Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Präzi: From Package-based to Call-based Dependency Networks

Joseph Hejderup · Moritz Beller* · Konstantinos Triantafyllou · Georgios Gousios*

Received: date / Accepted: date

Abstract Modern programming languages such as Java, JavaScript, and Rust encourage software reuse by hosting diverse and fast-growing repositories of highly interdependent packages (i.e., reusable libraries) for their users. The standard way to study the interdependence between software packages is to infer a package dependency network by parsing manifest data. Such networks help answer questions such as "How many packages have dependencies to packages with known security issues?" or "What are the most used packages?". However, an overlooked aspect in existing studies is that manifest-inferred relationships do not necessarily examine the actual usage of these dependencies in source code. To better model dependencies between packages, we developed Präzi, an approach combining manifests and call graphs of packages. Präzi constructs a dependency network at the more fine-grained function-level, instead of at the manifest level. This paper discusses a prototypical Präzi implementation for the popular system programming language Rust. We use Präzi to characterize Rust's package repository, Crates.io, at the function level and perform a comparative study with metadata-based networks. Our results show that metadata-based networks generalize how packages use their dependencies.

*Work largely conducted while the author was a researcher at TU Delft, The Netherlands.

✉ Joseph Hejderup
Delft University of Technology
E-mail: j.i.hejderup@tudelft.nl

Moritz Beller
Facebook, Inc.
E-mail: mmb@fb.com

Konstantinos Triantafyllou
University of Athens
E-mail: ks.triantafyllou@gmail.com

Georgios Gousios
Facebook, Inc.
E-mail: gousiosg@fb.com

Using PRÄZI, we find packages call only 40% of their resolved dependencies, and that manual analysis of 34 cases reveals that not all packages use a dependency the same way. We argue that researchers and practitioners interested in understanding how developers or programs use dependencies should account for its context—not the sum of all resolved dependencies.

Keywords package repository · dependency network · package manager · software ecosystem · network analysis · call graphs

1 Introduction

Converting information between different well-known formats, accessing external storage, manipulating information such as numbers, locations, and dates, or integrating with popular online services are examples of essential operations that developers need to handle in software projects. Unlike the standard library of programming languages, these essential operations change over time as a result of evolving technologies (e.g., shift from XML to JSON) or provide support to niche user communities (e.g., interfaces to Twitter API or Amazon AWS SDK). In addition to a standard library, modern programming languages such as Java, JavaScript, C#, and Rust also host public repositories for developers to contribute essential operations in the form of reusable libraries (also known as packages). A package manager such as MAVEN, NPM, NUGET, and CARGO enables developers to discover and import packages from repositories in their workspace.

To be modular, a package should perform a well-defined task, developed with simple interfaces, and be pluggable (composable) with other packages (Schlueter 2013; Abdalkareem et al. 2019). The manifest file such as Rust's CARGO.TOML and NPM's `package.json` in every package makes libraries composable: developers declare in the manifest how others can import their library and also if it utilizes external libraries by specifying dependencies on other existing packages. As packages can depend on one another through manifests, package repositories implicitly form a complex network, known as a Package Dependency Network (PDN) (Decan et al. 2018a; Hejderup 2015; Kikas et al. 2017).

In light of repository-wide incidents such as the `left-pad` package removal (Schlueter 2017), the hiding of a bitcoin wallet stealer in the legitimate `event-stream` package (Baldwin 2018), and malicious typosquatting packages in PyPI (Dunn 2017), researchers are conducting network analysis of package repositories for risk assessment (Zimmermann et al. 2019; Decan et al. 2018a; Kikas et al. 2017), sustainability evaluation (Valiev et al. 2018; Decan et al. 2019), license violations (Duan et al. 2017), and for detecting breaking changes (Mezzetti et al. 2018; Chen et al. 2020; Mujahid et al. 2020). Constructing a PDN for such analyses typically involves mining available manifests in the repository and then resolving dependency constraints in each manifest using a specific resolver (i.e., variations of `semver`) to infer relationships between packages (Kikas et al. 2017; Hejderup 2015).

Inferring networks solely from package manifests yields an incomplete representation of package repositories. Manifests only describe metadata of package dependencies and thus miss information on actual source code reuse, making network analysis prone to false positives. For example, a project might have redundant dependencies to packages whose functionality is not used anymore. Without knowing how packages actually use external libraries, Ponta et al. (2018) and Zapata et al. (2018)'s work on vulnerability checking packages demonstrates that metadata-based analysis have limited actionability, making it difficult for developers to understand how vulnerabilities in external libraries affect their code. Increasingly, package repository workgroups such as the Rust Ecosystem WG[1] are also calling for more comprehensive network analysis of package repositories to support code-centric analysis for more effective identification of critical yet unstable packages (Zhang et al. 2020a; Bogart et al. 2016). One such example is the `Libz Blitz` (Brian et al. 2020) initiative where community members come together and contribute to poorly maintained yet critical packages in Crates.io as an effort to stabilize highly reused code in the repository.

This work proposes code-centric dependency network analysis by inferring dependency relationships at the function call level. Call graphs capture how functions between packages use each other and thus naturally lend themselves to this objective. We coin networks generated from call graphs Call-based Dependency Networks (CDNs). To generate a CDN from a package repository, we devise Präzi, an approach that generates call graphs of packages and then merges them into a single network with functions embedding package qualifiers. The result is a more fine-grained dependency network that improves over current PDN analyses by examining the actual package dependencies in use.

We implement Präzi for Crates.io to demonstrate the feasibility of our approach. Unlike repositories hosting analyzable binaries such as Maven Central, Crates.io requires large-scale compilation of the repository to produce binaries for call graph generation. The resulting CDN comprises 90% of all compilable packages, achieving a near-complete representation of Crates.io. Then, inspired by Kikas et al. (2017)'s PDN study, we characterize and derive new insights on the evolution of Crates.io. We also compare CDNs against dependency networks derived from conventional metadata to understand their differences and similarities for dependency network analyses. Lastly, we manually investigate 34 direct and transitive package relationships to understand how reliably a CDN represents actual use of dependencies in the source code.

Our results find that one in two function calls in Crates.io are a call from a package to a dependency, suggesting high code reuse. On average, we find that a package calls at least one function in 78.8% of its direct dependencies and at least one function in 40% of its transitive dependencies, suggesting that more than half of all transitive dependencies of packages are potentially not called. When looking at APIs, packages have three times more indirect

[1] `https://github.com/rust-lang-nursery/ecosystem-wg`

(i.e., transitive) callers than direct callers. On average, a package has two new function calls every six months. Moreover, the number of calls from a package to its dependencies increases by 6.6 new direct calls and 12.2 indirect calls every six months. Reachability analysis reveals that a majority of packages in Crates.io have no or limited reachability. Only a handful packages (i.e., 0.37% of packages in 2020) are reachable by more than 10% of Crates.io. Among the most central packages, the most reachable function can reach up to 30% of all packages in Crates.io. The high indirect use of APIs in transitive dependencies of packages could constitute an important but missing confounding variable in API studies and manifest as an important threat to security and stability in practice.

The metadata-based networks and call-based networks report similar results for analysis involving direct package relationships. However, notable differences exist between the studied networks for transitive dependencies and for analyzing the most dependent packages. Metadata networks report twice the number of transitive dependencies than the CDN. Our findings in the manual analysis indicate that the high variance is a result of transitive dependencies not being indirectly reachable (utilized) from the package. A package uses a subset of its direct dependencies—not all available functionality. Thus, analysis of transitive dependencies is not generalizable but contextual. Two packages that depend on the same library and have two different use cases are likely to use their transitive dependencies differently. Thus, dependency checkers, such as GitHub's `Dependabot`[2] and Rust's `cargo-audit`[3], should consider augmenting their recommendations with call graph information to help developers make more informed decisions and reduce false positives. As a step towards inferring networks from the source code of package repositories, Präzi can enable both researchers and practitioners to estimate complex patterns of relationships between packages and their functions.

In summary, this work makes the following contributions:

- An approach to create call-based package dependency networks (CDNs) called Präzi.
- An open-source implementation for generating CDN of Rust's Crates.io
- An empirical study describing the structure, evolution, and fragility of Crates.io from a package and function view.
- A comparison of network analyses using Präzi CDN, metadata network, and compile-validated network.
- Two datasets for replication: CDNs for Crates.io and dataset of all generated call graphs.

For the reproducibility of our approach, generated CDNs, and study, we have made the source code, the processing scripts and our data publicly available in a replication package available (Hejderup et al. 2021).

[2] `https://dependabot.com/`
[3] `https://github.com/RustSec/cargo-audit`

2 Background

2.1 Related Work

Analyzing package repositories from a network perspective has become an important research area in light of numerous incidents such as the removal of the `left-pad` package in npm and recent moves to emulate such problems on package dependency networks (Kikas et al. 2017; Kula et al. 2018a; Decan et al. 2019; Zerouali et al. 2018). The aftermath of the `left-pad` incident (Schlueter 2017) in 2016 raised questions on how the removal of a single 11 LOC package downloaded over 575,000 times could break the build for large groups of seemingly unrelated packages in npm. To understand how certain packages exhibit such a large degree of influence in package repositories, Kikas et al. (2017)'s network analysis of three package repositories—npm, Crates.io, and RubyGems—uncovered that package repositories have scale-free network properties (Albert and Barabási 2002). As a result of a large number of end-user applications depending on a popular set of packages (such as the `babel` compiler), these popular yet distinct packages become hubs in package dependency networks. Packages that act as hubs are not isolated packages; they also depend on small and common utility packages such as `left-pad` that appear as transitive dependencies for end-users. By reversing the direction of package dependency networks, (Kikas et al. 2017) identify that utility packages are highly central in package dependency networks with the power to affect more than 30% of all packages in the studied repositories.

In a comprehensive study of the evolution of package repositories, Decan et al. (2019) observe that three out of seven studied repositories have superlinear growth of transitive relationships, forming and strengthening new network hubs over time. Half of the packages in Crates.io, npm, and, NuGet had in 2017 at least 41, 21, and 27 transitive dependencies, nearly two times more than their respective number in 2015. Although Decan et al. (2019) finds that the number of dependencies a developer declares in an application remains stable over time, the increasing number of transitive relationships in package repositories is still an active phenomenon after the `left-pad` incident. Apart from understanding the structure and evolution of package repositories, researchers have also studied known security vulnerabilities (Decan et al. 2018a; Zimmermann et al. 2019), maintainability (Valiev et al. 2018; Cogo et al. 2019; Zerouali et al. 2018), software reuse (Abdalkareem et al. 2019, 2017), and more recently breaking changes (Mezzetti et al. 2018; Mujahid et al. 2020) from a network perspective. Zimmermann et al. (2019) report that 40% of npm include a package with a known vulnerability, suggesting that npm forms a large attack surface for hackers to exploit. Despite developer awareness on using trivial and simple packages after the `left-pad` incident, Abdalkareem et al. (2019) still find a prevalent number of applications depending on trivial packages: 10% of npm and 6% of PyPI applications on GitHub depends on at least one package with less than 35 LOC.

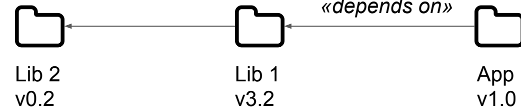
Network analysis of packages commonly makes use of metadata from package manifests to calculate the impact and severity of measured variables. Ponta et al. (2018)'s work on building a security dependency checker using call graphs highlights the limitations of using metadata and the importance of studying package dependencies with a contextual lens. Typically, a subset of an API is vulnerable—not the entire package—and how clients interact with API's is also highly contextual. Zapata et al. (2018) observed through manual analysis that 75% of 60 warned JavaScript projects did not invoke the vulnerability. As an alternative to vulnerability detection through call graphs, Chinthanet et al. (2020) explores the idea of building hierarchical structures of applications and their dependencies for Node.js. To pitch for code-centric instead of metadata-based representations of package repositories, Hejderup et al. (2018) propose dependency networks based on function calls which we concretize in this work.

By embedding function call relationships into package dependency networks, we aim to also bridge the gap between API and package repository research. Notably, PRÄZI could resolve the limitation of studying immediate API calls to include chains of API calls (i.e., transitive calls) such as in Robbes et al. (2012)'s work on determining the ripple effects on deprecated APIs in the Smalltalk ecosystem. Similarly, combining qualitative studies such as looking into deprecation (Sawant et al. 2018c,a), breakages (Raemaekers et al. 2017; Xavier et al. 2017; Bogart et al. 2016), and migration patterns (Zhong et al. 2010; Nguyen et al. 2019) with network analyses could provide an additional empirical dimension in such studies. In support of this, Zhang et al. (2020b)'s need-finding study calls for tooling that supports API designers with data-driven recommendations, for example, on when to deprecate an API.

