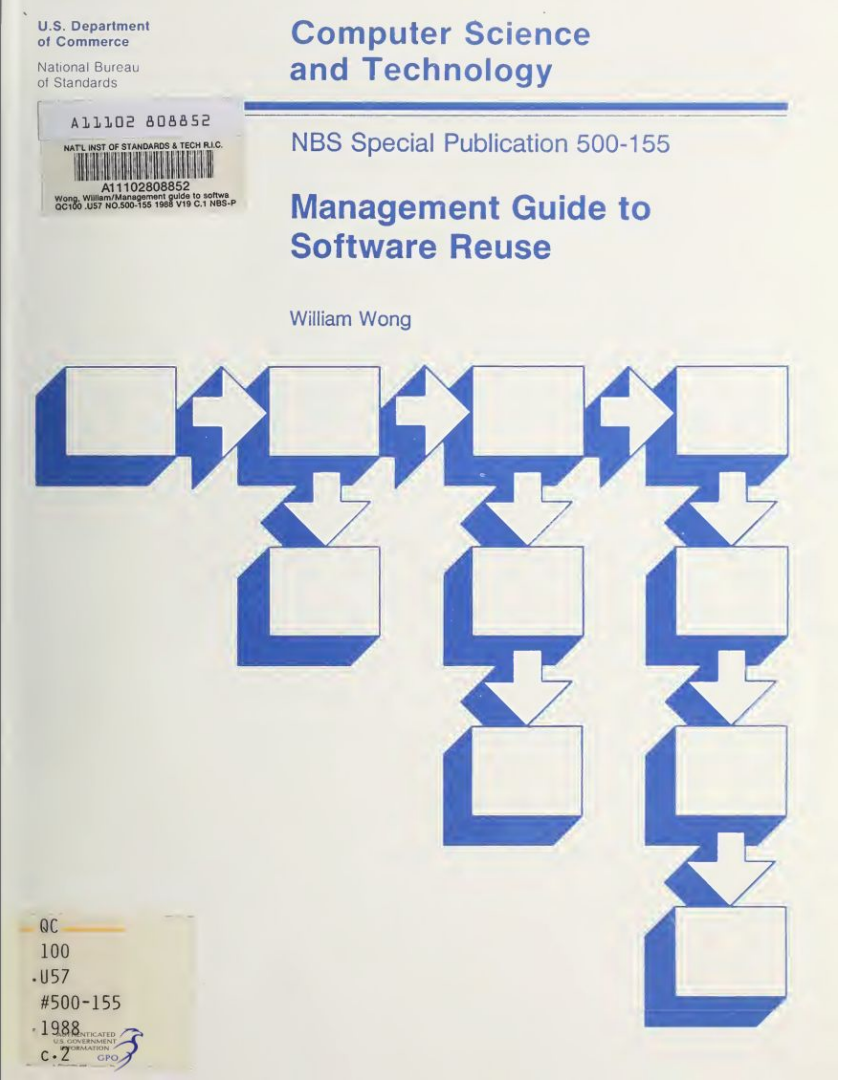
ENDOR LABS

Going Beyond Metadata

Why We need To Think of Adapting Static Analysis in Dependency Tools

Joseph Hejderup



Back to 1980

Let's first understand the ideas behind Software Reuse

Wong, William. *A management overview of software reuse*. US Department of Commerce, National Bureau of Standards, 1986.

