Delft University of Technology

Präzi: From Package-based to Precise Call-based Dependency Network Analyses

Hejderup, Joseph; Beller, Moritz; Gousios, Georgios

Publication date
2018

Citation (APA)
Hejderup, J., Beller, M., & Gousios, G. (2018). Präzi: From Package-based to Precise Call-based Dependency Network Analyses.

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

PRÄZI: From Package-based to Precise Call-based Dependency Network Analyses

Joseph Hejderup
Delft University of Technology
The Netherlands
j.i.hejderup@tudelft.nl

Moritz Beller
Delft University of Technology
The Netherlands
m.m.beller@tudelft.nl

Georgios Gousios
Delft University of Technology
The Netherlands
g.gousios@tudelft.nl

***Abstract*—Package-based dependency networks model which software packages depend on which other packages. Researchers and practitioners have used them to achieve a great number of analyses, including automatically warning for security vulnerability, ecosystem health and license compliance issues. However, traditional package-based dependency networks are in-precise, severely limiting their use in practice. In this paper, we present a novel and general approach named PRÄZI to construct call-based dependency networks beyond a single program, its initial prototypical implementation RUSTPRÄZI for the Rust library system CRATES.IO, an evaluation of its soundness and precision, and two sample applications with it. Our case study on security vulnerabilities showed that RUSTPRÄZI is three times more accurate than the current state of the art, package-based analyses. PRÄZI also opens the door to new applications, e.g., an analysis on the prolonged use of deprecated methods. It showed that 48% of the studied dependent packages break when a deprecated function gets removed. Several perils endanger a practical implementation of PRÄZI, affecting both its soundness and precision. We discuss and quantify them along the RUSTPRÄZI example, equipping researchers and practitioners with guidelines on how to implement PRÄZI. Finally, we also show that there is no principal objection to make PRÄZI fully sound and precise.**

***Index Terms*—Software tools, Software quality, Software libraries, Static analysis, Call-based dependency network analysis**

I. INTRODUCTION

In today's software development, most programs comprise a growing list of other software they depend on [1]. Online Package Repositories (OPRs, see Table I) of modern programming languages such as Java's MAVEN CENTRAL allow the efficient (re-)use and combination of already existing functionality in one's own program. While reuse is a core Software Engineering principle, promising higher development speed and quality [2], [3], uninhibited reuse from OPRs bears risks [4]. These became painstakingly obvious to the JavaScript community in the "leftpad incident", when the removal of one package lead to the breaking of hundreds of thousands of other programs depending on it [5], most transitively through a chain of other dependencies. As a result, researchers have analyzed the dependency graphs found in OPRs from a variety of different viewpoints [4], [6]–[9], including security vulnerabilities and ecosystem health. On the practical side, companies are using dependency graphs for an array of applications: GITHUB and TIDELIFT warn project owners when they (implicitly) depend on a known vulnerable library [10], [11]; Google's operation Rosehub supplied pull requests to over 2,600 GITHUB projects which imported a vulnerable version of the APACHE COLLECTIONS library, including the popular SPRING framework, thus fixing the many more transitive projects depending on it [12]; BLACKDUCK performs license compliance checking to avoid importing two dependencies with conflicting licenses [13].

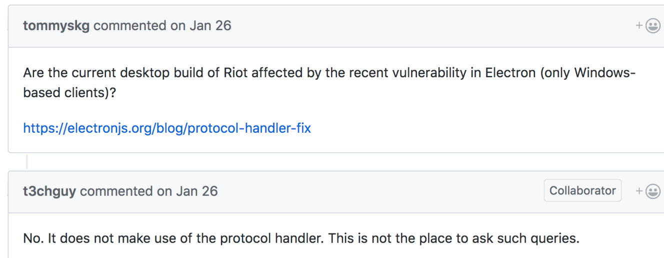


Fig. 1: False warning for `vector-im/riot-web`, #6044.

