Comparing dependency issues across software package distributions

Tom Mens
Software Engineering Lab
Faculty of Sciences

@tom_mens
tom.mens@umons.ac.be
"Excellence of Science" Research Project 2018-2021

SECO-ASSIST

@secoassist secoassist.github.io

UMONS
Université de Mons

VUB
VRIJE UNIVERSITEIT BRUSSEL

UNIVERSITÉ DE NAMUR

Universiteit Antwerpen

fnrs
LA LIBERTÉ DE CHERCHER

fwo
Comparing dependency issues across software package distributions

**An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems**
A Decan, T. Mens, Ph. Grosjean (2019) Empirical Software Engineering 24(1)

**What do package dependencies tell us about semantic versioning?**
A Decan, T Mens (2019) IEEE Transactions on Software Engineering

**A formal framework for measuring technical lag in component repositories – and its application to npm**

**On the impact of security vulnerabilities in the npm package dependency network**
A Decan, T Mens, E Constantinou (2018) Int’l Conf. Mining Software Repositories

**On the evolution of technical lag in the npm package dependency network**
Dependency issues
Dependency issues

“Technical lag” due to outdated dependencies
Missed opportunities to benefit from new functionality, or fixes of known bugs and security vulnerabilities

“Dependency hell”
• Too many direct and transitive dependencies
• Broken dependencies due to backward incompatibilities
• Co-installability problems

Unmaintained packages
due to departure of maintainers

Nontransparent update policies

Incompatible or prohibited licenses
Incompatible licenses

https://tidelift.com

dependencyci

We've researched these licenses so you can enforce your licenses policies with confidence.

- Converted to SPDX format (11)
- Lifter verified (13)
- Correct (251)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A package has no known license</td>
<td>unlicensed</td>
<td>fail</td>
</tr>
<tr>
<td>A release has security vulnerabilities</td>
<td>vulnerable</td>
<td>fail</td>
</tr>
<tr>
<td>A release has known critical bugs</td>
<td>broken</td>
<td>fail</td>
</tr>
<tr>
<td>A package uses a disallowed license</td>
<td>license_prohibited</td>
<td>fail</td>
</tr>
<tr>
<td>A package is using an inactive release stream</td>
<td>inactive_stream</td>
<td>warn</td>
</tr>
</tbody>
</table>
Fragility due to transitive dependencies

Unexpected removal of left-pad caused > 2% of all packages to become uninstallable (> 5,400 packages)

Release 0.5.0 of i18n broke dependent package ActiveRecord that was transitively required by >5% of all packages
Libraries.io monitors **6,901,989** open source packages across **37** different package managers

https://libraries.io (7 January 2020)
Characterising the evolution of package dependency networks

830K packages – 5.8M package versions – 20.5M dependencies *(April 2017)*

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cargo</td>
<td>2014</td>
<td>Rust</td>
<td>9k</td>
<td>48k</td>
<td>150k</td>
</tr>
<tr>
<td>CPAN</td>
<td>1995</td>
<td>Perl</td>
<td>34k</td>
<td>259k</td>
<td>1,078k</td>
</tr>
<tr>
<td>CRAN</td>
<td>1997</td>
<td>R</td>
<td>12k</td>
<td>67k</td>
<td>164k</td>
</tr>
<tr>
<td>npm</td>
<td>2010</td>
<td>JavaScript</td>
<td>462k</td>
<td>3,038k</td>
<td>13,611k</td>
</tr>
<tr>
<td>NuGet</td>
<td>2010</td>
<td>.NET</td>
<td>84k</td>
<td>936k</td>
<td>1,665k</td>
</tr>
<tr>
<td>Packagist</td>
<td>2012</td>
<td>PHP</td>
<td>97k</td>
<td>669k</td>
<td>1,863k</td>
</tr>
<tr>
<td>RubyGems</td>
<td>2004</td>
<td>Ruby</td>
<td>132k</td>
<td>795k</td>
<td>1,894k</td>
</tr>
</tbody>
</table>

Continuing Growth

Package dependency networks grow exponentially in terms of number of packages and/or dependencies.

Fastest growth for npm
Slowest growth for CRAN
Continuing Change

- Number of package updates grows over time
- >50% of package releases are *updated within 2 months*
- *Required* and *young* packages are updated more frequently

![Graph showing package update trends over time with logarithmic scale on the y-axis ranging from 10^0 to 10^6. The x-axis represents the years from 2012 to 2017. The graph compares the number of updates for various package managers, including cargo, npm, cpan, nuget, packagist, and rubygems. The fastest growth is indicated for npm, while the slowest growth is for CRAN.](image-url)
Increasing level of reuse

- Highly connected network, containing 60% to 80% of all packages
- Power law behavior: A stable minority (20%) of required packages collect over 80% of all reverse dependencies

**Reusability index**: Maximal value $n$ such that there exist $n$ required packages having at least $n$ dependent packages.