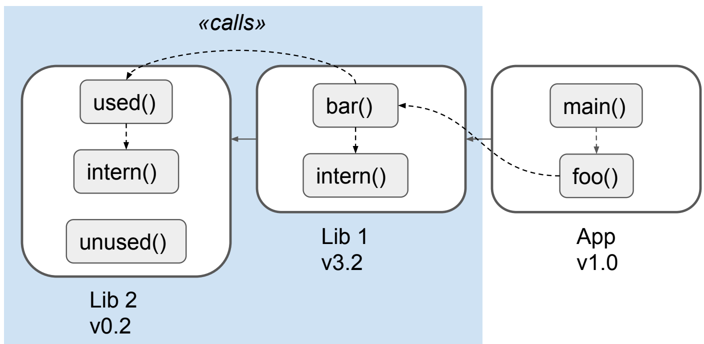
2.2 Rust Programming Language

Rust is a relatively new (first stable release 1.0 in 2015)[4] systems programming language that aims to combine the speed of C with the memory safety guarantees of a garbage-collected language such as Java. Rust is also unique because its package management system (CARGO) was designed from the ground-up to be part of the language environment (Katz 2016). CARGO not only manages dependencies but prescribes a build process and a standardized repository layout which helps facilitate the creation of automated large-scale analyses such as PRÄZI. Every CARGO package contains a file called `Cargo.toml`, specifying dependencies on external packages. Moreover, with CRATES.IO, there is one central place where all Rust packages (so-called "crates") live. As of 13 August 2020, CRATES.IO is the fifth most fast-growing package repository hosting over 44,745 packages and averaging 60 new packages per day.[5]

[4] https://blog.rust-lang.org/2015/05/15/Rust-1.0.html
[5] http://www.modulecounts.com/



(a) State of the art: Package-based Dependency Networks.



(b) Our proposal: Call-based Dependency Networks.

Fig. 1: Different granularities of dependency networks.

2.3 Call-based Dependency Networks

We distinguish two kinds of dependency networks, shown in Figure 1: i) coarse-grained *Package-based Dependency Networks* shown in Figure 1a, similar to what dependency resolution tools (e.g., CARGO or MAVEN) build internally or what researchers have used in the past, and ii) fine-grained *Call-based Dependency Networks* shown in Figure 1b, which we advocate in this paper.

Figure 1a models an example of an end user application `App`, which directly depends on `Lib1` and transitively depends on `Lib2`. In such a PDN, each node represents a versioned package. An edge connecting two nodes denotes that one package imports the other, for example `App 1.0` depends on `Lib1 3.2`.

Figure 1b consists of three individual call graphs for `App`, `Lib1`, and `Lib2`. Each of these call graphs approximate internal function calls in a single package. Every node represents a function by its name. The edges approximate the calling relationship between functions, e.g., from `main()` to `foo()` within `App` in Figure 1b. However, the function identifiers bear no version, nor do they have globally unique identifiers (e.g., there are two `intern()` functions in Figure 1b). We merge these two graph representations to produce a CDN:

Definition 1 *A **Call-based Dependency Network** (CDN) is a directed graph $G = \langle V, E \rangle$ where:*

1. *V is a set of versioned functions. Each $v \in V$ is a tuple $\langle$`id,ver`$\rangle$, where `id` is a unique function identifier and `ver` is a float value depicting the version of the package in which `id` resides.*
2. *E is a set of edges that connect functions. Each $\langle v_1, v_2 \rangle \in E$ represents a function call from v_1 to v_2.*

Applying the above definitions, the function `used()` in Figure 1b becomes a node with the fully qualified identifier ⟨`Lib2::used, 0.2`⟩ $\in V$. The dependency between `App` and `Lib1` is represented as ⟨⟨`App::foo, 0.1`⟩, ⟨`Lib1::bar, 3.2`⟩⟩ $\in E$.

CDNs offer a white-box view of the more coarse-grained PDNs. In particular, we can see that `unused()` is never called. If `unused()` was affected by a vulnerability, we can deduce from Figure 1b that we should *not* issue a security warning for `App`, since it does not use the affected functionality. In contrast to the CDN, the PDN in Figure 1 by its nature cannot provide such a fine-grained precision level.

3 Präzi: Generating CDNs from Package Repositories

In this section, we describe a generic approach, PRÄZI, to systematically construct CDNs for package repositories. PRÄZI can be applied to any programming environment that features i) a way of expressing dependency information between packages, and ii) tooling to generate call graphs for a package.

PRÄZI constructs a CDN in a two-phase process illustrated in Figure 2. In the first phase, *Call Graph Generation* (step [1], [2], and [3]), PRÄZI generates a static dataset of annotated call graphs from packages in a repository. In the second phase, *Temporal Network Generation* (step [4] and [5]), PRÄZI first generates an intermediate package dependency network by resolving dependencies between packages at a user-provided timestamp t, and then unifies the call graphs of resolved packages into one temporal call-based network, the CDN_t.

3.1 Call Graph Generation

Local Mirror Package managers keep an updatable index of package repositories to lookup available packages and their versions. PRÄZI uses such indices to extract and download available packages in step [1] in order to create local mirrors of repositories (i.e., clones of repositories). A minimal local mirror needs to contain the manifest and publication (or creation) timestamp for each version of a package.

Package Call Graphs A call graph is a data representation of relationships between functions in a program and serves as a high-level approximation of its runtime behavior (Ryder 1979; Ali and Lhoták 2012). From a static analysis

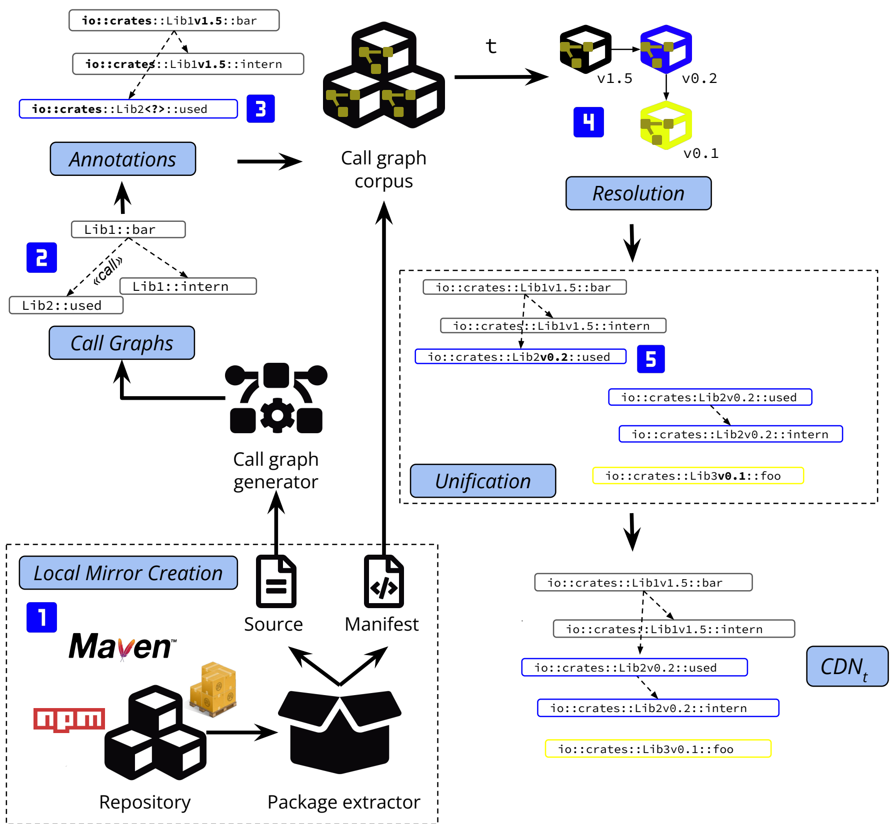


Fig. 2: Generic approach to generate CDNs from package repositories

perspective, a call graph is useful for investigating and understanding interprocedural communication between code elements (i.e., how functions exchange information). In PRÄZI, we view a call graph as a partial graph of a resulting CDN. We increase the scope of a call graph from a single package (i.e., program) to a package and its dependencies. We denote inter-package function relationships as the actual specific code resources that packages use between each other (i.e., a dependency relationship at the function granularity) and are first-class citizens in CDN analyses. The call from `Lib1::bar` to `Lib2::used` in [2] exemplifies an inter-package function relationship. PRÄZI requires nodes in call graphs to have function identifiers with fully resolved return types and arguments.

In the presence of dynamic features, such as virtual dispatch or reflection, there are implications to the precision and soundness of call graphs that indirectly also affect generated CDNs. Theoretically, it is impossible to have both a precise and sound call graph of a program. Thus, PRÄZI uses *soundy* call graph algorithms that follow a best-effort approach for the resolution of

most language features (Livshits et al. 2015). Precise yet unsound call graph algorithms could miss actual inter-package function calls, making certain dependency analyses (e.g., security) of CDNs incomplete. Examples of soundy call graph algorithms for typed languages include subclasses of Class Hierarchy Analysis (CHA) (Tip and Palsberg 2000; Sundaresan et al. 2000) and Points-to analyses (Steensgaard 1996; Shapiro and Horwitz 1997; Emami et al. 1994) such as k-CFA (Shivers 1991). In the case of untyped languages such as Python or JavaScript, a middle-ground is hybrid approaches combining both dynamic analysis and static analysis such as Alimadadi et al. (2015)'s Tochal or Salis et al. (2021)'s PyCG.

Annotating Call Graphs To prepare call graphs for unification, we need to rewrite function identifiers in each package call graph so that they are globally unique. Without annotating function identifiers, inconsistencies can arise from packages that have identical namespaces and multiple versions of the same package in a dependency tree. Präzi solves these issues by annotating the function names, return types, and argument types in function signatures with three components: i) repository name, ii) package name, and iii) static or dynamic (i.e., constraint) package version.

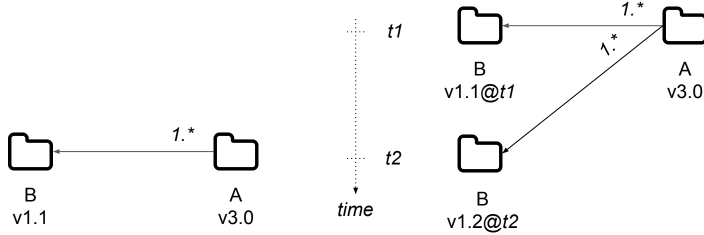
For each function signature in a call graph of a package version, Präzi maps each type identifier found in the signature to the package that declares it. There are three potential mappings of a type identifier to a package that do not reside in the standard library of the language:

- **Local**, resulting in an annotated qualifier with the repository name, and its package name and version as exemplified in `io::crates::Lib1v1.5::bar`.
- **Dependency with a static version**, resulting in an annotated qualifier with the repository name, and the name and version of the dependency.
- **Dependency with a dynamic version**, resulting in an annotated qualifier with the repository name and name of the dependency. However the version is missing as exemplified in `io::crates::Lib2<?>::used` in [3].

The first two mappings denote a resolved type annotation, and the last one is an unresolved type annotation. Function identifiers with unresolved type annotations have their dynamic versions resolved to a specific version at dependency resolution time (i.e., at the *Temporal Network Generation* phase). Finally, Präzi splits the annotated call graph into two sections, one immutable section containing resolved function signatures, and another section containing unresolved function signatures. The annotated call graphs are then stored in a dataset. The final dataset should contain all downloaded packages that include creation timestamp, manifest file, and annotated call graph with global identifiers.

3.2 Temporal Network Generation

Retro-active Dependency Resolution To study the evolution of the relationships between packages in a repository, we perform retroactive dependency



(a) Package A depends on B version *1.**. (b) Full dependency resolution tree with time.

Fig. 3: Retro-active dependency resolution

resolution [4] that generates a concrete dependency network valid at a given timestamp t. The use of dynamic versions in package manifests complicates network generation of package repositories. During resolution time of package dependencies, a dynamic version instructs the dependency resolver to fetch the most recent version within its allowed version boundary, making the relationship between packages contemporary. Package A depending on the dynamic version $1.*$ of package B that satisfies any version with a leading *1.* (e.g., *1.0*,*1.8*, or *1.20.2*) in Figure 3a exemplifies a dynamic version. Given that Package B releases version 1.1 at t_1 and 1.2 at t_2 ($t_1 < t_2$) in Figure 3b. At t, where $t_1 < t < t_2$, a dependency resolver will select version 1.1. However, at $t > t_2$, it will select version 1.2, highlighting the temporal changes in package relationships.

Given a timestamp t, PRÄZI creates a subset $mirror_t$ of our local mirror (i.e., copy of the CRATES.IO index) containing packages and versions with a creation timestamp t_c satisfying $t_c \leq t$. Then, for each package version manifest file in $mirror_t$, we resolve its dependencies using a dependency resolver. Dependency resolvers are usually integrated into package managers and are available as independent libraries.

Call Graph Unification The unification is a two-phase process. In the first phase, we build a resolved dependency tree for each package version in $mirror_t$ and then perform a level-order traversal of each tree to merge call graphs of child nodes with their parent nodes. The output is a unified call graph of statically dispatched function calls for each package version in $mirror_t$. In the merge phase of a parent and a child call graph, we complete the unresolved function identifiers in the parent call graph with the resolved version available in the child node. The function `io::crates::Lib2v0.2::used` in [5] replaces the unresolved function `io::crates::Lib2<?>::used` in [3] with $v0.2$.