However, these state-of-the-art analyses operating on package-based dependency networks (PDN) share one shortcoming: they use the readily accessibly dependency specification in a project's metadata (such as MAVEN's `pom.xml`) and might be imprecise, because the actual dependency use happens at the source code level. Hence, existing PDNs are sound, but an over-approximation of the real dependency use. For example, a project might have redundant dependencies to packages whose functionality is not used anymore, creating false positive warnings. False positives, are a dominating factor for the slow adoption of static analysis tools in practice [14], with developers having to sift through several false positives just to find one relevant warning [15], [16]. The faulty security warning raised by GITHUB's package-based dependency checker in Figure 1 exemplifies the confusion false positives can create.

In this paper, we present an approach to build dependency analyses not at the package but at the function call level, called PRÄZI. Call graphs represent the inter-procedural control flow of source code and thus naturally lend themselves to this objective. With PRÄZI, we build the call graph of each package and its transitive dependencies and merge them together, resulting in a call-based dependency network (CDN). With PRÄZI, we can improve the state-of-the-art PDN analyses with more

TABLE I: Acronyms used in this paper

Acronym	Meaning	Definition
CDN	Call-based Dependency Network	Section III
IR	LLVM's Intermediate Bytecode Representation	Section V-B
OPR	Online Package Repository	Section I
PDN	Package-based Dependency Network	Section III
UFI	Unique Function Identifier	Section IV-C

precise CDN analyses. Our technique allows the application of the above analyses on a finer level. The main advantage of PRÄZI is that it i) is more precise, avoiding spurious warnings such as the one in Figure 1 and ii) also opens the door to new applications on an entire OPR from change impact analyses ("Which clients break if I as a library maintainer remove this deprecated method?") to network health ("What are the most important methods and are they tested well?").

In the remainder of this paper, we first describe the generic PRÄZI approach to produce a CDN comprising the merger of several packages' call graphs. PRÄZI is simple to explain but difficult to implement at scale. We thus present a prototypical implementation and an evaluation of PRÄZI on Rust and CRATES.IO. We describe a list of perils of how practical implementation choices can threaten the soundness and precision of PRÄZI, and quantify the effects of the perils in our Rust implementation RUSTPRÄZI during its construction process. While our prototypical implementation is only tailored to statically dispatched functions, our evaluation of the CDN in two case studies on the propagation of security vulnerabilities and deprecated functions shows that it is still useful in practice. In comparison to traditional coarse-grained package network based analyses, PRÄZI eliminates false positives. In a case study with 482 warnings, RUSTPRÄZI reports an accuracy which is three times higher as compared to a traditional PDN, despite reporting a lower recall score than a PDN due to non-static function dispatch.

II. BACKGROUND

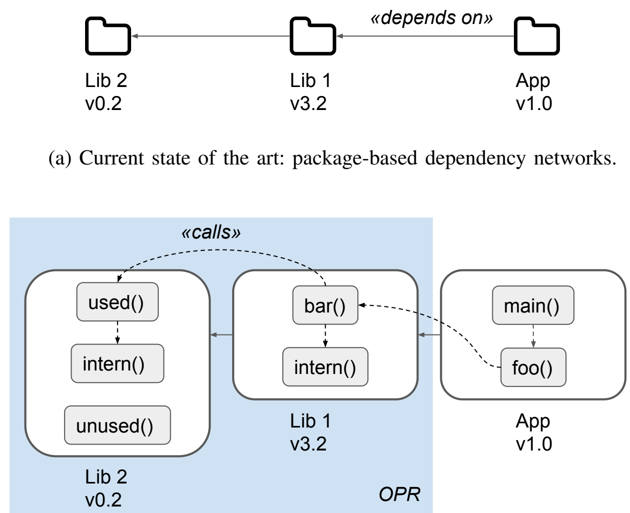
In this section, we give background information over call graphs and the Rust programming language.

A. Call Graphs

A call graph is a reduced control flow graph [17], which only represents function calls and their relationship within a single program, be it an executable or a library. We can produce call graphs from static program artifacts, but there also exist techniques to infer call graphs from dynamic program execution traces [18].