- Fastest growth for npm
High number of deep transitive dependencies

- Fragile packages may have a very high transitive impact
- Over 50% of top-level packages have a deep dependency graph

![Graph showing number of packages and transitive dependency depth](image-url)
Outdated Dependencies

Should package maintainers upgrade their dependencies to more recent versions?

😀 Upgrades benefit from bug and security fixes
😀 Upgrading allows to use new features
😢 Upgrading requires effort
😢 Upgrading may introduce breaking changes
Outdated Dependencies

Outdatedness is related to the type of dependency constraint being used.

Strict (i.e. pinned) constraints represent about 33% of all outdated dependencies.

*Outdated* runtime dependencies in npm
**Technical Lag**

*Technical lag* measures how outdated a package or dependency is w.r.t. the “ideal” situation where “ideal” = “most recent”; “most secure”; “least bugs”; “most compatible”; ...

---

Technical lag measures how outdated a package or dependency is w.r.t. the “ideal” situation where “ideal” = “most recent”; “most secure”; “least bugs”; “most compatible”; ...

- **load-script**: 1.0.0 - 08-03-2015
- **debug**: 2.7.0 - 22-09-2017
  - Missed versions:
    - 3.0.0 - 08-08-2017
    - 3.0.1 - 24-08-2017
    - 3.1.0 - 26-09-2017
- **ms**: 2.0.1 - 16-05-2017
  - Missed versions:
    - 2.1.0 - 30-11-2017
    - 2.1.1 - 30-11-2017
- **youtube-player**: 5.5.0 - 20-02-2018
  - Missed versions:
    - ^2.6.6
    - ^3.0.0
Need for dependency monitoring tools

Example: David Dependency Manager for npm projects

<table>
<thead>
<tr>
<th>DEPENDENCY</th>
<th>REQUIRED</th>
<th>STABLE</th>
<th>LATEST</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>async-foreach</td>
<td>^0.1.3</td>
<td>0.1.3</td>
<td>0.1.3</td>
<td>green</td>
</tr>
<tr>
<td>chalk</td>
<td>^1.1.1</td>
<td>3.0.0</td>
<td>3.0.0</td>
<td>red</td>
</tr>
<tr>
<td>cross-spawn</td>
<td>^3.0.0</td>
<td>7.0.1</td>
<td>7.0.1</td>
<td>red</td>
</tr>
<tr>
<td>gaze</td>
<td>^1.0.0</td>
<td>1.1.3</td>
<td>1.1.3</td>
<td>green</td>
</tr>
<tr>
<td>get-stdin</td>
<td>^4.0.1</td>
<td>7.0.0</td>
<td>7.0.0</td>
<td>red</td>
</tr>
<tr>
<td>glob</td>
<td>^7.0.3</td>
<td>7.1.6</td>
<td>7.1.6</td>
<td>green</td>
</tr>
</tbody>
</table>

My npm Project: 4.13.1
https://david-dm.org
Avoiding breaking changes through Semantic Versioning

Is semantic versioning respected by software package distributions?

- Breaking changes
- Backwards compatible changes
- Bug fixes

Major: 3
Minor: 9
Patch: 2

Most permissive: ~
Most restrictive: strict
Semantic versioning

Different package managers interpret version constraints in different ways:

<table>
<thead>
<tr>
<th>Constr.</th>
<th>Cargo</th>
<th>npm</th>
<th>Packagist</th>
<th>Rubygems</th>
</tr>
</thead>
<tbody>
<tr>
<td>=1.0.0</td>
<td>[1.0.0]</td>
<td>[1.0.0]</td>
<td>[1.0.0]</td>
<td>[1.0.0]</td>
</tr>
<tr>
<td>1.0.0</td>
<td>[1.0.0, 2.0.0[</td>
<td>[1.0.0]</td>
<td>[1.0.0]</td>
<td>[1.0.0]</td>
</tr>
<tr>
<td>1.0</td>
<td>[1.0.0, 2.0.0[</td>
<td>[1.0.0, 1.1.0[</td>
<td>[1.0.0]</td>
<td>[1.0.0]</td>
</tr>
<tr>
<td>1</td>
<td>[1.0.0, 2.0.0[</td>
<td>[1.0.0, 2.0.0[</td>
<td>[1.0.0]</td>
<td>[1.0.0]</td>
</tr>
<tr>
<td>~1.2.3</td>
<td>[1.2.3, 1.3.0]</td>
<td>[1.2.3, 1.3.0]</td>
<td>[1.2.3, 1.3.0]</td>
<td>[1.2.3, 1.3.0]</td>
</tr>
<tr>
<td>~1.2</td>
<td>[1.2.0, 1.3.0]</td>
<td>[1.2.0, 1.3.0]</td>
<td>[1.2.0, 2.0.0]</td>
<td>[1.2.0, 2.0.0]</td>
</tr>
<tr>
<td>~1</td>
<td>[1.0.0, 2.0.0]</td>
<td>[1.0.0, 2.0.0]</td>
<td>[1.0.0, 2.0.0]</td>
<td>N/A</td>
</tr>
<tr>
<td>^1.2.3</td>
<td>[1.2.3, 2.0.0]</td>
<td>[1.2.3, 2.0.0]</td>
<td>[1.2.3, 2.0.0]</td>
<td>N/A</td>
</tr>
<tr>
<td>&gt;1.2.3</td>
<td>[1.2.3, +∞[</td>
<td>[1.2.3, +∞[</td>
<td>[1.2.3, +∞[</td>
<td>[1.2.3, +∞[</td>
</tr>
<tr>
<td>~0.1.2</td>
<td>[0.1.2, 0.2.0[</td>
<td>[0.1.2, 0.2.0[</td>
<td>[0.1.2, 0.2.0[</td>
<td>N/A</td>
</tr>
<tr>
<td>^0.1.2</td>
<td>[0.1.2, 0.2.0]</td>
<td>[0.1.2, 0.2.0]</td>
<td>[0.1.2, 0.2.0]</td>
<td>N/A</td>
</tr>
</tbody>
</table>