In the second phase, we need to deal with dynamically dispatched functions and localize call targets across package boundaries. To illustrate this process, we introduce the following scenario: Package `A` depends on package `B` and package `C`. Both `B` and `C` depend on the library `serde`. Furthermore, `B`

has a class `Foo` that implements the function `serialize()` in the `Serialize` interface of `serde`. `C` has a function called `bar()` that takes a `Serialize`-like object as an argument and invokes the dynamically-dispatched `serialize()` call on the object.

Before merging the call graphs (i.e. first phase), `bar()` is only aware of call targets that are within `C`. In this example, there are no call targets available (i.e., there is no function implementing `serialize()` in `C`). Thus, in the second phase, we search for other compatible function implementations across packages that are available after merging their call graphs. Here, we would create a call target from `bar()` in `C` to the `serialize()` implementation in `Foo` in `B`. It is possible that `A` may never pass an object of `Foo` from `B` to function `foo()` in `C` in practice. However, the second phase is necessary to ensure that dynamically dispatched functions remain sound after merging all call graphs together.

After constructing a package-level call graph for each package version in $mirror_t$, we merge all partial call graphs into a single CDN. The process consists of aggregating all package-level call graphs and then merging them to remove duplicate nodes and edges. The result is a CDN corresponding to the package repository at the given timestamp t.

4 Implementing Präzi for Crates.io

We implement PRÄZI for CRATES.IO, the official package repository for Rust. Unlike mainstream package repositories such as MAVEN CENTRAL, PyPI, NPM, and NuGet, CRATES.IO do not host pre-built binaries but the source code of its packages. To generate call graphs for Rust packages, we need to first perform a large-scale compilation of CRATES.IO and then extract call graphs from generated binaries. Attempting to reproduce the build of a piece of software is known to be challenging (Sulír and Porubän 2016), Tufano et al. (2017)'s compilation of 219,395 Apache snapshots yielded a success rate of 38%, and Martins et al. (2018)'s compilation of 353,709 Github Java projects yielded a success rate of 56%. An overall low success rate could potentially endanger representative studies of CRATES.IO.

In the remainder of this section, we describe key implementation choices and results from our large-scale compilation of CRATES.IO.

Creating a local mirror We clone a snapshot of CRATES.IO's official git-based index[6] at revision `6c550c8` (14th February 2020) containing 35,896 packages, 208,023 releases, and 1,151,001 dependency relationships. By validating the dependency specification in the index for invalid names or dependency constraints, we can save resources by avoiding building broken releases. We identify 1,506 releases from 201 packages having dependencies that do not match existing packages, and 5,667 releases from 4,427 packages having dependencies with unsolvable constraints (i.e., no available versions for the constraint)

[6] `https://github.com/rust-lang/crates.io-index`

The documentation hosting service for CRATES.IO, `Docs.rs`,[7] provides Rust users API documentation for every published package release. In addition to automatically generating documentation for package releases, `Docs.rs` also documents the build log and compile status publically. We create a web scraper that extracts the build status on `Docs.rs` for each release in our dataset. In total, we found that 43,893 indexed releases belonging to 10,154 packages have build failures, amounting to 20% of CRATES.IO. In addition to the CRATES.IO index, we use `Docs.rs` as externally validated metadata source in our study.

After subtracting build failures and invalid dependency specifications, our final index amounts to 156,484 releases from 29,480 packages. Lastly, we use the official API at `https://crates.io/api/v1/crates` to download all packages and their creation timestamp (not available in the index).

Choosing a Call Graph Generator There are two approaches for constructing a call graph from a Rust program, the higher-level LLVM analysis,[8] and the lower-level MIR analysis (Matsakis 2016). Rust functions and its calls are either of monomorphized (i.e., static dispatch) or virtualized (i.e., dynamic dispatch) nature. From the documentation[9] and a comprehensive benchmark (Triantafyllou 2019), we can learn that there are two monomorphized features, `macros`[10] and `generic functions`, and two virtualized features, `trait Objects`,[11] and `function pointers`,[12] that dispatch functions in Rust.

As part of the output in compilation of Rust programs, we can use the optionally generated LLVM IRs[13] to build call graphs using the LLVM call graph generator[14] or `cargo-call-stack` (Aparicio 2019). Due to the absence of Rust-specific type information in LLVM IRs,[15] call graph generators can only resolve monomorphized features and cannot provide complete type information needed in PRÄZI. By analyzing at the MIR level instead of the LLVM level, `rust-callgraphs`[16] offers a more feature-complete call graph by implementing a CHA algorithm and is our choice for building CDNs. In addition to monomorphized features, it can resolve function calls dispatched through `Trait Objects`, making it a more soundy choice over the LLVM-based call graph generators. Although `rust-callgraphs` does not support `function pointers`, it is a negligible trade-off as the documentation[17] state that `function pointers` are mostly useful for calling C code from Rust.

[7] https://github.com/rust-lang/docs.rs
[8] https://llvm.org/docs/Passes.html
[9] https://doc.rust-lang.org/book/ch03-03-how-functions-work.html
[10] https://doc.rust-lang.org/stable/reference/macros.html#trait-objects
[11] https://doc.rust-lang.org/stable/reference/types.html
[12] https://doc.rust-lang.org/book/ch19-05-advanced-functions-and-closures.html
[13] https://doc.rust-lang.org/rustc/command-line-arguments.html#--emit-specifies-the-types-of-output-files-to-generate
[14] http://llvm.org/doxygen/CallGraph_8h_source.html
[15] https://github.com/rust-lang/rust/issues/59412
[16] https://github.com/ktrianta/rust-callgraphs
[17] https://rust-lang.github.io/unsafe-code-guidelines/layout/function-pointers.html

Table 1: Build statistics

Build	**#Releases**	**#Packages**	**Time (hrs)**
CRATES.IO index	208,023	35,896	—
`Docs.rs`	156,484 (-24.78%)	29,480 (-17.87%)	—
`rust-callgraphs`	142,301(-9.06%)	23,767 (-19.38%)	10 days

For annotating call graphs, the metadata in call graph nodes contains package information and access identifiers. Moreover, the complementary type hierarchy output contains complete type information for creating resolved function identifiers. We also keep the edge metadata that includes dispatch information (i.e., static, dynamic, or macro) in the annotated call graphs.

Large-Scale Compilation of CRATES.IO Some Rust packages depend on external system libraries such as `libavcodec` or `libxml2` to successfully compile. Knowing which external libraries to install for compiling such packages is a manual and tedious process. Luckily, the Rust infrastructure team maintains a Docker image, `rust-lang/crates-build-env`, that bootstraps a Rust build environment pre-installed with community curated systems libraries, increasing the chances for successful compilations. We use `Rustwide`, an API for spawning Rust build containers, and configure it to use `rustc 1.42.0-nightly` compiler together with `rust-callgraphs`'s compiler plugin. After compilation, we use the analyzer component in `rust-callgraphs` to generate and store the call graphs in our dataset.

We set up a compilation pipeline on four build servers running 34 docker containers to compile packages and build call graphs. It took 10 days to complete it. Table 1 shows the compilation results in comparison with index entries and `Docs.rs` results. Overall, our call graph corpus (CG Corpus) has a call graph for 90% of all compilable versions (70% of all indexed versions) and at least one version for 80% of all packages built by `Docs.rs`. The high success rate showcases the practical feasibility of PRÄZI for CRATES.IO.

Dependency Resolution For each CARGO.TOML manifest in our downloaded dataset, we extract dependencies intended for source code use. These include library dependencies (i.e., `[dependencies]`), platform-specific dependencies (i.e., `[target]`), and also enabled optional dependencies in `[features]`. Both Kikas et al. (2017) and Decan et al. (2019) do not take into account both enabled optional dependencies and platform-specific dependencies, considering only library dependencies when analyzing CRATES.IO.

The CARGO.TOML manifest supports specifications of dependencies using the `semver` schema (Preston-Werner 2013). A version is a three-part version number: major version, minor version, and patch version. An example of a version is 1.0.0. An increase in the major number denotes incompatible changes, an increase in the minor number denotes backward-compatible changes, and

an increase in the patch number denotes small bug fixes. With the support of range operators (i.e., dynamic version) in dependency specifications such as caret (e.g., $\wedge$1.0.0), tilde (i.e., $\sim$ 1.0.0), wildcard (e.g., 1.$*$), and ranges (e.g., $>$ 1.0.0. $<=$ 2.0.0), the dependency resolver in Cargo will attempt to resolve the latest version satisfying the constraint. When multiple constraints of the same dependency appear in the dependency tree, Cargo first attempts to find the most recent version satisfying all constraints. For example, for the two constraints, `log` *0.4.** and `log` *0.4.4*, the dependency resolver will resolve `log` *0.4.4*. However, for example, if the resolver has to resolve `log` *0.4.** and `log` *0.5.**, there is no single compatible version that matches both constraints. Instead, the resolver will include two versions of the same dependency (e.g., `log` *0.4.4* and `log` *0.5.5*) through name mangling to avoid conflicts (Katz 2016). The resolution strategy of having multiple versions of the same dependency is similar to npm.[18]

To emulate dependency resolution in Rust, we use the native Rust-implementation of the `semver` library for use in Python by invoking its native implementation through FFI (Foreign Function Interface) bindings. Thus, we resolve dependencies and their constraints using the same library as the Cargo package manager. For every timestamp t in the CDN generation process, we set the resolution to solve the latest version available at t satisfying the constraint.

5 Structure and Evolution of the Crates.io CDN

We address three research questions to contrast the similarities and differences when using three different network sources (i.e., metadata, compile-validated metadata, and control-flow data) for characterizing the structure and evolution of Crates.io. In addition to comparing the networks, we also investigate how reliably package-based dependency networks mirror the use of dependencies in the source code.

5.1 Research Questions

RQ1: What are the network characteristics of* Crates.io*?

We characterize the calling relationship between packages in Crates.io, and then identify various influential packages featuring a high number of callers and callees within the networks. Specifically, we describe our data corpus and the degree distribution to gain an overall understanding of the direct relationship between functions for a large package repository such as Crates.io.

[18] `http://npm.github.io/npm-like-im-5/npm3/dependency-resolution.html`

RQ2: How does* Crates.io *evolve?
The frequent number of new package releases and the adoption of `semver` range operators in dependency specifications make the relationship between packages highly temporal in Crates.io. We capture these dynamics using both a package-level perspective and the more fine-grained, function-level perspective. In comparison to previous studies (Kikas et al. 2017; Decan et al. 2019), we use three different sources, namely metadata, compile-validated metadata, and control-flow data, to understand their differences and similarities for package-based dependency analysis.

As all our snapshots deviate from a normal distribution according to Shapiro-Wilk ($p < 0.01 \leq \alpha$), we use the non-parametric Spearman correlation (ρ) coefficient for correlation analysis. Using Hopkins's guidelines (Hopkins 1997), we interpret $0 \leq |\rho| < 0.3$ as no, $0.3 \leq |\rho| < 0.5$ as a weak, $0.5 \leq |\rho| < 0.7$ as a moderate, and $0.7 \leq |\rho| \leq 1$ as strong correlation We answer the following **sub-RQs** using a package-level and call-level perspective:

RQ2.1: How do package dependencies and dependents evolve?
RQ2.2: How does the use of external APIs in packages evolve?
RQ2.3: How prevalent is function bloat in package dependencies?
RQ2.4: How fragile is Crates.io to function-level changes?

For deciding on reasonable time points for evolution studies of package repositories, we include a guideline with analysis in Appendix A.

RQ3: How reliable are dependency networks?

A dependency network approximates how packages use each other in a repository. Both metadata-based networks and call-based networks have trade-offs and limitations that affect how reliable they estimate actual package relationships. To understand how accurate these networks are in practice, we perform a manual analysis of 34 random cases where a metadata-based and call-based dependency network infers relationships differently. The cases involve both direct and transitive package relationships.

5.2 RQ1: Descriptive Analysis

5.2.1 Summary of Datasets

Before investigating the calling relationship among packages in Crates.io, we first describe our datasets of generated call graphs (i.e., **CG Corpus**) and our largest CDN dated February 2020 in Table 2. After removing all function calls to the standard libraries of Rust, the call graph corpus has over 121 million functions and 327 million function calls from $142,301$ compiled releases. When merging call graphs into a CDN, we generate a compact representation with

Table 2: Summary of Datasets

	CG Corpus	**CDN Feb'20**
Functions	**121,825,729**	**44,190,643**
... public access	46,236,696	20,157,155
... private access	75,589,033	24,033,488
Call edges	**327,535,934**	**216,239,360**
Intra-Package Calls	**169,579,315**	**102,136,956**
... macro invocation	693,148	356,329
... static dispatch	28,570,266	20,650,000
... dynamic dispatch	140,315,901	83,130,627
Inter-Package Calls	**157,956,619**	**114,102,404**
... macro invocation	7,183,797	2,178,547
... static dispatch	29,650,173	13,319,367
... dynamic dispatch	121,122, 649	98,604,490

over 44 million functions and 216 million function calls, a sizeable reduction of 2.5 and 1.5 times of the **CG Corpus** (i.e., functions and calls), respectively.