// Software Reuse: Ideas

5.1 Productivity

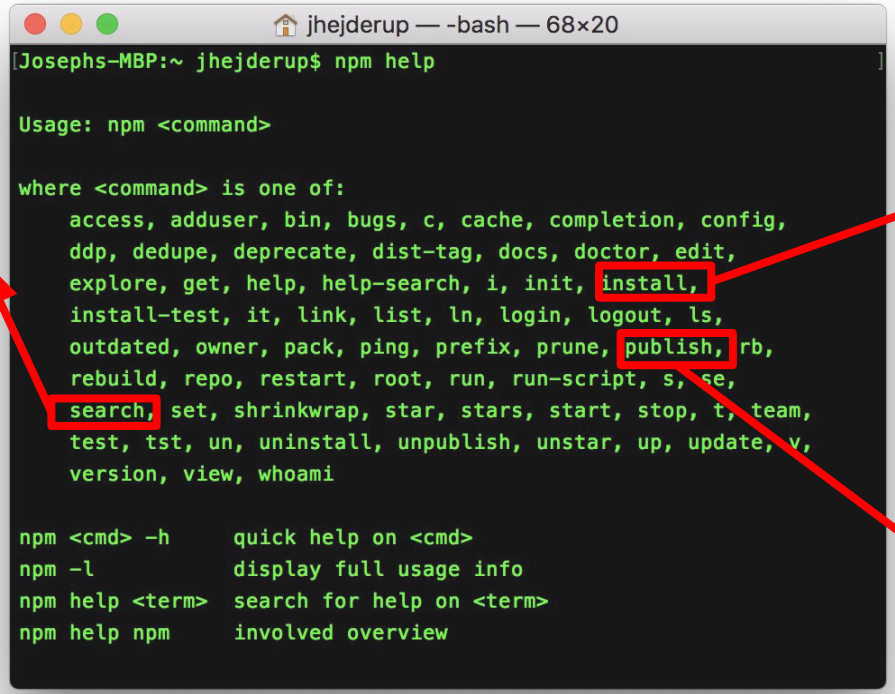
Reusing well-designed, well-developed, and well-documented software improves productivity and reduces software development time, costs, and risks.

5.2 Quality

Improvements in the quality of software developed from well-designed, well-tested, and well-documented reusable software components

Wong, William. *A management overview of software reuse*. US Department of Commerce, National Bureau of Standards, 1986.

// Software Reuse: Implementation



Gateway to thousands of libraries and frameworks

Dependency management with automated conflict resolution

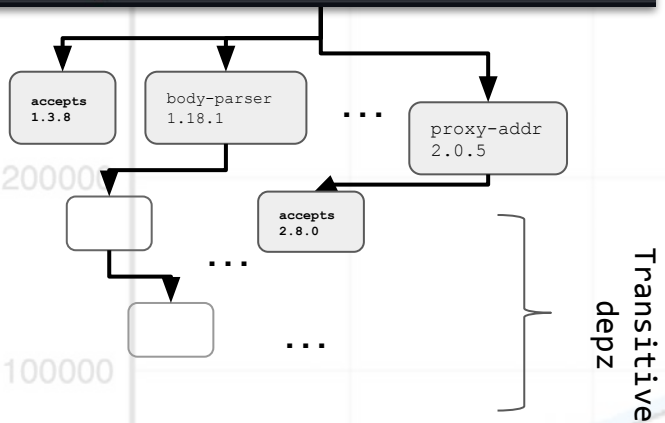
Centralized distribution

Package Managers

// Dependency Management

Problem #1: Local Level

```
Compiling async-executor v1.4.1
Compiling rand_chacha v0.2.2
 Building [==============>          ] 194/307
```



Complex & Large Package Compositions

```
[package]
name = "webframework"
version = "4.16.3"

[dependencies]
accepts = "~1.3.5"
body-parser = "1.18.2"
depd = "~1.1.2"
encodeurl = "1.0.2"
escape-html = "~1.0.3"
etag = "1.8.1"
proxy-addr = "~2.0.3"
```

	TODAY	...3 DAYS LATER
accepts	1.3.7	1.3.12
depd	1.1.3	1.1.3
escape-html	1.0.6	1.0.9
proxy-addr	2.0.5	2.0.5

Temporal Properties

http://www.modulecounts.com

// Dependency Management

Problem #2: Global/Repository Level

QZ Quartz

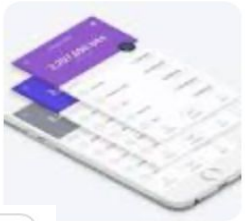
How one programmer broke the internet by deleting a tiny piece of code