Static call graphs are ideally suited to represent static function calls, but we need more advanced techniques such as aliasing to precisely deal with dynamic invocations, frequently occurring in languages such as Java [19]. Dynamic program analysis can complement static call graph generation, but it requires running the program with all potential combinations of inputs to be fully sound.

To circumvent the problem of aliasing, many call graph generators sacrifice precision for soundness, blowing up the



(a) Current state of the art: package-based dependency networks.

(b) Our proposal: call-based dependency networks.

Fig. 2: Different granularities of dependency networks.

call graph with lots of possible, but unlikely calls. In contrast, Feldthaus et al. have advocated the use of unsound, but highly useful call graphs for JavaScript programs in practice [20]. Reif et al. have invented and shown the practical usability of two unsound algorithms to create call graphs specifically for libraries in Java [21]. While our general PRÄZI approach is unaffected by these issues, we made a similar trade-off for our RUSTPRÄZI, which uses LLVM's call graph generator. LLVM also sacrifices soundness for precision in edge cases.

B. Rust

Rust is a relatively new (first stable release 1.0 in 2015 [22]) systems programming language that aims to combine the speed of C with the memory safety guarantees of a garbage-collected language such as Java. Among languages that we could select for analysis, Rust is unique because its package management system (CARGO) was designed from the ground-up to be part of the language environment. CARGO not only manages dependencies, but prescribes a compilation process and a standardized repository layout that facilitate the creation of automated, large-scale analyses such as ours. Every Rust package contains a file called `Cargo.toml`, which defines the packages' dependencies. Moreover, with CRATES.IO, there is one central place where all Rust packages (so-called "crates" in Rust terminology) live. CRATES.IO currently hosts 17,624 packages (August 16th 2018).

III. CALL-BASED DEPENDENCY NETWORKS

We distinguish two kinds of dependency networks, shown in Figure 2: *package-based dependency networks* (PDNs) similar to the ones that a dependency resolution tool (e.g., CARGO or MAVEN) would build in Figure 2a, and fine-grained *call-based dependency networks* (CDNs) we advocate in this paper in Figure 2b.

Figure 2 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 package name and version. An edge connecting two nodes that one package imports the other, for example from `App 1.0` to `Lib1 3.2`.

Figure 2b consists of three individual call graphs for `App`, `Lib1`, `Lib2`. Each such traditional call graph approximates 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 2b. However, the function identifiers bear no version, nor do they have globally unique identifiers.

To produce a CDN, we merge the two graph representations in a call-based dependency network:

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 $\langle \texttt{id}, \texttt{ver} \rangle$, 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 $\langle \texttt{v}_1, \texttt{v}_2 \rangle \in E$ represents a function call from $\texttt{v}_1$ to $\texttt{v}_2$.

Applying the above definitions, the function `used()` Figure 2b is a node with the fully qualified identifier $\langle \texttt{Lib2::used, 0.2} \rangle \in V$. The dependency between `App` and `Lib1` would be represented as $\langle \langle \texttt{App::foo, 0.1} \rangle, \langle \texttt{Lib1::bar, 3.2} \rangle \rangle \in E$.

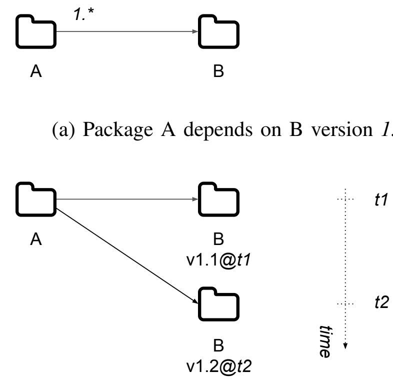
CDNs offer a white-box view of the more coarse-grained packaged-based dependency networks. In particular, we can see that `unused()` is never called. If only `unused()` was affected by a vulnerability, we can deduce from Figure 2b that we should not issue a security warning for `App`, since it does not use the affected functionality. In contrast to the CDNs proposed in this paper, the PDN in Figure 2 by its nature can not provide such a fine-grained resolution level.