Table 2 also breaks down function calls into their dispatch type, namely macro, static, and dynamic calls. Notably, nearly 80% of all edges in the **CG Corpus** are of a dynamic dispatch type, followed by static dispatch (18%) and macro invocations (2%). The high number of dynamically dispatched calls in the network indicates that CRATES.IO has a large pool of possible target implementations to virtual functions—not necessarily magnitude more function calls than statically dispatched calls. When comparing the access modifiers between functions, we can see that 40% of all functions inside CRATES.IO are publicly consumable. Also, we can see that calling functions in external packages is widespread in CRATES.IO; half of all the function calls invoke a function from an external package (i.e., inter-package call). Unlike the other two dispatch forms, 91% of all macro dispatched calls exclusively target macros defined in external packages. Overall, the high number of declared public functions and the large degree of inter-package calls indicate that code reuse in the form of functions between packages is a prevalent practice in CRATES.IO.

Function reuse is prevalent; 40% of functions are public and 49% of call edges target a dependency.

5.2.2 Function Call Distribution

Figure 4 presents the degree distribution for all function calls grouped by their dispatch type, and Figure 5 is a narrowed-down version looking at only inter-package function calls. The out-degree of a function is the number of function calls to other unique functions (i.e., number of caller-callee relationships). The in-degree of a function is the number of callers to a function across CRATES.IO (i.e., number of callee-caller relationships). Given a function `a()` in a package,

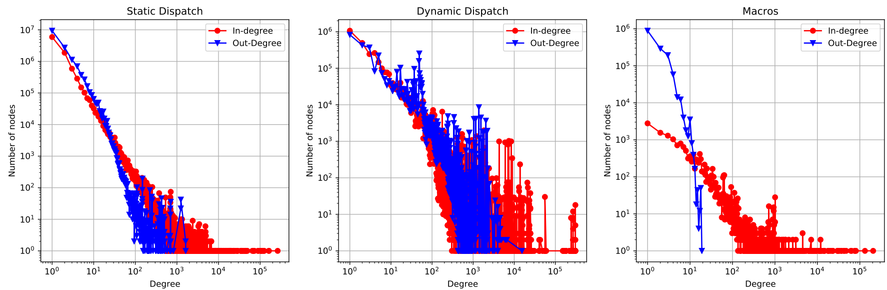


Fig. 4: Degree distribution of all function calls

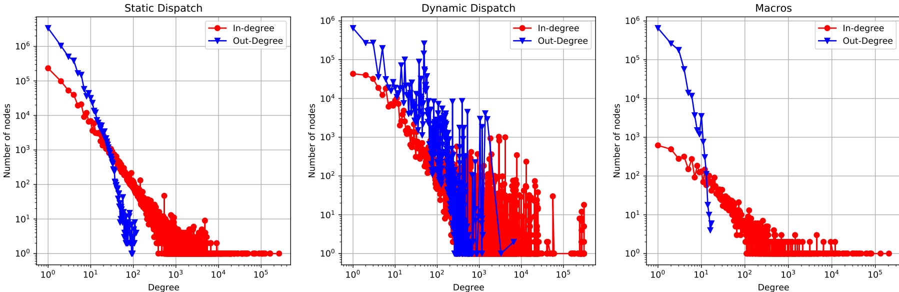


Fig. 5: Degree distribution of inter-package function calls

the out-degree looks at what calls `a()` makes. The in-degree looks at which functions in CRATES.IO call `a()`. As mentioned earlier, inter-package calls are only function calls between packages (i.e., pruning all internal calls). The out-degree distribution for dynamic dispatch represents the number of possible target functions in a virtual method table,[19] and, for static- and macro dispatch, the number of function calls. The in-degree distribution presents the aggregated number of callers for a function (i.e., callee) and implementations of virtual functions for dynamic dispatch, respectively. Overall, we can observe a long tail for both the in-degree and the out-degree of each dispatch mechanism, suggesting that the CRATES.IO CDN is a scale-free network with the presence of a few nodes that are highly connected to other nodes in the network (i.e., hubs). Finally, Tables 3 to 5 describe the top 5 functions with the highest in-degree and out-degree calls per dispatch type. The top 5 list is an aggregation of functions per package. For example, the `serde` package in Table 4 has over 300 serialization functions with an in-degree similar to 264, 281. Thus, we present the top 5 functions as the top most called function(s) per package. In the following, we describe key results for each of the three dispatch forms.

[19] https://alschwalm.com/blog/static/2017/03/07/exploring-dynamic-dispatch-in-rust/

Table 3: The top 5 functions with most statically-dispatched calls

Outdegree			Indegree		
Package	Function	#	Package	Function	#
epoxy	`load_with`	1,625	serde	`missing_field`	264,281
sv-parser-syntaxtree	`next, into_iter`	1,243	log	`max_level`	162,747
python-syntax	`__reduce`	821	vcell	`set`	125,287
rustpython-parser	`__reduce`	720	serde_json	`from_str`	73,171
mallumo-gls	`load_with`	712	futures	`and_then`	65,043

Table 4: The top 5 functions with most dynamically-dispatched calls

Outdegree			Indegree		
Package	Function	#	Package	Function	#
hyperbuild	`match_trie`	15,460	serde	`deserialize_any`	307,976
heim-common	`to_base, from_base`	6,597	serde_json	`from_str`	268,887
uom	`to_base, from_base`	6,045	serde_urlencoded	`deserialize_identifier`	59,737
fpa	`I1F7, I2F6`	3,966	yup-oauth2	`token`	42,737
rtdlib	`deserialize`	2,470	cpp_core	`cast_into`	28,278

Static dispatch The median out-degree for statically dispatched function call is 1 call (mean: 2.25) in both cases and at the 99th percentile being 15 calls (13 calls for inter-package calls). When comparing the out-degree between statically dispatched calls in Figure 4 and Figure 5, we can notice that there are 1865 functions (0.012%) that call more than 100 other internal functions in Figure 4. The highest number of calls made by a single function in both plots is to 1625 local functions and 116 external functions, respectively. The relatively high number of internal function calls among the outliers seems unrealistic at a first glance. Upon manual inspection of the source code of the only two packages having functions with an out-degree greater than 1000 (see Table 3), namely `epoxy`[20] and `sv-parser-syntaxtree`[21], we identify that this is the result of generic instantiations for creating bindings to the `libepoxy` (an OpenGL function pointer manager) and tokens for parsing SystemVerilog files.

The median in-degree for statically dispatched function calls are 1 (mean: 3.6) and the 99th quantile is 24. When omitting all internal calls and considering only inter-package calls, the median is 2 (mean: 24) and the 99th quantile is 208. There are three functions having over $100,000$ external calls in Table 3, `serde` for *serialization*, `log` for *logging*, and `vcell` for *memory management*. While the first two are the most downloaded and depended upon packages in Crates.io, `vcell` stands out for only having nearly 300 dependent packages. After inspection of the source code of those packages for the specific `set` call, we could identify extensive implementations of low-level drivers to interface various microcontrollers such as the `Cortex-M` and `STM32` series.

[20] `https://docs.rs/crate/epoxy/0.1.0/source/`
[21] `https://docs.rs/crate/sv-parser-syntaxtree/0.6.0/source/`

Table 5: The top 5 functions with most macro-dispatched calls

Outdegree			Indegree		
Package	Function	#	Package	Function	#
item	`path_segment`	19	log	`log!`	205,810
fungi-lang	`fgi_module`	18	bitflags	`__impl_bitflags!`	77,848
syn	`path_segment`	17	lazy_static	`__lazy_static_internal!`	64,161
device_tree_source	`parse_data`	17	trackable	`track!`	47,648
npy	`map`	17	serde	`forward_to_deserialize_any_method`	43,063

Dynamic dispatch We use `vtable` to refer to all implementations of a virtual function of a Trait object. In practice, each Trait object points to compatible Trait Implementations (having a `vtable` with function and other member implementations). The median number of function targets function `vtable` is 9 (mean: 42 (all), 32 (inter-package)) for both all function targets and only inter-package function targets. The main deviation is at the 99th percentile, the outdegree for all function targets is 800 for all targets, two times higher than when only considering inter-package function targets. The highest outdegree function in Table 4 is `match_trie` in the package `hyperbuild v0.0.10`, a HTML minification library, having a `vtable` with 15,460 function targets. The function takes as an argument a `trie: &dyn ITrieNode<V>` Trait, invoking `get_child` and `get_value` of the Trait `ITrieNode`. The Trait is implemented for all forms of HTML entities, explaining this high outdegree value. In total, there are 38,352 (< 0.94%) functions that populate a vtable with more than 1000 function targets. Similarly, we can observe 11,906 (< 0.36%) inter-package function calls with over 1000 function targets.

The median in-degree for implementing a virtual (i.e. trait) function is 3 (mean: 53) and the 99th percentile is 608. When only considering inter-package relationships, the median is 3 (mean: 64) and the 99th percentile is 875. As shown in Table 4, the most commonly implemented trait function stems from serializer packages such as `deserialize_any` in `serde`, `from_str` in `serde_json` and `deserialize_identifier` in `toml`. In addition to serialization functions, we can also observe that 42,737 functions implement the trait function `token` in `yup-oauth2` for user authentication with OAuth 2.0.

Macro dispatch When comparing the out-degree for both all and inter-package calls, we can observe a similar trend between them: the median is 1 (mean: 1.7) and the 99th quantile is 6, suggesting that macro-dispatched calls are largely inter-package calls. This resonates with our observations for macro-dispatched calls in Table 2. Looking at functions calling the most number of macros in Table 5, we can observe that the outdegree generally is relatively low in comparison to the other two dispatch types. The function `path_segement` in `item` makes in total 19 macro calls, the highest in CRATES.IO. The median in-degree is 7 (mean: 146) and the 99th quantile is 1427. When only considering inter-package calls, the median is 12 (mean: 391) and the 99th quantile is 6433. We can observe comparable numbers to the in-degree with the other two dispatch types in Table 5. With over

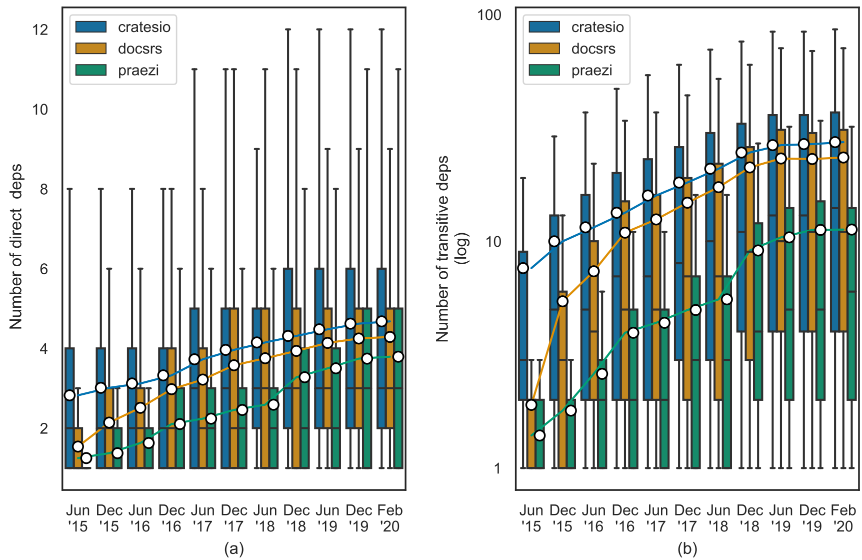


Fig. 6: The evolution of package dependencies on two metadata-based networks, CRATES.IO and `Docs.rs`, and one call-based network, PRÄZI.

200,000 functions in CRATES.IO calling `log!`, it is the most called macro followed by `__impl_bitflags!` and `__lazy_static_internal!`. Generally, we can observe that the top most called macros belong to popular packages in CRATES.IO that are known to simplify logging (`log`), generate bit flag structures (`bitflags`), and wrapping error messages (`quick-error`).

The median function in CRATES.IO makes one static call, one macro call and has a `vtable` with nine function targets. The median function is also dependent upon by one static call, one macro call, and implemented by three functions.

CRATES.IO is a scale-free network, indicating the presence of a handful of functions or hubs that are highly connected to other functions in the repository

5.3 RQ2: Evolution

5.3.1 RQ2.1: How do package dependencies and dependents evolve?

Figures 6 and 7 present the number of direct and transitive package relationships split by network type over time. Each sub-plot also features line plots

showing the mean with a circle for each snapshot. By using three different network representations, we can understand and contrast the differences between the three approximations of dependency relationships.

Direct dependencies Direct dependencies refer to the dependencies that a developer specifies in a package manifest. For each network group in Figure 6a, we see a marginal growth in the median number of direct dependencies over time. The median number of dependencies for a package grew from two to three between 2015-2020 for the Crates.io index network as an example. The median is also similar in the other two networks. Although there are notable differences in the overall spread in the formative years of Crates.io, the growth curve is relatively comparable between the networks. The correlation between the number of direct dependencies between the three networks (normalized) yields a significantly strong $\rho = 0.89$ between 2017 and 2020 (2015-2017: $\rho = 0.71$), indicating that the networks approximate each other.