Lots of npm packages
how this tiny bit of co
Mar 27, 2016

ZDNet

Hacker backdoors popular JavaScript library to steal Bitcoin funds

The library loading the malicious code is named Event-Stream, a JavaScript



WIRED

A Second SolarWinds Hack Deepens Third-Party Software Fears

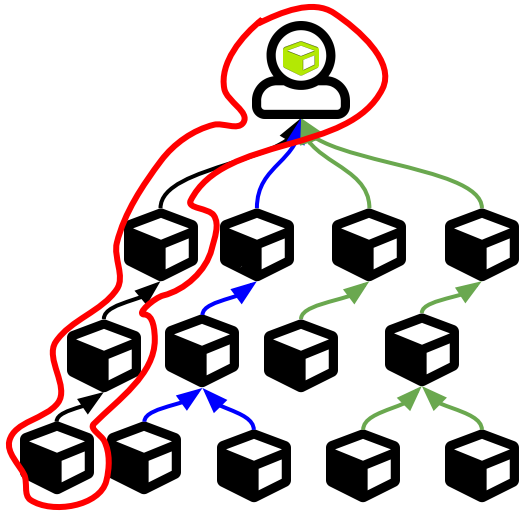
It's been more than two months since revelations that alleged Russia-backed hackers broke into the IT management firm SolarWinds and used ...

5 days ago

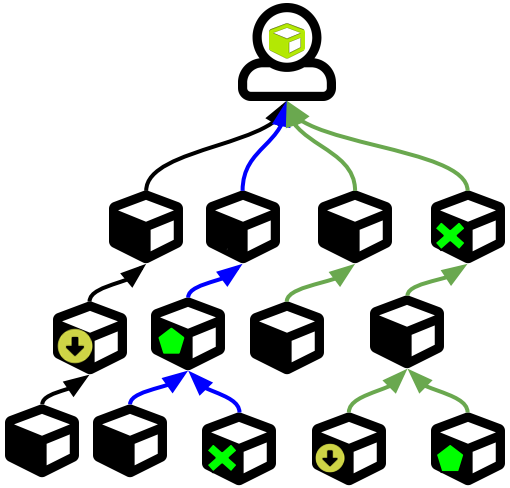
// Dependency Management

Temporal Properties => Version Pinning/Lock files
Everything Else => Dependency Analyzers/Bots/Plugins

* vulnerabilities
* updates
* audit
* quality
* deprecation
* ...more to come!



* alert fatigue
* actionability?
* precision?



// Revisit

5.1 Productivity

Reusing well-designed, well-developed, and well-documented software improves productivity and reduces software development time, costs, and risks.

Wong, William. *A management overview of software reuse*. US Department of Commerce, National Bureau of Standards, 1986.

// Classic Alert Fatigue? Nope

We need first address the quality of analyzers!

