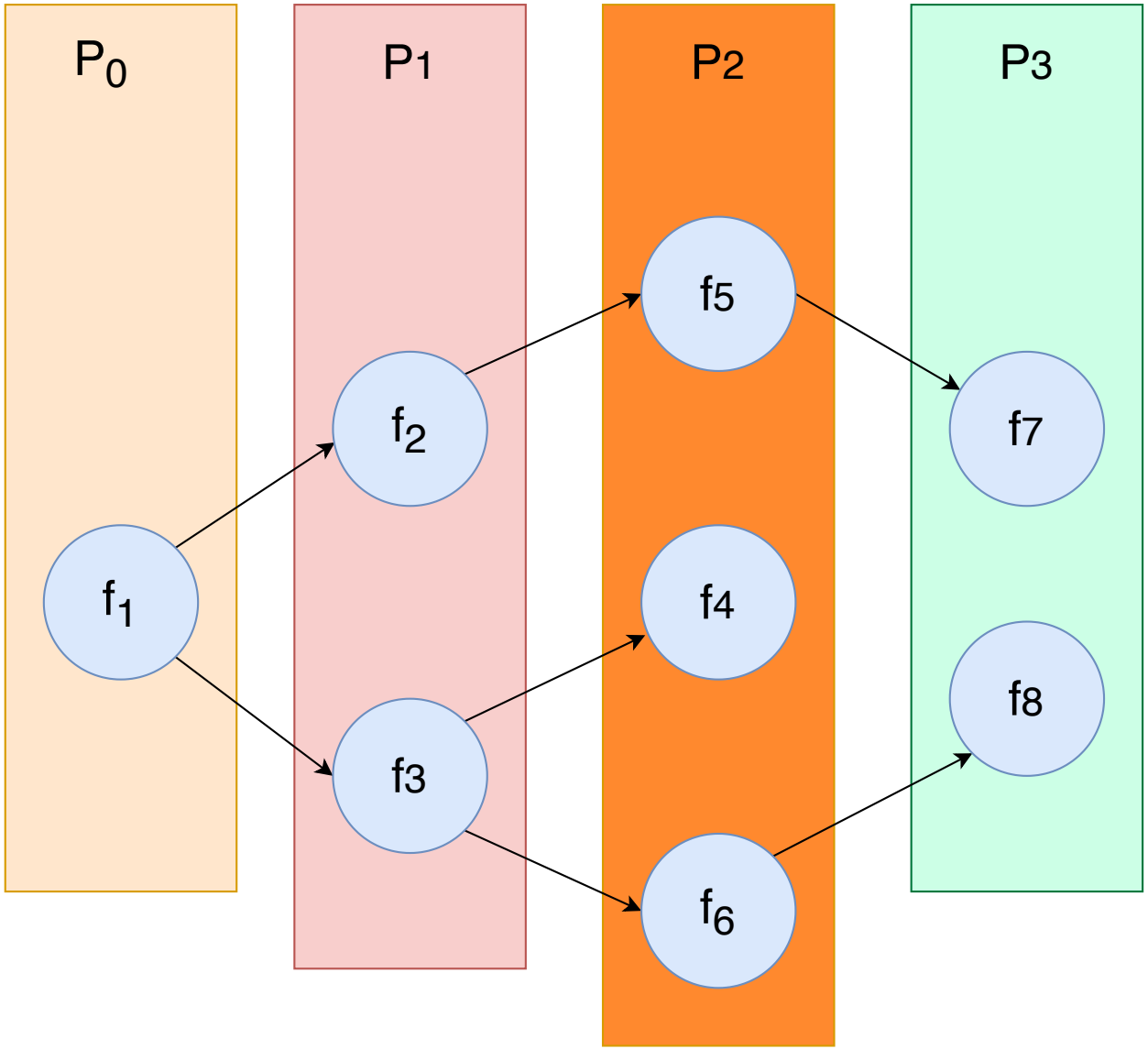
[5]Impact Analysis of Cross-Project Bugs on Software Ecosystems, ICSE 20, Wanwangying Ma et al.

Function Call Graph: Formalization

Formalization

Denote P as a package, f as a function. We have $P = \{f_1, f_2, \ldots, f_n\}$.
Denote $V = P_1 \cup P_2 \cup \cdots \cup P_m$, which forms a vertex set.
f_i has an edge to f_j **iff.** f_i calls f_j, recorded as $\langle f_i, f_j \rangle$.
Thus, an edge set can be represented as $E = \{\langle f_i, f_j \rangle \| f_i, f_j \in V\}$.
A function call graph can be defined as $G = \langle V, E \rangle$.