When comparing the mean between the CDN and the Crates.io index network, we find the average package call at least one function in 78.8%[22] of its direct dependencies. As the Crates.io index network has a higher overall spread than the `Docs.rs` network, and the `Docs.rs` network has a higher overall spread than the CDN, we can derive that the Crates.io network represents an upper-bound and the CDN a lower-bound on the number of direct dependencies. With 75% of all packages having less than six direct dependencies, the results are overall similar to the findings of Decan et al. (2019) and Kikas et al. (2017).

Package maintainers use 2 to 3 direct dependencies and are unlikely to increase their use over time. The three networks have comparable results.

Transitive dependencies Transitive dependencies represent the indirect dependencies of a package after resolving its specified dependencies. In comparison to the direct dependencies, in Figure 6b, we can observe an initial superlinear growth, followed by a period of stabilization (since 2018) for the three networks. The median number of transitive dependencies in 2015 is 5 for the Crates.io index network and 1 for the other two networks. The median number of transitive dependencies grew with a delta of 5 additional packages for the CDN, 9 for the `Docs.rs` network, and 12 for the Crates.io index network in five years. While we can find a similar continuing growth trend to Figure 6a, we observe higher degrees of dispersions between the CDN and the other two networks. The third-quartile in nearly all CDN snapshots is the same or below the median of the other two networks. Thus, half the packages in the Crates.io index network and `Docs.rs` network report a higher number of transitive dependencies than 75% of packages in the CDN. When normalizing the networks and comparing the mean between the CDN and the Crates.io index network in 2020, we find the average package call at-last

[22] after normalizing the networks (i.e., inner join of common packages in all three networks)

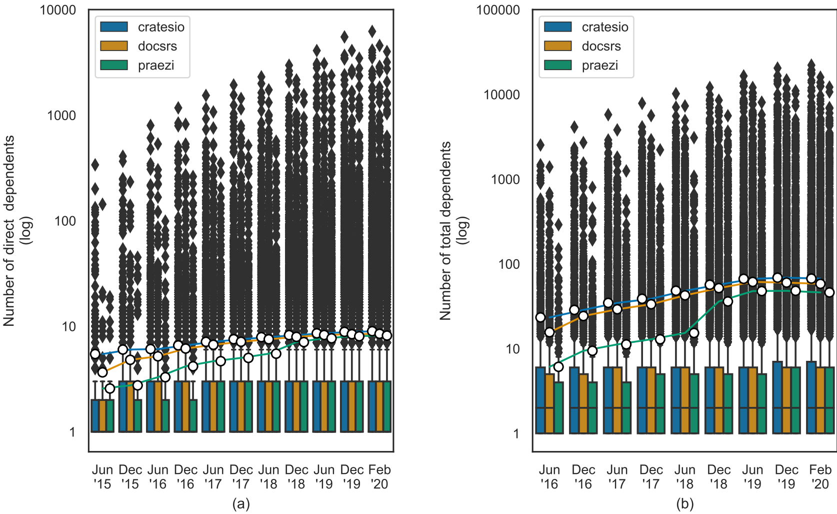


Fig. 7: The evolution of package dependents on two metadata-based networks, CRATES.IO and `Docs.rs`, and one call-based network, PRÄZI.

one function in 40%[23] of all resolved transitive dependencies. The discrepancy indicates substantial differences between call-based and metadata-based networks in network analyses; CDNs will overall report a notably lower number of transitive dependencies than the metadata-based ones.

Finally, the correlation between the number of transitive dependencies between the three networks (normalized) is generally strong, with an average $\rho = 0.84$ between 2017 and 2020 (2015-2017: $\rho = 0.70$). In other words, the more resolved transitive dependencies a package has, the more transitive dependencies it will call (and vice versa). However, we identify a moderate average correlation $\rho = 0.62$ between the number of direct dependencies (i.e., either metadata-based or call-based) and the number of call-based transitive dependencies in 2017-2020. In 2015-2017, we observe a general weaker correlation, with $\rho = 0.47$. Thus, two packages with the same number of direct dependencies are likely to have different number of transitive dependencies.

> The average dependency tree of resolved packages has nearly grown thrice (5 to 17 transitive dependencies) in 5 years. Substantial differences exist between the networks; packages are not calling 60% of their resolved transitive dependencies.

Direct dependents In addition to dependencies, dependents measure the number of consumers a package has. In the context of this study, we consider a

[23] See footnote 22

consumer as an internal consumer (i.e., a package making use of another package within CRATES.IO). Figure 7a presents the number of dependents over time. Irrespective of the network, we can see that the median number of consumers per package remains unchanged at one over time. Similarly, we can also find the interquartile ranges of the networks to be identical from June 2017 and onwards. In that period, the top 25% packages have at least three or more consumers. The correlation between the number of direct dependents for the three networks (normalized) yields a strong $\rho = 0.81$ between 2017 and 2020 (2015-2017: $\rho = 0.75$), indicating (similar to direct dependencies) that the networks closely approximate each other.

When comparing the mean over time, we see a steady growth of the number of direct dependents for all three networks. The growth pattern is a result of a few commonly used packages (e.g., `serde` and `log`) having the largest share of consumers in CRATES.IO (see also Figure 5). The outliers in the boxplot represents the top-most used packages for each network. Here, we can observe notable differences in the range and number of outliers between the networks. The number of top dependent packages in June 2018 is 651 for the CDN, 1245 for the `Docs.rs` network, and 1680 for the CRATES.IO index network. There are 2.5x more top-dependent packages for CRATES.IO than in the CDN. When comparing the top-most dependent packages in each network, the most consumed package has 566 dependents in CDN, 1735 in the `Docs.rs` network, and 2305 in the CRATES.IO index network. Although the gap between the outliers in the networks reduces over time (i.e., from 2.5x to 1.8x in 2020), there are notable differences between the networks when analyzing the top-most dependent packages in CRATES.IO.

Overall, the results are similar to the findings of both Decan et al. (2019) and Kikas et al. (2017), suggesting that an average CRATES.IO package has a relatively constant and low degree of consumers in general. While the networks seem comparable and interchangeable at large, there is a notable discrepancy between the outliers (i.e., topmost used packages in CRATES.IO) in metadata-based networks and call-based networks in earlier snapshots, potentially yielding differences in network analyses of top dependent packages.

The average number of consumers of a package remains at one over time. Similar to direct dependencies, the networks approximate each other (except for top-dependent packages).

Total dependents Figure 7b shows the total number of dependents per package. The total number of dependents include both direct and transitive dependents. We omit both June and December 2015 as these snapshots only have 19 and 47 transitive dependents in the CDN, respectively. Except for June 2016, the median number of total dependents remains constant at two for the three networks. Thus, in addition to the one median direct consumer in Figure 7a, packages also have one median transitive consumer. When looking at the top 25% consumed packages, the number of total dependents ranges from 8 or

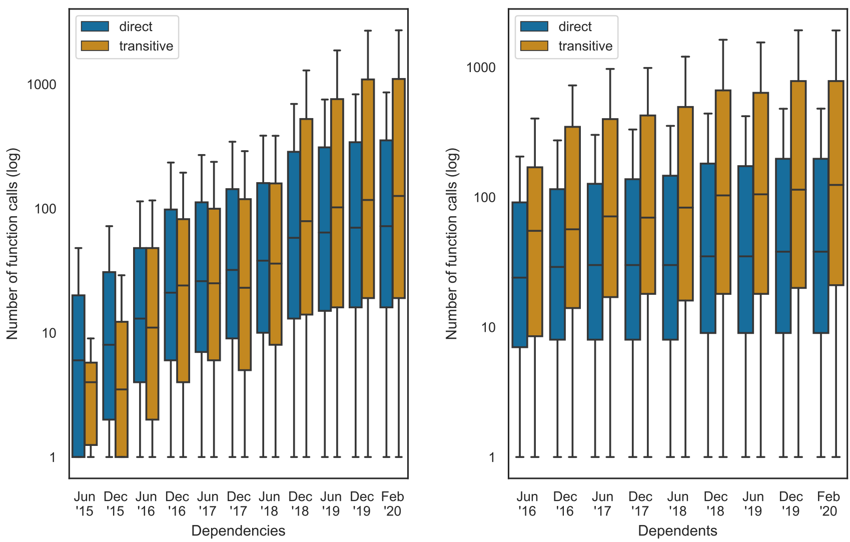


Fig. 8: The evolution of the number of functions calls to dependencies and dependents

more consumers for the CRATES.IO index network and 7 or more consumers for the remaining networks. There is also a slight increase in the overall range at two occurrences for the CDN (Feb'17, Dec'19) and one occurrence for the `Docs.rs` network (Dec'17) and the CRATES.IO index network (Dec'19). When comparing the mean and outliers between the networks, we find a similar growth pattern and gap to Figure 7a.

Similar to transitive dependencies, we also find a general strong correlation between the number of transitive dependents between the three networks (normalized) ($\rho = 0.77$), and also a moderate correlation between the number of direct dependents and transitive dependents ($\rho = 0.54$).

Overall, we see that the total number of dependents remains stable over time with a few cases of gradual increase. Moreover, we see that the distributions of dependents are generally much lower in comparison to the transitive dependency relationships in Figure 6b. Thus, the results indicate that an average package in CRATES.IO has a handful stable number of consumers.

> The average package also has one transitive consumer that remains unchanged over time. Similar to direct top-most dependent packages; indirect consumers are using them to a much higher degree than previously.

5.3.2 RQ2.2: How does the use of external APIs in packages evolve?

Figure 8 describes the evolution of the number of direct and transitive inter-package (i.e., API) calls per package for dependencies on the left-hand side

and dependents on the right-hand side. When looking at the number of calls to dependencies over time, we make two major observations. First, the number of direct and transitive calls to dependencies has an initial superlinear growth, followed by a period where the growth slows down from December 2018 and onwards. From December 2016 to December 2019, the number of direct calls grew from 21 (transitive: 24) to 70 (transitive: 116), a three-fold increase in three years. On average, we also see a growth of 6.6 new function calls to direct dependencies and 12.2 new indirect calls to transitive dependencies every six months. Second, we can see that the median number of transitive calls overtakes the median number of direct calls in December 2018. Our findings unveil that the amount of calls to indirect APIs are comparable in numbers to calls of direct APIs. Recent snapshots further indicate that packages invoke more indirect APIs than direct APIs. The transitive median API calls in December 2019 is 1.6x larger than the median direct API calls.

> The average API usage of transitive dependencies is both greater and comparative to direct dependencies in recent years.

Similar to the total dependents in Figure 6d, we also omit the two snapshots in 2015 due to an insignificant number of transitive dependents. Generally, we can observe a continuous growth of the number of direct and transitive consumers of package APIs over time. The median number of consumer grew from 25 callers in 2015 to 38 callers in 2020, an average growth of 1.6 new functions per year. The median of indirect consumers is larger than that of direct consumers, growing from 55 callers in 2015 to 124 callers in 2020, an average of 8.75 new functions every six months. When comparing the growth pattern between direct and transitive dependents, the gap between the median of direct dependents and transitive dependents expands over time. Moreover, we also find that the interquartile ranges and overall range is greater for transitive dependents than for direct dependents in all snapshots. A package with transitive dependents is likely to have more indirect callers than direct callers of their APIs. Notably, the median number of transitive dependent callers (median: 114) is three times larger than the median of direct dependent callers (median: 38) in 2020. When also taking into account the findings of transitive dependency callers, our results strongly indicate that indirect users of library APIs is both highly prevalent in Crates.io, and comparable to direct users of library APIs. Despite the largely unchanged number of direct and total dependents (See Figure 7) over time, we see indications that developers are increasingly using more APIs over time.

> Packages with transitive consumers have three times more API callers stemming from indirect consumers than direct consumers.

Below, we summarize the two perspectives of package relationships using both the metadata-based results with function-based results:

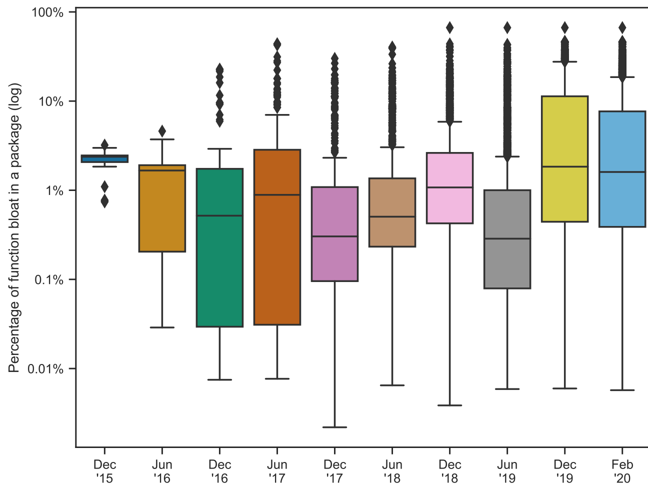


Fig. 9: Percentage of co-existing functions (i.e., bloat) in Crates.io packages

Dependencies: Packages depend on an increasing number of transitive dependencies over time. Package maintainers, however, are not declaring more dependencies. Although there is an increase of new direct and indirect API calls to dependencies over time, roughly 60% of all resolved transitive dependencies are not called.