IV. Building CDNs

In this section, we devise a generic technique, PRÄZI, to systematically analyze a set of packages residing in an OPR to construct a CDN. We argue that PRÄZI is generic enough to 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. As the overview in Figure 4 shows, PRÄZI first has to resolve and retrieve all packages (and their dependencies) to be analyzed. It then has to generate call graphs for them, unify these to avoid name clashes and link them together in one giant call graph, the CDN. In the following sections we present each step of PRÄZI in detail, along with a set of perils that may affect the soundness and completeness of the produced CDN.

A. Resolving Dependencies and Retrieving Packages

PRÄZI starts with a pre-defined set of packages for which a CDN should be built, the seed set. Packages in the seed set should have dependencies that can be resolved within the context of an OPR. The first step for PRÄZI is to resolve all dependencies in the seed set in a recursive fashion, until it has calculated the full transitive closure of the packages and their associated versions it needs to retrieve.



(a) Package A depends on B version *1.**.

(b) Full dependency resolution tree including time.

Fig. 3: Retroactive dependency resolution.

Dependency resolution is complicated by the need to build a full tree of package versions not only for the present, but also for the past, and possibly future. Almost all current package managers allow developers to specify *ranges* of dependency versions, often in the semantic versioning format. For example, any dependency version with a leading *1.* fulfills the version range $1.*$ (e.g. *1.0*, *1.8*, or *1.20.2*). The package manager will generally choose the latest available version *at the time of its invocation.* This complicates the retroactive resolution of dependency versions.

Suppose that package A depends on version range $1.*$ of package B, as shown in Figure 3a. Package B releases versions 1.1 at t_1 and 1.2 at t_2 ($t_1 < t_2$) (Figure 3b). If the dependency resolution happens at t, where $t_1 < t < t_2$, then the dependency manager will select version 1.1. However, if dependency resolution happens at $t > t_2$, it will select version 1.2, even though A's dependency specification ($1.*$) remained unchanged. Removal or black-listing of packages, a practice supported by many OPRs, further complicate retroactive resolution.

To deal with this issue, PRÄZI expects the nodes in a PDN to be timestamped. It resolves dependency constraints by linking dependents to *all* versions of their dependencies that would satisfy the constraints (in the example above, A would thus depend *both* on B 1.1 and B 1.2). This creates an over-approximation of the actual PDN; to resolve the exact dependencies for package C version v_1 released at timestamp t_{v_1}, PRÄZI removes all nodes, and the corresponding edges, whose release timestamp t_n is $t_n > t_{v_1}$; it then performs a breadth-first search of the dependency graph from C_{v_1}. If it finds a node (e.g., A) that links to multiple versions of another node (e.g., B), it must apply an equivalent selection strategy to that of the actual package manager. Usually, this strategy is to select the latest of those versions.

After resolving dependencies, PRÄZI must download the releases of the identified package versions. What this step will retrieve depends on the call graph generator requirements for the target programming language. For example, in languages where the call graph generator can work on intermediate formats (e.g., Java), binary packages may be sufficient; in most other languages though, PRÄZI needs to retrieve the source code of the package.

Peril 1. For the PRÄZI method to be complete, all packages specified in the seed set must be retrievable at analysis time along with their metadata. Not all OPRs can guarantee this: for example, until the leftpad incident, NPM allowed developers to remove packages.

Peril 2. To generate the PDN, PRÄZI needs the dependency metadata descriptor for each package. Several package managers, including NPM and CARGO, do not check whether the dependency metadata are well-formed or do not reference resources local to the developer's workstation when new package versions are uploaded. This may lead to missing packages in the PDN, and subsequently, our CDN.

Peril 3. Precise runtime dependency resolution complicates accurate construction of dependency sets, and may affect the replicability of the PRÄZI process. The version of depended-on packages may have significantly changed externally without any changes to the importing package. Thus, two consecutive builds of the CDN may be different.