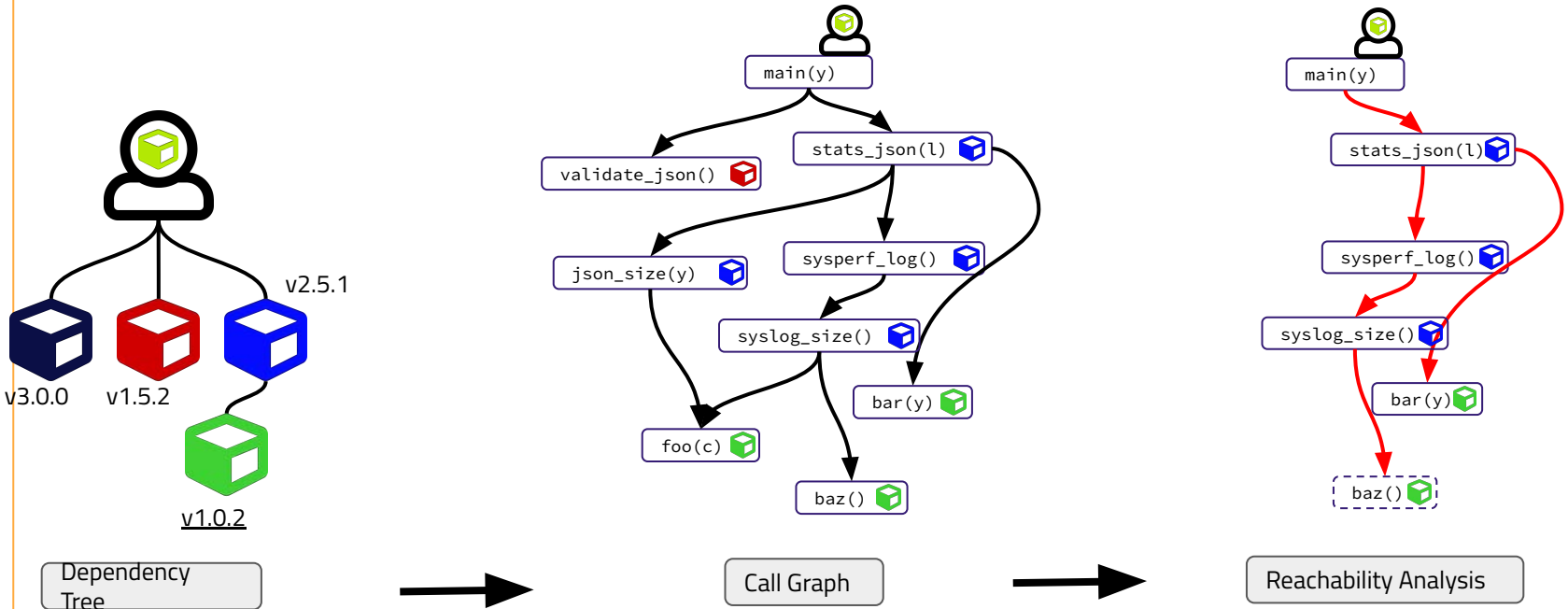
```
[package]
name = “webframework”
version = "4.16.3"


[dependencies]
accepts = "~1.3.5"
body-parser = "1.18.2"
depd = "~1.1.2"
encodeurl = "1.0.2"
escape-html = "~1.0.3"
etag = "1.8.1"
proxy-addr = "~2.0.3"
```

Rule #1: Metadata is not source code!

Declaration ≠ usage

// Classic Alert Fatigue? Nope

We need first address the quality of analyzers!



Rule #2: Make Code first-class citizens

// Program Analysis

great, got it but expensive and not scalable, dependencies are many, right?

ktrianta/rust-callgraphs

35,896 Packages

208,023 Releases

Repository index

23,767 Packages

142,301 Releases

call graphs

10

days

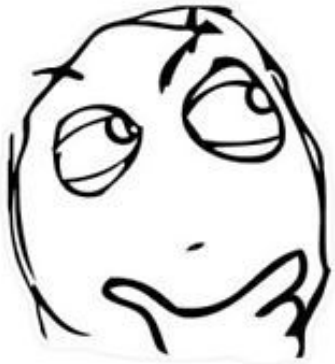
67%

(actually 80%)

Rule #3: Aim for Light-Weight Analysis Techniques!

// Program Analysis

* “Overkill for me to add Program Analysis”
* “What about Python/JavaScript?”
* “Program Analysis suffers from False Negatives, my security customers won’t be happy about it”