Dependents: The number of total dependents, one direct and one transitive consumer, remains constant over time. However, consumers have a growing number of callers over time. For packages with transitive consumers, there is a higher number of calls stemming from indirect callers than direct callers.

5.3.3 RQ2.3: How prevalent is function bloat in package dependencies?

Packages depending on a growing number of external packages are also likely to introduce dependency conflicts. Conflicts arise when a dependency resolver is unable to eliminate the co-existence of a package in a dependency tree due to version incompatibility. For example, a resolver may arrive that there is no overlapping version when two packages in a dependency tree depend on package `A` where the former specifies a version constraint $1.*$ and the latter $2.*$. Rust's Cargo package manager avoids such conflicts by allowing multiple versions of the same package to co-exist in a dependency tree using *name mangling* techniques (Katz 2016). A potential drawback of this strategy is the risk of bloating binaries due to multiple copies of identical yet obfuscated functions.

As a proxy for function bloat in binaries, we calculate the percentage of co-existing functions for all public functions in Crates.io. We denote a co-existing function as multiple copies of identical function identifiers loaded from different versions of the same package. It is important to note that the measure is an estimation and does not guarantee the semantic equivalence of functions. Before measuring the percentage of co-existing functions, we first inspect the presence of co-existing functions in all Crates.io packages. On average, we find packages having at least one co-existing function to be 5.4% of Crates.io in Dec 2015-Dec 2017 and 28% of Crates.io in Jun 2018-Feb 2020. There are no packages with co-existing functions in June 2015. Largely non-existent in the formative years of Crates.io, we find that function co-existing among dependencies is relatively prevalent in recent years.

Among packages having co-existing functions, Figure 9 breaks down the percentage of co-existing functions in dependencies of packages over time. We can observe that the median fluctuates between 0.3% and 1.6% over time, indicating a constant yet insignificant amount of function co-existence in packages. 75% of all packages range between 1 to 10% co-existing functions in their dependencies, suggesting that a majority of packages have a small amount of possible bloat in their binaries. Thus, bloating of binaries from co-existing dependency functions are highly unlikely for packages with at least one co-existing function in Crates.io.

Finally, we find a small minority (i.e., outliers) of packages with a high degree of possible function bloat between December 2018 and February 2020. The package reporting the highest bloat of this time frame is `downward` with 67% bloat. However, it is an invalid outlier as it has a circular dependence on itself.[24] Thereby, the two packages with highest bloat is `const-c-str-impl` and `mpris` with 43% and 46% bloat, respectively. Upon manual inspection of their respective dependency tree, we identify that the packages have a dependence on multiple versions of `proc_macro`, `quote`, `syn`, and `unicode_xid`, common libraries for creating procedural macros. For example, `mpris` indirectly uses four different versions of `syn` and `quote`.[25]. We also make similar observations in three other outliers: `js-object` (33%), `js-intern-proc-macro` (41%), and `mockers_derive` (43%). Further investigation could perhaps reveal whether the combination of certain procedural macros libraries are highly likely to always result in bloated dependency tree configurations.

> 28% of all packages in Crates.io have a co-existing function in their dependencies. Among those packages, between 1-10% of imported functions from dependencies are bloated.

5.3.4 RQ2.4: How fragile is Crates.io *to function-level changes?*

Our goal is to identify packages that indirectly reach most of Crates.io and understand the differences and similarities in using different networks

[24] `https://crates.io/crates/downwards`
[25] `https://docs.rs/crate/mpris/2.0.0-rc2/source/Cargo.lock`

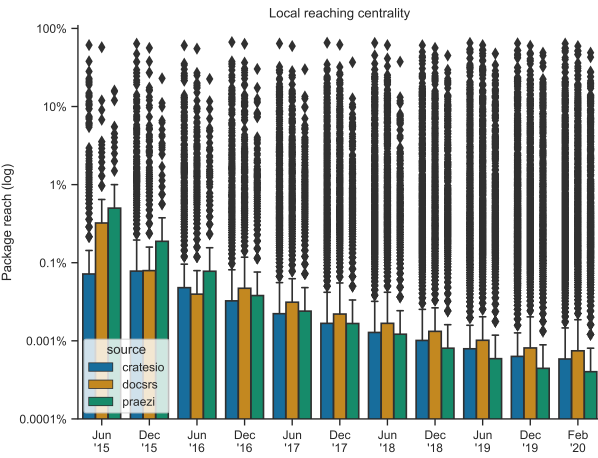


Fig. 10: Distribution of Package Reachability

Table 6: Most central APIs in the largest component in Dec 2015, Dec 2017, and Feb 2020

Dec 2015 (size: 534)			Dec 2017 (size: 6,004)			Feb 2020 (size: 24,857)		
Package	Function	Reach	Package	Function	Reach	Package	Function	Reach
pkg-config::0.3.6	`find_library`	10%	log::0.3.8	`__log`	16%	log::0.4.10	`max_level`	30%
gcc::0.3.20	`Build::new`	6%	libc::0.2.34	`memchr`	12%	serde::1.0.104	`next_element`	24%
libc::0.2.1	`memchr`	6%	lazy_static::0.2.11	`get`	11%	bitflags::1.2.1	`__fn_bitflags`	23%
log::0.3.4	`__static_max_level`	6%	bitflags::1.0.1	`__fn_bitflags`	8%	lazy_static::1.4.0	`get`	21%
bitflags::0.3.3	`bitflags`	3%	unicode-width::0.1.4	`width`	8%	libc::0.2.66	`sysconf`	18%
gcc::0.3.20	`Build::compile`	3%	serde::1.0.24	`deserialize`	6.3%	libc::0.2.66	`isatty`	18%
log::0.3.4	`log::macros log`	6%	lazy_static::0.2.11	`lazy_static`	5.95%	memchr::2.3.2	`memchr`	18%
gcc::0.3.20	`compile_library`	4%	byteorder::1.2.1	`write_u32`	5.1%	itoa::0.4.5	`Buffer::new`	17.5%
time::0.1.34	`precise_time_ns`	2.5%	libc::0.2.34	`localtime_r`	5%	ryu::1.0.2	`Buffer::format_finite`	17%
libc::0.2.1	`sysconf`	2.5%	time::0.1.38	`num_seconds`	4.1%	serde_json::1.0.48	`from_str`	10%

for impact analyses of package repositories. We use the local reaching centrality (Mones et al. 2012) to measure the reach of individual packages in the CDN, compile-validated metadata (i.e., `Docs.rs`), and regular metadata (CRATES.IO) networks. With reach, we measure the fraction of CRATES.IO packages that depend on a particular package (i.e., its transitive dependents).

Figure 10 presents the evolution of the reach of each package per network. When comparing the third-quartile between the snapshots, we can observe a gradual decrease in reachability over time. The decrease is a result of new packages being added to the network and at the same time not being widely used by other packages. The top 25% of the distribution of the CRATES.IO index network has a ten-fold decrease of 0.07% in June 2015 to 0.008% in June 2019. Both the CDN and `Docs.rs` distribution also follow a similar pattern. In comparison to recent years, the higher reach of packages in the formative years reflects the small network size. In the remaining 75% of packages, they have no or limited reach of CRATES.IO irrespective of network choice, indicating that a majority of packages do not exhibit any influence in CRATES.IO. However, we can observe that the range and number of outliers expand over time, indicating that there is an increasing number of packages that exhibit a degree of influence

in CRATES.IO. The number of outliers with greater than 10% reachability grew from 19 (`Docs.rs`/CDN: 3) to 92 (`Docs.rs`: 80, CDN: 66) packages as an example.

For each snapshot, we can see that the top-most outlier and the number of outliers is lower than that of the metadata-based network in each network. The most reachable package in June 2019 reaches 65% in the CRATES.IO index network, 61% in the `Docs.rs` network, and 47% in the CDN network.

Upon inspection of the top 10 highest reaching outliers in each network, we see that a similar set of packages such as `libc`, `log`, `lazy_static`, and `bitflags` remains prominent over time across the networks. These packages are also among the most directly called packages in Section 5.2.2. `libc`, one of the most downloaded packages in CRATES.IO, is the package exhibiting the highest all-time influence in CRATES.IO. There are also packages in decline: `rustc-serialize`, a serializer package, decreased in reach from its peak of 17% in 2016 to 2% in 2020. A potential explanation for its decline could be the adoption of `serde`, a rivaling serializer package, that grew its reach from 6% in 2016 to 42% in 2020.

We derive the ten most influential APIs by measuring the local reach centrality on functions of the CDN for 2015, 2017, and 2020 in Table 6.[26] Although `libc` exhibit the highest reach at the package-level, functions in `log` or `serde` exhibit higher influence than individual functions in `libc`. Moreover, we can see that `libc`, `log`, and `bitflags` have remained important since the inception of CRATES.IO. However, we can observe that the most called function changes over time. For example, `log` reports three distinctly different API functions. A possible explanation could be that new features or best practices over time change the use of APIs. Finally, we can also see a new fast-growing entrant in 2020: `serde` is second to `log`.

> A large majority of packages in CRATES.IO have no or limited reachability; a handful of packages are reachable from 47% of CRATES.IO, and single functions are reachable from 30% of CRATES.IO.

5.4 RQ3: Reliability

We identify two occurrences with significant differences between the studied networks, namely transitive dependencies and outliers in the top-most dependent packages in **RQ2.1**. These differences have practical implications on dependency analysis use cases. For example, security-based dependency analysis such as `cargo-audit` would generally favor soundness over precision. Failing to account for an actual dependency relationship could lead to vulnerabilities being undetected. On the other hand, automated dependency updating such as GITHUB's `Dependabot` would favor precision over soundness. False-positive updates steal valuable time from developers (Mirhosseini and Parnin

[26] due to presentation reasons, we showcase for only three years

2017; Beller et al. 2016). Thus, our goal in **RQ3** is to obtain an understanding of how accurate and reliable metadata-based and call-based networks are in estimating actual relationships between packages.

Selection As packages can have many transitive dependencies and have complex use cases, manually mapping out how packages use each other in a dependency tree is a tedious and error-prone task. Attempting to scale the analysis to the entire Crates.io is also impractical. Thus, we sample dependency relationships in packages where both the metadata-based networks and the call-based networks report differently (e.g., between a package and a dependency, the metadata-network reports an edge between them, and the call-based network does not). We can then focus our manual investigation on whether call-based networks are missing function calls due to limitations with static analysis or whether metadata-based networks over-approximate unused dependencies. Moreover, analyzing a narrow set of direct and transitive dependencies further reduces the overhead of manually tracking uses of code elements across packages and their dependencies.

In the span of five workdays, we randomly sampled and reviewed 34 cases, 7 cases involving direct relationships, and 27 cases involving transitive relationships.

Review Protocol We initiate the review by first finding import statements of the direct library for the package under analysis and then track successive uses of imported items in variable assignments and definitions such as functions (e.g., return type) and `trait` implementations. After mapping out all use scenarios that trace back to the original set of import statements, we later can conclude whether a package reuses code from a dependency. The procedures for direct and transitive dependencies are slightly different. For direct dependencies, we investigate the entire package for any sign of reuse. For transitive dependencies, we inspect the context of how a package reuses its direct dependency, and whether the specific reuse of the direct dependency leads to reuse of the transitive dependency. Given the following example: Package `Foo` depends on `Bar`, and `Bar` depends on `Baz`. `Foo` also reuses `Bar`, and `Bar` also reuses `Baz`. A function `bar()` in `Bar` calls `baz()` in `Baz` and `foo()` in `Bar` does not rely on external code. If `Foo` only calls `foo()`, then `Foo` only reuses `Bar` and not `Baz` despite `Bar` reusing `Baz`. If `Foo` would call `bar()`, then there is an indirect reuse of the transitive dependency `Baz`. A step-by-step review protocol is available in the replication package.

Manual Analysis Table 7 tabulates the reasons for misclassification split by network and number of use cases. Overall, the metadata-based network over-approximates the dependency usage in 80% of the analyzed cases. Among direct dependencies where the metadata-based networks over-approximate, we identify seven instances where a package did not import any item from the dependency relationship under analysis. Moreover, metadata-based networks cannot distinguish dependency usage in non-runtime or conditionally compiled

Table 7: Manual inspection and classification of 34 dependency relationships between PRÄZI and the CRATES.IO index network.

Categorization	#Samples
i) Over-approximation in metadata-based networks	**27**
... no import statements	3
... import statement and no usage	4
... resides in a `#[cfg(...)] block`	1
... derive macro libraries	2
... test dependency	1
... non-reachable transitive dependency	16
ii) Under-approximation in PRÄZI	**7**
... importing a constant	1
... importing data type and usage	1
... importing data type in definitions	4
... handling C-function call	1
Σ	**34**

sections of the source code. We found two cases; one case where a developer uses a runtime dependency solely in test code and one conditional compilation case where a dependency code runs only on Windows environments.