B. Generating Call Graphs

After all package dependencies have been resolved and fetched locally, the PRÄZI technique generates a call graph for each package version. Depending on the call graph generator implementation and the programming environment, the package may need to be built. PRÄZI treats the call graph generator as a pluggable component. Any implementation that adheres to the following requirements is suitable for PRÄZI: i) Function types and their arguments must be fully resolved. For example, a Rust call graph generator will resolve a call to function `parse(input: &str)` in struct `Url` that resides in package `url` as `url::Url::parse(input: &str)`, where `str` is the Rust standard string type. ii) The graph output format is an edge list of function name pairs.

Peril 4. Generating sound call graphs statically is a challenging problem [23]: Certain language features, such as dynamic dispatch and reflection do not allow all function calls that can occur at runtime to be visible with basic static analysis. Macros and templates generate (or exclude) code at compile time, which limits a static call graph generator's ability to accurately include all possible calls. Advanced programming techniques, such as profile-based optimizations, runtime code generation or meta-programming, hinder the call graph generator's ability to track function applications statically. Since PRÄZI inherits these limitations, it is important to know the intended use case upfront. For example, using an otherwise unsound call graph generator in a security-critical project that adheres to only using static dispatch, is perfectly sound. The same implementation might also be fine for projects with lesser security requirements or to check for performance warnings, in case a missed warning is acceptable or where the cost of flooding developers with warnings is perceived as higher than a rare missed warning. It will not work for projects which require that all security warnings be found, even at the expense of a large number of false positives. In such cases, a different call graph generator should be chosen.

Peril 5. In compiled programming languages, call graph generators work as part of the compiler or operate directly on the binary output. This is because the call graph generator needs the compiler to perform macro expansion and type resolution prior to resolving function types. In the context of PRÄZI, this means that all packages under analysis must be built, provided that there is a build process for the package. Retroactively building software is known to be difficult [24]. Dependencies on system libraries, custom build environments, compiler flags, and flaky tests affect the automated buildability of packages, a-priori limiting the number of package versions one can analyze.

C. Generating Unique Function Identifiers (UFIs)

To merge the call graphs for each individual package version into a single CDN, PRÄZI needs to ensure that the contained function signatures are globally unique. Complications can arise in case one program imports multiple versions of the same package, packages include similar function names (e.g. `log(x: &str)` in languages with flat namespaces, or duplicate function names when processing function calls across OPRs. To solve these issues, PRÄZI prepends all calls, including function names and the types in their arguments, with three attributes: i) OPR name, ii) package name and iii) package version. For example, the UFI for the above function `parse(input: &str)`, which can be found in the CRATES.IO OPR in version `1.6.1` of the `url` crate, is `io::crates::url::1.6.1::Url::parse(input: &str)`.

D. Unifying Call Graphs

The last step of the PRÄZI technique involves merging together all individual package-level CDNs into a single CDN. The process consists of aggregating all individual package call graphs and filtering out duplicate nodes. The end result is the CDN corresponding to the input seed set.

V. IMPLEMENTING PRÄZI FOR RUST

We chose Rust to showcase the practical feasibility and scalability of PRÄZI for a number of reasons: As a new language, Rust's OPR, CRATES.IO, is relatively contained compared to NPM's 650,000 packages (see Section II-B). LLVM, Rust's standard compiler, can output call graphs as a side-artifact of compilation. Thus, we can have high confidence in their correctness.

These facts set Rust apart from almost all legacy languages, such as C and C++, where building and dependency management is usually done at the operating system level. Moreover, it is hard to define what exactly the OPR of, e.g., Java is,