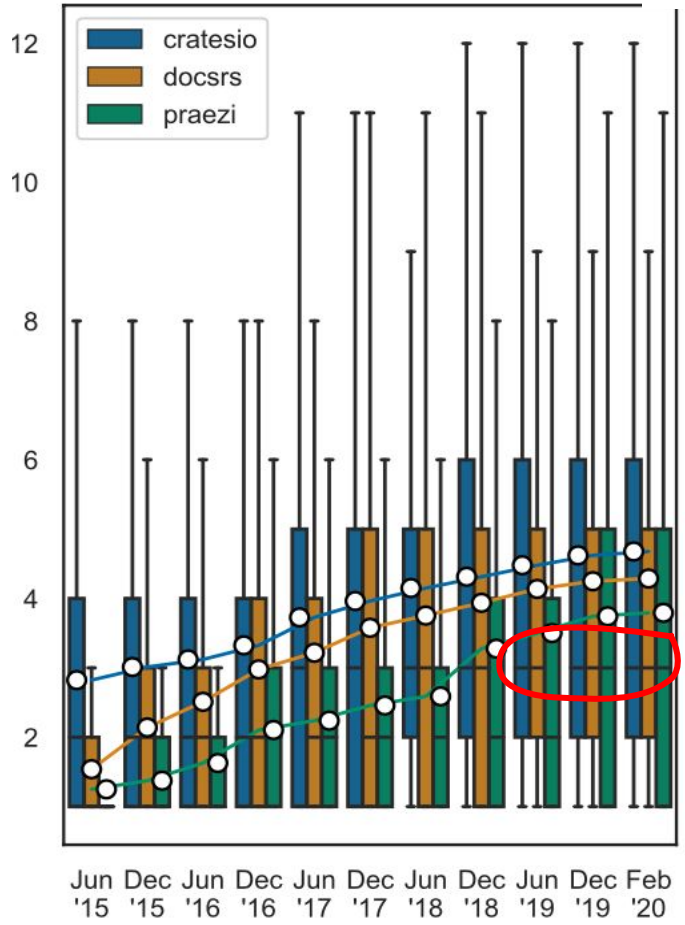


// Program Analysis

Let’s settle the question through some research!

RQ: What is the difference in the number of reported dependencies between traditional metadata-based approaches vs program analysis approaches?

// Number of Direct Dependencies

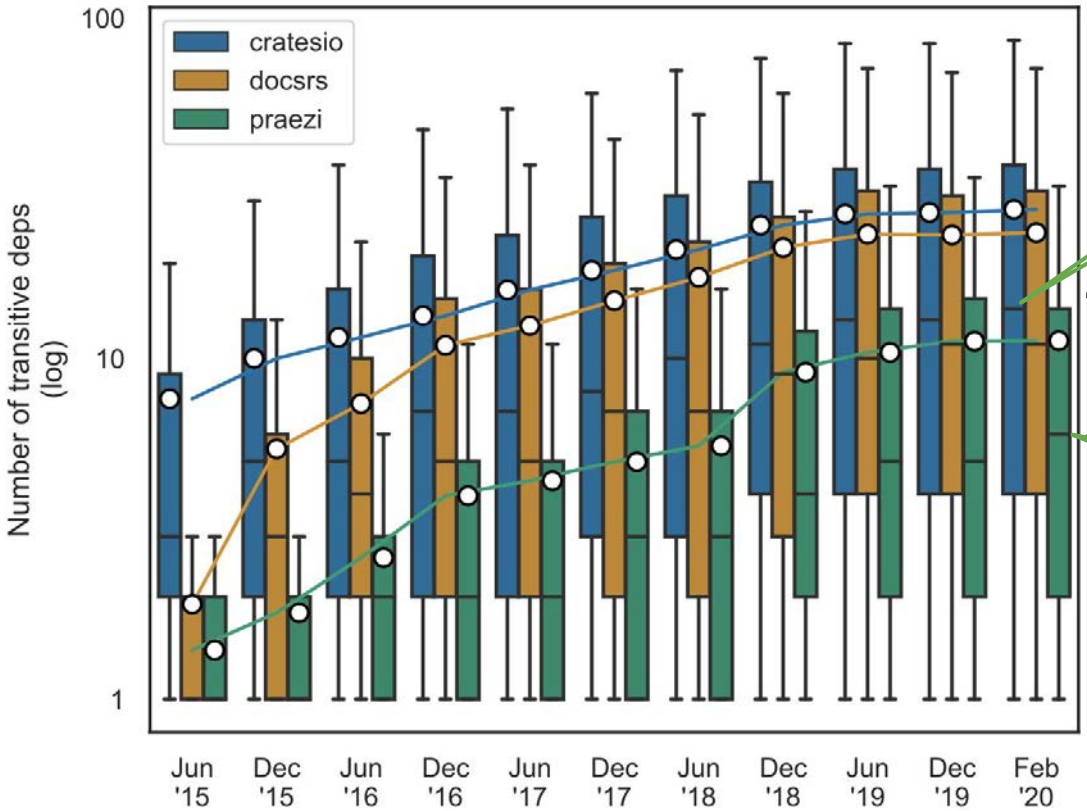


The median is similar; the mean relatively when comparing the three representations.

A metadata-based direct deps slightly over-approx. a static analysis inferred direct deps!