More restrictive than semver

More permissive than semver
Semantic versioning

- Cargo, npm and Packagist are mostly semver-compliant. All three are more permissive than semver for 0.y.z versions.
- All considered ecosystems become more compliant over time.
- >16% of restrictive constraints in npm, Packagist and Rubygems prevents adoption of backward compatible upgrades.
You are likely vulnerable:

- If you do not know the versions of all components you use … This includes components you directly use as well as nested dependencies.
- If software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.
- If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use.
- If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, which leaves organizations open to many days or months of unnecessary exposure to fixed vulnerabilities.
- If software developers do not test the compatibility of updated, upgraded, or patched libraries.
Security vulnerabilities in npm

<table>
<thead>
<tr>
<th>Vulnerable packages</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># vulnerabilities</td>
<td>399</td>
</tr>
<tr>
<td># vulnerable packages</td>
<td>269</td>
</tr>
<tr>
<td># releases of vulnerable packages</td>
<td>14,931</td>
</tr>
<tr>
<td># vulnerable releases</td>
<td>6,752 (45%)</td>
</tr>
<tr>
<td># dependent packages</td>
<td>133,602</td>
</tr>
<tr>
<td># dependent packages affected by the vulnerable packages</td>
<td>72,470 (54%)</td>
</tr>
</tbody>
</table>

On the impact of security vulnerabilities in the npm package dependency network.
A Decan, T Mens, E Constantinou (2018) Int’l Conf. Mining Software Repositories
Security vulnerabilities in npm
When are vulnerabilities discovered?

>40% of all vulnerabilities are not *discovered* even 2.5 years after their introduction, regardless of their severity.
Security vulnerabilities in npm
When are vulnerabilities fixed?

~20% of vulnerabilities take more than 1 year to be fixed.
Security vulnerabilities in npm

When are vulnerabilities fixed in dependent packages?

<table>
<thead>
<tr>
<th># vulnerable packages</th>
<th># releases of vulnerable packages</th>
<th># vulnerable releases</th>
<th># dependent packages</th>
<th># dependent packages affected by the vulnerable packages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>269</td>
<td>14,931</td>
<td>6,752</td>
<td>133,602</td>
</tr>
</tbody>
</table>

>33% of all affected dependents are not (yet) fixed!
Security vulnerabilities in npm

Why do vulnerabilities remain unfixed in dependent packages?

- Improper or too restrictive use of dependency constraints
- Package is no longer actively maintained
- Maintainers are unaware of the vulnerability or how to fix it
- Fixed version of the dependency contains incompatible changes
Tool support: Monitor and update vulnerable dependencies

GitHub
Automated security alerts and updates

Snyk
Continuously find and fix known vulnerabilities in a package’s dependencies
https://snyk.io

Retire.js
Scans for the use of JavaScript libraries with known vulnerabilities
http://retirejs.github.io/retire.js/

OWASP Dependency-Check
Detects publicly disclosed vulnerabilities contained within a project’s dependencies.
https://github.com/jeremylong/DependencyCheck

Eclipse Steady
Detects known vulnerabilities in dependencies to open source Java and Python components through combination of static and dynamic analysis techniques
https://eclipse.github.io/steady/
Conclusion

• Package dependency networks are affected by multiple dependency issues
  • Many and deep transitive dependencies
  • Outdated dependencies
  • Breaking changes
  • Vulnerable dependencies

• Automated tools and policies can help mitigating these issues
  • Measuring, monitoring and updating outdated and vulnerable dependencies
  • Supporting semantic versioning
  • Supporting transitive dependencies
  • Detecting vulnerabilities that matter (avoid false positives/negatives)