While CARGO has labels for build, test, optional, and platform-specific dependencies in the manifest file, *derive macro* dependencies are not distinguishable from runtime dependencies. A *derive macro* library performs code-generation at compile time. However, such libraries do not provide runtime functionality and are closer to the role of being a build dependency. We identify two such libraries, `cfg-if` and `thiserror`. Including such dependencies influences the count of runtime dependencies; for example, depending on the widely popular `serde_derive`[27] library would incorrectly add six dependencies to the total count of runtime dependencies. Without no specific metadata label or heuristic, a call-based dependency network avoids including such libraries.

The most prominent case with over-approximation by metadata-based networks are non-reachable transitive dependencies. The context of how a package uses its direct dependencies plays a central role in whether a package indirectly uses its transitive dependencies. As an example, the package `selfish` uses `nom v3.2.1` that then depends on `regex 0.2.11`. `nom` is a parser library and exports a set of regex parsers that uses the `regex` library. Although `selfish` enables the regex feature in `nom`, it does not import any of the regex parsers in `nom`, effectively making the `regex` library unused.

In the four cases where a developer imports type definitions from dependencies for use in function declarations. One such example is the case of importing `c_int` in `libc` for function declarations in `whereami v1.1.1`. Although a call graph does not track data references, we could still mitigate this by tracking the type declarations in argument and return types of functions in the

[27] `https://docs.rs/crate/serde_derive/1.0.106/source/Cargo.lock`

call graph. PRÄZI embeds full type qualifiers including package information in functions belonging to call-based dependency networks (See Section 3.1).

Finally, we identify one instance where the call graph generator could not resolve a call from `subprocess v0.1.0` to the `libc` function `pipe()`. Although there is a `pipe` call without clear identifiers in the call graph, it is not via the `libc` library. Thus, there are possible limitations with handling cross-language calls.

A call-based dependency network is more precise than a metadata-based network. Data-only dependencies could affect its soundness.

6 Discussion

We center our discussion on two key aspects; differences and similarities between using three different networks for network analyses and studying function relationships on a network level.

6.1 Strengths and Weaknesses between Metadata and Call-based Networks

As package repositories do not test whether a package can build or not, developers can by mistake or unknowingly publish broken versions to CRATES.IO. By verifying the build of package releases, the `Docs.rs` network excludes package releases that do not have a successful build record. When comparing the results of the network analyses in Figure 6 with CRATES.IO index network, overall, we find them to have comparable results except in the formative years of CRATES.IO. The diverging results in the initial years show that a large number of releases are not reproducible and consumable, stressing the importance of performing additional validation besides the correctness of packages manifests. Thereby, we urge researchers to minimally validate package manifests with external information such as publically available build and test data for network studies of package repositories.

When comparing the network analysis results in Figure 6, we find notable similarities and differences between metadata-based and call-based networks for CRATES.IO. Except for the formative years of CRATES.IO, the distributions of recent snapshots for direct dependencies, direct dependents, and total dependents are mostly similar between the networks. Thus, a network inferred from CRATES.IO metadata closely approximates the presence of function reuse relationships between packages without needing to construct and verify with call graphs. Recent snapshots of CRATES.IO further indicate that recent package releases are highly likely to be reproducible and compile as well. On the other hand, there are also significant differences between the networks, specifically for transitive dependencies and outliers in dependent distributions. By taking into account that a developer does not make use of all APIs available in a package, we identify a two-fold difference between call-based and

metadata-based networks. These differences also manifest among the most popular dependent packages (i.e., outliers)—despite the networks reporting similar results for the average dependent package.

Based on these similarities and differences, we conduct a manual analysis to understand which network has a more accurate representation of package repositories. Our investigation indicates that call-based dependency networks are more precise than metadata-based networks; the prominent finding is that the number of transitive dependencies a package uses is highly contextual and moderately correlates with the number of declared dependencies. From a statistical viewpoint, we identify a strong correlation between the number of dependencies derived from a metadata-based network and the number of called dependencies. In other words, the more resolved transitive dependencies a package has, the more transitive dependencies it will call. On the other hand, we only observe a moderate correlation between declared (direct) dependencies and called transitive dependencies, indicating that the number of called transitive dependencies potentially varies for the same number of direct dependencies. Based on our studied use cases, we find examples of packages only importing non-core functionality from libraries or specific modules of packages that use individual libraries by themselves. Despite limitations with data-only dependencies, we argue that calculating the number of transitive dependencies should not be generalized to the sum of all resolved dependencies. In line with previous work on the fine-grained analysis of known security vulnerabilities, we also argue both researchers and practitioners interested in understanding how developers or programs use dependencies should account for its context—not the number of compiled dependencies.

As a summary, we make the following recommendations based on the trade-offs and costs for constructing a call-based dependency network:

- **Direct dependencies:** Given the relative proximity of results between a metadata-based and call-based network, a metadata-based network is sufficient for use cases involving direct dependencies if precision is not crucial. The cost of building a call-based dependency network would be overly expensive.
- **Transitive dependencies:** Where transitive dependencies are central in any analysis, we recommend call-based dependency networks over metadata-based networks.
- **Data-dependencies:** Where data references are crucial to track or studying data-centric packages in Crates.io, we recommend metadata-based dependency networks or use additional (cheap) static analysis to identify data dependencies. Although metadata-based networks are imprecise, they will not miss such relationships.

6.2 Transitive API Usage

For studying the evolution, impact, and the decision-making of deprecation (Robbes et al. 2012; Sawant et al. 2018b) and refactorings (Kula et al.

2018b) of library APIs, datasets such as `fine-GRAPE` (Sawant and Bacchelli 2017) provide valuable insights into how a large number of clients in the wild make use of a few popular libraries. These datasets extract API usage by mining direct invocation of library APIs (i.e., a client calling a public API function). By analyzing the use of APIs in transitive dependencies of clients (i.e., indirect API use) in addition to direct dependencies, we find that there are more calls to transitive dependencies than direct dependencies in recent years. Thus, the transitive relationship where either an intermediate client or library relays a call between a client and a library could potentially present new confounding variables and implications to the evolution and decision-making of APIs. Although developers do not have control of transitive package dependencies, they have the same execution rights and follow the same laws of software evolution (Lehman 1980) as direct dependencies. Thus, API decisions in transitive dependencies can equally impact clients as direct dependencies.

As package managers allow the same dependency (albeit different versions of them) to co-exist in a client, our results in **RQ2.4** show growing signs that more and more copies of the same function identifier from multiple versions exist in a client. In cases where such a function is dependent on the environment (e.g., a specific implementation of an OpenSSL library), there is a potential risk for introducing unexpected incompatibilities. Such problems that arise from the use of transitive dependencies can directly influence the decision-making of APIs. For example, a user in PR #20 of IDnow SDK,[28] an identity verification framework, is persuading the maintainers to drop dependence on Sentry, an application monitoring platform, due to the user having problems with Sentry as several versions of that dependency exist in its application.

Given the increasing growth of indirect API calls and a slight increase of multiple copies of the same function identifier appearing in clients, we call for researchers to also account for the dynamics of dependency management—particularly transitive dependencies—when studying the evolution and decision-making around APIs.

7 Threats to Validity

In this section, we discuss limitations and threats that can affect the validity of our study and show how we mitigated them.

7.1 Internal validity

For CDNs to closely mirror actual package reuse in Crates.io, we only consider packages specified under the `#[dependencies]` section and optionally-enabled packages as these are consumable in the source code. As packages in `#[dependencies]` are also available in the test portion of packages, developers could potentially specify packages for testing purposes that do not attribute

[28] `https://github.com/idnow/de.idnow.ios.sdk/issues/20`

towards package reuse. We mitigate the risk of inferring test specific calls by restricting the build of packages to compilation without further execution steps such as tests.

The `rust-callgraphs` generator can resolve function invocations that involve static and dynamic dispatch except for function pointer types. Although the documentation[29] states that function pointers have a specific and limited purpose, we acknowledge that we cannot make any claims around the completeness of generated CDNs due to the general absence of ground truth for package repositories. When limiting the scope to the features that the call graph generator supports, the generated CDNs represent an over-approximation of function calls in Crates.io. It is an over-approximation as function targets in dynamic dispatch may never be called by the end-user in practice (i.e., it is inexact). Using additional analysis such as dynamic analysis to remove all unlikely function targets is error-prone and could result in unsound inferences. Thus, we avoid considering both static (i.e., exact) and dynamic (i.e., inexact) function calls as the same. Instead, we view the results of dynamically dispatched calls from the perspective of virtual method tables (i.e., its concrete representation during runtime).

Real-world constraints such as non-updated caches of the repository index, user-defined dependency patches, and deviating semver specifications could influence the actual version resolution of package dependencies. The selection of packages and their versions for creating snapshots has additional implications on the representativeness of Crates.io and its users. To mitigate the risk of making incoherent versions resolutions, we use the exact resolver component implemented in Cargo, ensuring the same treatment of version constraints.

Kikas et al. (2017) report the highest package reach to be up to 30% in 2015 while our Crates.io metadata network report over 60%, nearly twice the number. The difference lies in the selection of packages when creating the networks: Kikas et al. (2017) build a dependency tree for all available versions of a package valid at timestamp t and we build a tree for the single most recent version of a package at a timestamp t. As there is no consensus on best practices for which packages and releases to include in a network, we take a conservative approach that avoids including dormant and unused releases. For example, we argue that it is rare that a user today would declare a dependence on version dating back to 2017 when newer versions from 2019 exist. Kikas et al. (2017) would include such versions.

7.2 External and reliability validity

We acknowledge that the results of network analysis are not generalizable to other package repositories and only explain properties of Crates.io. Due to differences in community values (Bogart et al. 2016) and reuse practices of packages, we expect network analyses to yield different results. However,

[29] `https://doc.rust-lang.org/book/first-edition/trait-objects.html`

based on (Decan et al. 2018b) comparison of seven package repositories, we believe certain repositories, for example, NPM and NUGET may share some similarities with CRATES.IO than with CRAN and CPAN.

The PRÄZI approach to constructing a CDN is general applicability as long as the programming language has a resolver for package dependencies and a call graph generator. However, the soundness of generated CDNs may vary depending on the programming language. For example, CDNs generated for Java are more accurate and practical than CDNs for Python due to limited call graph support. Therefore, evaluating trade-offs in terms of precision and recall plays an important role in whether a study scenario is suitable for CDN analysis.

8 Future work

Our work opens an array of opportunities for future work in data-driven analysis of package repositories, both for researchers and tool builders.

8.1 Enabling data-driven insights into code reuse with network analysis

As functions are not the only form of achieving code reuse, we aim to explore how we can model reuse of interfaces, generics, class hierarchies, and wrapper classes as networks. In a similar spirit to enabling data-driven insights of APIs, language designers can use data-driven models to understand patterns and adoption of certain code reuse practices. As Rust advocates developers to prefer using generics over trait objects and limit the use of unsafe code constructs, language designers can verify such premises with feedback through network- and data-driven analyses of package repositories.

Following Zhang et al. (2020a)'s need-finding study on data-driven API design, we are investigating possibilities to mine program contexts and error-inducing patterns using PRÄZI to extract API usage patterns beyond syntactic features and frequencies. Insights into involved API usage patterns can help library maintainers to make changes echoing improvements that simplify code reuse and strengthening the stability of a package repository.

8.2 Modeling socio-technical risks of package abandonment

Package repositories are successful in attracting developers to release new packages. However, they are less successful in keeping these packages maintained on a long-term perspective. As a result of developers abandoning packages due to shifting priorities, unmaintained packages are increasingly jeopardizing the security and stability of package repositories. Notably, the *event-stream* incident (Baldwin 2018) is emerging as a textbook example of how the abandonment of a package turned itself into a bitcoin stealing apparatus affecting thousands of users. While survival analysis of packages can yield insights into the

stages of abandonment (Valiev et al. 2018), understanding the social-technical motives behind developer abandonment could potentially help develop a risk control model that package repository owners can exercise. As an example, when a package repository recognizes the slowdown of development activities of popular yet central packages, they could explore incentives such as monetary support, developer assistant in resolving long-running bug reports, or discuss possible handover to a network of trustful developers. We are exploring both quantitative and qualitative strategies on how to model and mitigate risks around package abandonment using Präzi.

9 Conclusions

In this work, we devise Präzi, an approach combining manifests and call graphs of packages to infer dependency networks of package repositories at the function granularity. By implementing Präzi for Rust's Crates.io, we showcase the feasibility of compiling and generating call graphs for 70% of all indexed releases. Then, we compare the Crates.io CDN against a conventional metadata-based network and an enhanced corroborated version with compile data in a study to understand their differences and similarities in network analysis common to package repositories and derive new insights of Crates.io. By using function call data, we find that packages do not indirectly call 60% of their transitive dependencies. Packages that have transitive consumers are likely to have three times more calls from indirect users than direct users. When we investigated the trends of function calls, we observed that packages make 6.6 new direct and 12.2 new indirect calls to dependencies every six months. A majority of packages in Crates.io have no or limited reachability; the most reachable function in 2020— `max-level()` in package `log`—reaches 30% of all Crates.io packages. When comparing the three studied networks, we find that metadata-based networks closely approximates the CDN for analysis involving direct package relationships. Analysis of transitive package relationships and top-most dependent packages, on the other hand, yield significantly different results for the studied networks. A manual investigation of 34 cases reveals that a CDN is more precise as it accounts for the context of how packages use each other. Thus, dependency checkers such as, Rust's `cargo-audit` and GitHub's `Dependabot`, can benefit from call graph analysis to generate more precise recommendations for developers on transitive dependencies. Overall, Präzi opens up new doors to precise network analysis of code reuse and APIs of package repositories.