Hejderup, J., Beller, M., Triantafyllou, K., & Gousios, G. (2022). Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering*, *27*(5), 1-42.

// Number of Transitive Dependencies



17

Gap

6

No close approximation of each other!

Mean: packages are indirectly not calling **60% of its resolved transitive dependencies**

Hejderup, J., Beller, M., Triantafyllou, K., & Gousios, G. (2022). Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering*, *27*(5), 1-42.

// Why is there such a large difference for transitive depz?

diff metadata-based call-based dependency tree
in 34 randomly selected cases

Hejderup, J., Beller, M., Triantafyllou, K., & Gousios, G. (2022). Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering*, *27*(5), 1-42.

// Why is there such a large difference for transitive depz?

Diff #1

* 3 x No import statements
* 4 x Import statements but no usage

Not totally unexpected but also not that common!

Hejderup, J., Beller, M., Triantafyllou, K., & Gousios, G. (2022). Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering*, *27*(5), 1-42.

// Why is there such a large difference for transitive depz?

Diff #2

* 1 x conditional compilation (e.g., [#cfg(...)])
* 2 x derive macro libraries (code generation)
* 1 x test dependency (in wrong section!)

Not all dependencies are runtime libraries!
Optional dependencies surface and enabled in code; how do we handle them?

Hejderup, J., Beller, M., Triantafyllou, K., & Gousios, G. (2022). Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering*, 27(5), 1-42.

// Why is there such a large difference for transitive depz?

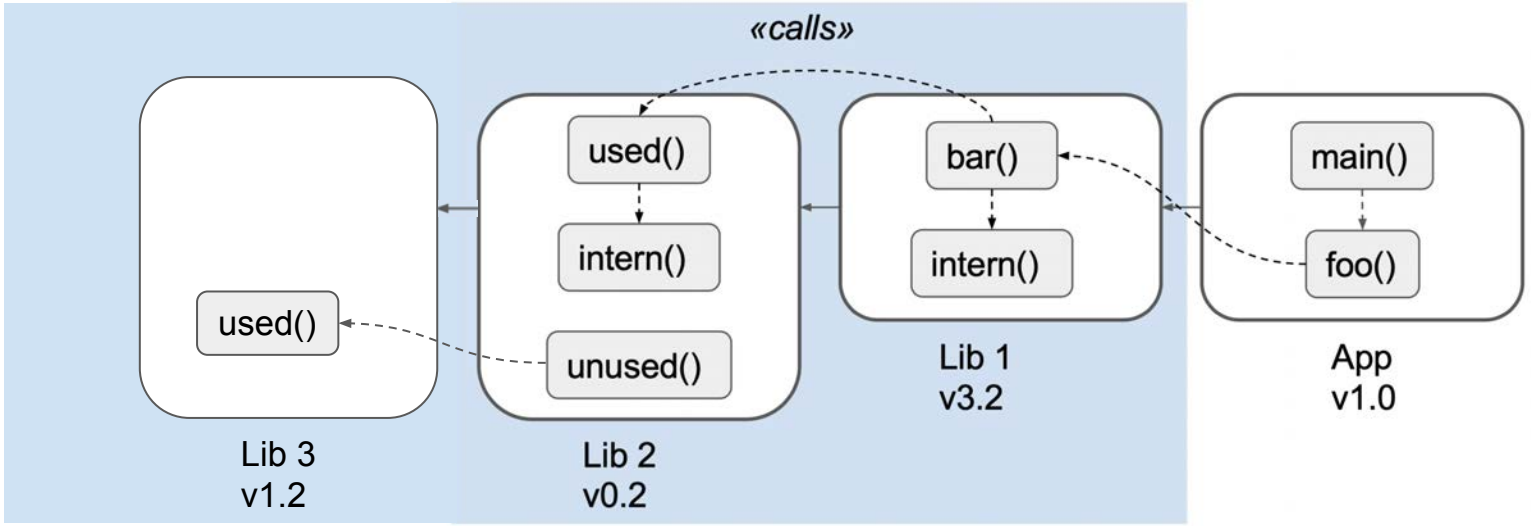
Diff #3

* 16 x non-reachable transitive dependencies!