Function Call Graph: Formalization

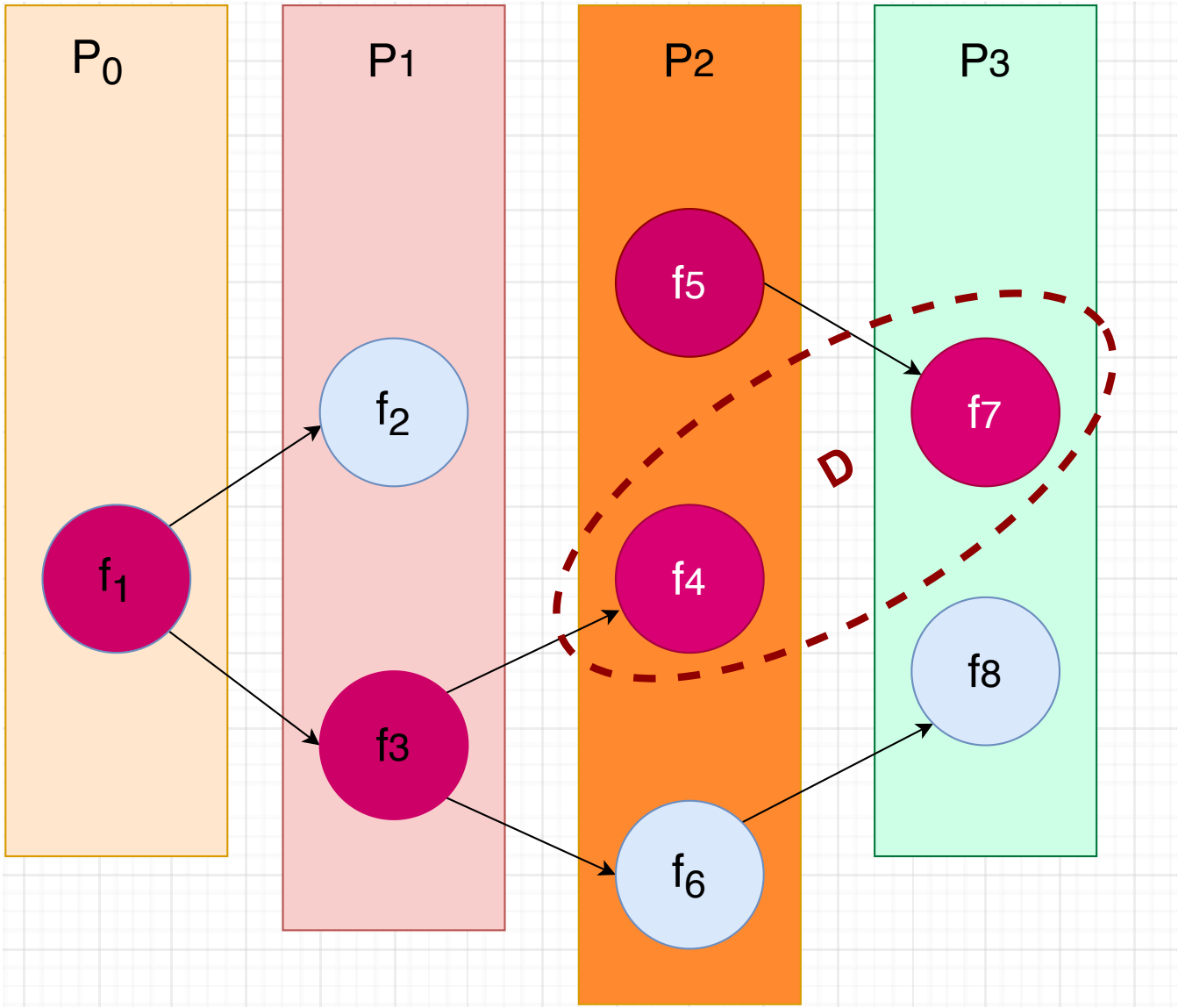


Dependency Analysis: Formulation

Problem Formulation

Suppose we have $G = \langle V, E \rangle$ and $V = P_0 \cup P_1 \cup \cdots \cup P_m$
Where P_0 denotes the entry package of the analysed project.
Apparently, $P_0, P_1, \ldots, P_m$ compose a partition of V.
Denote $D = f_k1, f_k2, \ldots, f_kn$, where f_ki refers to a function that previously included in vulnerability database.
Find if " $\exists f_i \in P_0, f_j \in D \wedge f_i$ and f_j is connected " holds.

Dependency Analysis: Formalization



Dependency Analysis: Obstacles

The main obstacles of call graph generation are Accuracy and Performance.
To remedy these issues, optimizations are made under different scenarios.
To name a few:

- bundled JS with symbolic execution: Building Call Graphs for Embedded Client-Side Code in Dynamic Web Applications, FSE 14, Hung et al.
- bundled JS with test case generation: Slimming javascript applications: An approach for removing unused functions from javascript libraries, IST 19, Vázquez et al.
- JS engine with test case generation: CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines, NDSS 19, HyungSeok et al.

Dependency Analysis: Sub-problems

RQ_1: How to Judge if $\langle f_i, f_j \rangle$ Exists?

While such edge construction is well supported by techniques like AST in languages such as C/C++ and Java (e.g., through compiler IR), it is lacking for loosely typed language(e.g., JavaScript, Python). There for accuracy is always a problem for JavaScript Call Graph generation.

Dependency Analysis: Sub-problems

RQ_2: How to Tell $f_i \in D$?

Since package managers or repository service provider(e.g. GitHub) usually only give descriptive text on vulnerability of a package release. There exists work that try to aggregate structured data from JavaDocm with Machine Learning methods[a]. Yet, the accuracy are limited(79%) and NPM documents are not semantically as good as JavaDoc. More structured descriptions on NPM **functions** or **APIs** are needed.

[a] On Using Machine Learning to Identify Knowledge in API Reference Documentation, arxiv 19, Davide et al.

Dependency Analysis: Sub-problems

RQ_3: How to Construct G within Reasonable Time?

Although there's not a dependency analysis based on NPM call graph yet, such tasks under PYPI[a] and Java Libraries[b] shows its time cost is non-ignorable.

---

[a] Impact Analysis of Cross-Project Bugs on Software Ecosystems, ICSE 20, Wanwangying Ma et al.

[b] An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects, arxiv 20, Ying Wang et al.

Conclusion & Future Work

This survey reports on prior conflicting analysis studies on NPM dependency ecosystem, highlighted and formalized the approach for a more concrete analysis.

First Phase: Call Graph Construction

Compare and test different call graph construction methods on Express.js, and manually evaluate the results.

THANKS!

Q & A