Acknowledgments: The work in this paper was partially funded by NWO grant 628.008.001 (CodeFeedr) and H2020 grant 825328 (FASTEN). Georgios Gousios is the main recipient of both funding grants.

References

Abdalkareem R, Nourry O, Wehaibi S, Mujahid S, Shihab E (2017) Why do developers use trivial packages? an empirical case study on npm. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, pp 385–395

Abdalkareem R, Oda V, Mujahid S, Shihab E (2019) On the impact of using trivial packages: an empirical case study on npm and pypi. Empirical Software Engineering pp 1–37

Albert R, Barabási AL (2002) Statistical mechanics of complex networks. Reviews of modern physics 74(1):47

Ali K, Lhoták O (2012) Application-only call graph construction. In: European Conference on Object-Oriented Programming, Springer, pp 688–712

Alimadadi S, Mesbah A, Pattabiraman K (2015) Hybrid dom-sensitive change impact analysis for javascript. In: 29th European Conference on Object-Oriented Programming (ECOOP 2015), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik

Aparicio J (2019) cargo-call-stack: Static, whole program stack analysis. `https://github.com/japaric/cargo-call-stack`

Baldwin A (2018) Details about the event-stream incident. `https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident`

Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the state of static analysis: A large-scale evaluation in open source software. In: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, IEEE, pp 470–481

Bogart C, Kästner C, Herbsleb J, Thung F (2016) How to break an API: Cost negotiation and community values in three software ecosystems. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, pp 109–120

Brian A, David T, Aaron T (2020) The rust libz blitz. `https://blog.rust-lang.org/2017/05/05/libz-blitz.html`

Chen L, Hassan F, Wang X, Zhang L (2020) Taming behavioral backward incompatibilities via cross-project testing and analysis. In: IEEE/ACM International Conference on Software Engineering

Chinthanet B, Ponta SE, Plate H, Sabetta A, Kula RG, Ishio T, Matsumoto K (2020) Code-based vulnerability detection in node. js applications: How far are we? In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 1199–1203

Cogo FR, Oliva GA, Hassan AE (2019) An empirical study of dependency downgrades in the npm ecosystem. IEEE Transactions on Software Engineering

Decan A, Mens T, Constantinou E (2018a) On the impact of security vulnerabilities in the npm package dependency network. In: International Conference on Mining Software Repositories

Decan A, Mens T, Grosjean P (2018b) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering

Decan A, Mens T, Grosjean P (2019) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering 24(1):381–416

Dietrich J, Pearce D, Stringer J, Tahir A, Blincoe K (2019) Dependency versioning in the wild. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, pp 349–359

Duan R, Bijlani A, Xu M, Kim T, Lee W (2017) Identifying open-source license violation and 1-day security risk at large scale. In: Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security, pp 2169–2185

Dunn J (2017) Pypi python repository hit by typosquatting sneak attack. https://nakedsecurity.sophos.com/2017/09/19/pypi-python-repository-hit-by-typosquatting-sneak-attack/

Emami M, Ghiya R, Hendren LJ (1994) Context-sensitive interprocedural points-to analysis in the presence of function pointers. ACM SIGPLAN Notices 29(6):242–256

Hejderup J (2015) In dependencies we trust: How vulnerable are dependencies in software modules? Master's thesis, Delft University of technology

Hejderup J, van Deursen A, Gousios G (2018) Software ecosystem call graph for dependency management. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ACM, pp 101–104

Hejderup J, Beller M, Triantafyllou K, Gousios G (2021) Präzi: From Package-based to Call-based Dependency Networks. DOI 10.5281/zenodo.4478981, URL https://doi.org/10.5281/zenodo.4478981

Hopkins WG (1997) A new view of statistics. Will G. Hopkins

Katz Y (2016) Cargo: predictable dependency management. https://blog.rust-lang.org/2016/05/05/cargo-pillars.html

Kikas R, Gousios G, Dumas M, Pfahl D (2017) Structure and evolution of package dependency networks. In: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, pp 102–112

Kula RG, De Roover C, German DM, Ishio T, Inoue K (2018a) A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 288–299

Kula RG, Ouni A, German DM, Inoue K (2018b) An empirical study on the impact of refactoring activities on evolving client-used apis. Information and Software Technology 93:186–199

Lehman MM (1980) Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68(9):1060–1076

Livshits B, Sridharan M, Smaragdakis Y, Lhoták O, Amaral JN, Chang BYE, Guyer SZ, Khedker UP, Møller A, Vardoulakis D (2015) In defense of soundiness: a manifesto. Communications of the ACM 58(2):44–46

Martins P, Achar R, Lopes CV (2018) 50k-c: A dataset of compilable, and compiled, java projects. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), IEEE, pp 1–5

Matsakis N (2016) Introducing mir. https://blog.rust-lang.org/2016/04/19/MIR.html

Mezzetti G, Møller A, Torp MT (2018) Type regression testing to detect breaking changes in node. js libraries. In: 32nd European Conference on Object-Oriented Programming (ECOOP 2018), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik

Mirhosseini S, Parnin C (2017) Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, pp 84–94

Mones E, Vicsek L, Vicsek T (2012) Hierarchy measure for complex networks. PloS one 7(3):e33799

Mujahid S, Abdalkareem R, Shihab E, McIntosh S (2020) Using others' tests to identify breaking updates. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp 466–476

Nguyen HA, Nguyen TN, Dig D, Nguyen S, Tran H, Hilton M (2019) Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 819–830

Ponta SE, Plate H, Sabetta A (2018) Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 449–460

Preston-Werner T (2013) Semantic versioning. https://semver.org/

Raemaekers S, van Deursen A, Visser J (2017) Semantic versioning and impact of breaking changes in the maven repository. Journal of Systems and Software 129:140–158

Robbes R, Lungu M, Röthlisberger D (2012) How do developers react to api deprecation? the case of a smalltalk ecosystem. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp 1–11

Ryder BG (1979) Constructing the call graph of a program. IEEE Transactions on Software Engineering (3):216–226

Salis V, Sotiropoulos T, Louridas P, Spinellis D, Mitropoulos D (2021) Pycg: Practical call graph generation in python. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, pp 1646–1657

Sawant AA, Bacchelli A (2017) fine-grape: fine-grained api usage extractor–an approach and dataset to investigate api usage. Empirical Software Engineering 22(3):1348–1371

Sawant AA, Aniche M, van Deursen A, Bacchelli A (2018a) Understanding developers' needs on deprecation as a language feature. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, pp 561–571

Sawant AA, Aniche M, van Deursen A, Bacchelli A (2018b) Understanding developers' needs on deprecation as a language feature. In: Proceedings of the 40th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '18, pp 561–571

Sawant AA, Robbes R, Bacchelli A (2018c) On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. Empirical Software Engineering 23(4):2158–2197

Schlueter I (2013) Unix philosophy and node.js. https://blog.izs.me/2013/04/unix-philosophy-and-nodejs

Schlueter I (2017) The npm blog — kik, left-pad, and npm. http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm

Shapiro M, Horwitz S (1997) Fast and accurate flow-insensitive points-to analysis. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp 1–14

Shivers O (1991) Control-flow analysis of higher-order languages. PhD thesis, PhD thesis, Carnegie Mellon University

Steensgaard B (1996) Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp 32–41

Sulír M, Porubän J (2016) A quantitative study of Java software buildability. In: Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, ACM, pp 17–25

Sundaresan V, Hendren L, Razafimahefa C, Vallée-Rai R, Lam P, Gagnon E, Godin C (2000) Practical virtual method call resolution for Java, vol 35. ACM

Tip F, Palsberg J (2000) Scalable propagation-based call graph construction algorithms. In: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp 281–293

Triantafyllou K (2019) A benchmark for rust call-graph generators. https://users.rust-lang.org/t/a-benchmark-for-rust-call-graph-generators/34494

Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2017) There and back again: Can you compile that snapshot? Journal of Software: Evolution and Process 29(4)

Valiev M, Vasilescu B, Herbsleb J (2018) Ecosystem-level determinants of sustained activity in open-source projects: a case study of the pypi ecosystem. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, pp 644–655

Xavier L, Brito A, Hora A, Valente MT (2017) Historical and impact analysis of api breaking changes: A large-scale study. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 138–147

Zapata RE, Kula RG, Chinthanet B, Ishio T, Matsumoto K, Ihara A (2018) Towards smoother library migrations: A look at vulnerable dependency mi-

grations at function level for npm javascript packages. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 559–563
Zerouali A, Constantinou E, Mens T, Robles G, González-Barahona J (2018) An empirical analysis of technical lag in npm package dependencies. In: International Conference on Software Reuse, Springer, pp 95–110
Zhang T, Hartmann B, Kim M, Glassman EL (2020a) Enabling data-driven api design with community usage data: A need-finding study. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, pp 1–13
Zhang T, Hartmann B, Kim M, Glassman EL (2020b) Enabling data-driven api design with community usage data: A need-finding study. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, pp 1–13
Zhong H, Thummalapenta S, Xie T, Zhang L, Wang Q (2010) Mining api mapping for language migration. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pp 195–204
Zimmermann M, Staicu CA, Tenny C, Pradel M (2019) Small world with high risks: A study of security threats in the npm ecosystem. In: 28th USENIX Security Symposium (USENIX Security 19), pp 995–1010

A Selecting a time window for dependency resolution

Instead of using a single fixed version at all times, version constraints allow developers to use a time-constrained version that updates itself at new compilations. Nearly all dependencies in CRATES.IO specify a dynamic version constraint—only 2.92% of all dependency specifications in CRATES.IO use a single (immutable) version (Dietrich et al. 2019). Before studying the evolution and structure of CRATES.IO, we first decide the number of time points and a time window between each time point. Although popular studies such as (Kikas et al. 2017) and (Decan et al. 2019) use a time window of one year to study structural changes, we, instead, determine a time window based on the frequency of structural changes in CRATES.IO.

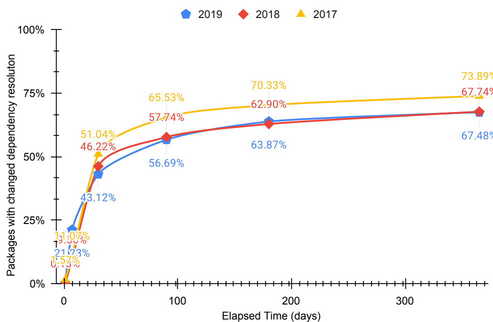


Fig. 11: Retroactive resolution of dependencies over a time period of one year in 2017, 2018, and 2019

After resolving the dependency tree of a set of packages in Crates.io at a time t, we then re-resolve it using six different time points (i.e., one day, one week, one month, three months, six months, and one year) to find a time window where a large fraction of them have a changed dependency tree. We perform this using a set of packages having at least one non-optional dependency at the beginning of 2017 (5,252 package releases), 2018 (9,716 package releases), and 2019 (16,098 package releases).

Figure 11 shows the fraction of packages with a changed dependency tree (i.e., a tree with at least one different version) over time. We observe a logarithmic trendline for each year group; a high increase of packages with changed dependency between time points before three months, and then it levels out. After one month, we already find that 40% of all packages have a changed dependency tree due to new releases of 148 packages in 2017, 190 packages in 2018, and 240 packages in 2019. In all year groups, we find that the dependence on `libc` triggers a new version resolution for most packages, followed by other popular packages such as `quote`, `serde`, and `syn`. A manual inspection of the release log for `libc`[30] and `serde`[31], suggests a frequency of at least two releases per month.

Finally, we also observe that 26% of all packages in 2017 have an identical dependency tree after one year. Among those unchanged packages, nearly all of them (2017: 83%, 2018: 93%, 2019: 90%) are outdated packages. With outdated, we mean that no recent releases for those packages in more than one year. Although packages may be outdated, they still could use flexible version constraints. In roughly one-third (2017: 31%, 2018: 34% 2019: 40%) of all dependency constraints, the dependencies are outdated packages (i.e., there are no recent releases). In the remaining cases (i.e., where more recent versions exist), the version constraints cover old releases (e.g., depending on `serde` 2.x when 4.x exists), and less than 1% are fixed versions. For example, `xml-attributes-derive::0.1.0`[32] depends on older versions of `syn`, `quote`, and `proc-macro2`, and `trie-root::0.11.0`[33] depends on an old version of `hash-db`.

Given these observations, we select a time window of one month and thus perform dependency resolution every month per year.

> 40% of all Crates.io packages have at least one dependency resolving to a new version after 30 days.

[30] https://crates.io/crates/libc/versions
[31] https://crates.io/crates/serde/versions
[32] https://docs.rs/crate/xml-attributes-derive/0.1.0/source/Cargo.toml
[33] https://docs.rs/crate/trie-root/0.11.0/source/Cargo.toml