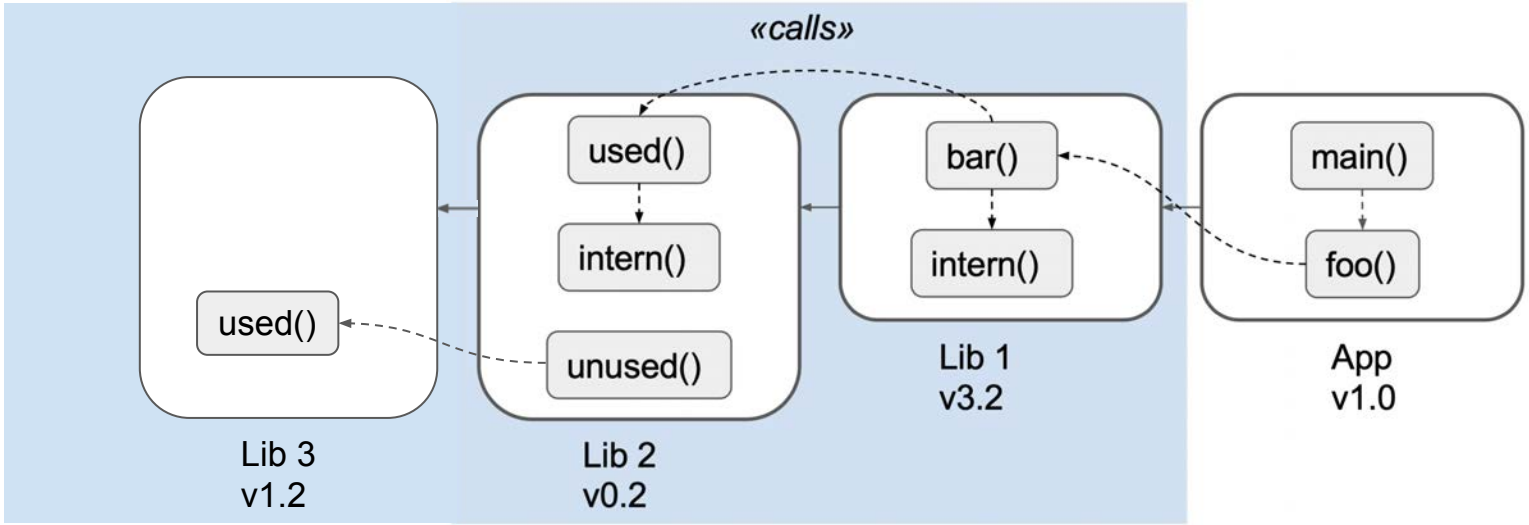
What do you mean?

Hejderup, J., Beller, M., Triantafyllou, K., & Gousios, G. (2022). Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering*, *27*(5), 1-42.

// How many dependencies is App v1.0 using?



// Context Matters!



We make the general assumption that we use

- **ALL APIs** of all direct dependencies,
- and **then ALL APIs** of transitive dependencies

// Practical Implications - Trade-off

Analysis Scope - Direct Dependencies

* Declared dependency closely estimates a utilized dependency.
* Metadata > Static Analysis

Pros:
- No need to implement program analysis
- Higher recall for security/soundness-sensitive applications

Cons:
- Insensitive to simple things like no import statement & specifics of APIs being utilized
- Cannot eliminate dependencies that are solely doing code-generation (i.e, no actual runtime dependency)

// Practical Implications - Trade-off

Analysis Scope - Transitive Dependencies

* Static Analysis > Metadata

Pros:
- Capture the actual usage, reducing false positives.
- Higher actionability

Cons:
- False Negatives => May not be ideal for security applications.
- Highly dynamic libraries are challenging to analyze

// Program Analysis: Precision & Recall

* Recall
    - Coverage of language features
    - Lost Precision, still better than metadata
* Precision
    - Algorithms like Points-to algorithm not that scalable

* General Implications
    - Scope of analysis: Project + its dependency tree
    - Package repositories are not homogeneous collection of libraries

Joseph Hejderup

joseph@endor.ai

ENDOR LABS