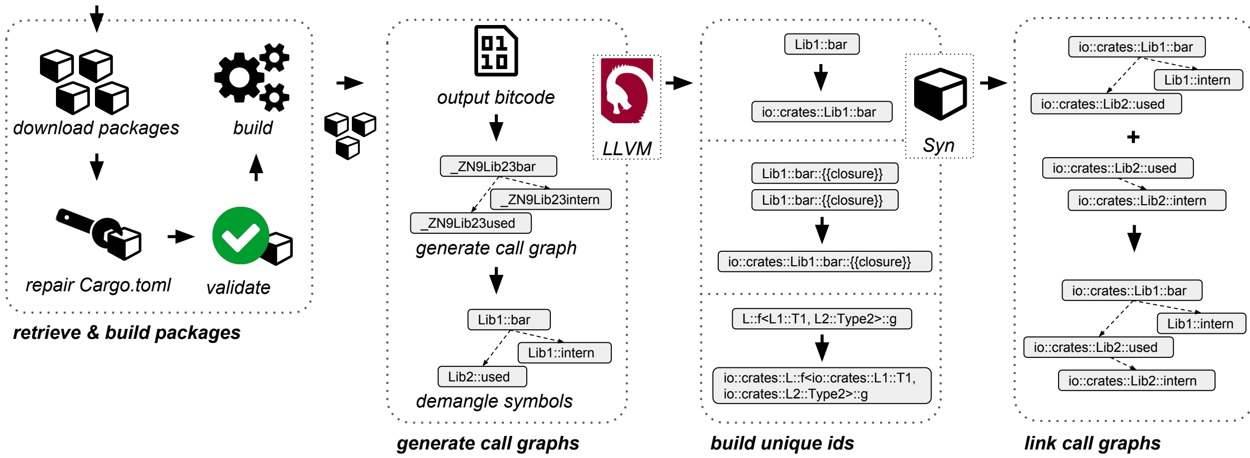


Fig. 4: Our approach to generate a CDN for the CRATES.IO OPR.

as MAVEN CENTRAL also contains build artifacts from Scala, Groovy, Closure, JRuby and Kotlin. Dynamic dispatch, which is hard to capture via static call graphs, is common practice in Java and C. Moreover, with CARGO, Rust has a unified way to build projects, unlike Java, where projects can use ANT, MAVEN, or GRADLE. Finally, in contrast to highly dynamic languages such as JavaScript, Rust as a strongly typed language lends itself to static analysis, and thus, to building accurate call graphs. In fact, the Rust language documentation itself advocates static over dynamic dispatch [25].

A. Resolving Dependencies and Retrieving Packages

CRATES.IO hosts an official up-to-date index of its packages in a GITHUB repository [26]. We clone a snapshot of the index at revision `b76c5ac` (16th February 2018) and populate our seed set with all its entries. This set constitutes of $79,724$ releases (i.e., package versions) from $13,991$ unique packages. The complete seed set makes the step of resolving dependencies to fetch missing packages redundant. To download a specific package from CRATES.IO, we use its dedicated API [27]. In total, we could download and uncompress $79,701$ package versions from our seed set.

Peril 1. We are missing 23 package versions due to unauthorized access and 8 package versions due to malformed tar archive headers.

Peril 2. Because CRATES.IO does not validate build manifests (i.e., the `Cargo.toml` file) in released packages in CRATES.IO, our downloaded set may contain invalid build manifests. A package with an invalid build manifest is not compilable, and hence cannot be used as a dependency in other projects. Using CARGO's manifest validator tool, we could identify 477 package versions with invalid manifests. We further exclude "packages" that are client applications, excluding $6,277$ non-library package releases. In total, $72,947$ package versions (i.e., 92% of our pre-defined seed set) can be used to build the Rust CDN.

Peril 3. In RUSTPRÄZI, we have simplified the retroactive dependency problem and created a network that is only valid for one point in time, at revision `b76c5ac`. We now want to quantify how different the PDN looks when we build it a day, a week, a month, 3 months and 6 months later. We construct the new PDNs by keeping the source code from `b76c5ac`, and only updating the CRATES.IO index to the new timestamps. This ensures that the code will not change, but will fetch new dependencies, if available. From Table II, we can observe that CRATES.IO is very volatile: even within one month, a quarter of the dependencies have resolved to a different version. Thus, if having an accurate complete history is a concern for a practical implementation of PRÄZI, it seems vital to follow the general approach outlined in Section IV-A.

TABLE II: Retroactive dependency changes.

Revision	Time point	#(Changed) Packages
`b76c5ac`	Feb 16'18	297,757
`6e9b751`	1 day (Feb 17'18)	429 (0.1%)
`76a24f9`	1 week (Feb 23'18)	8,484 (2.9%)
`eb7b311`	1 month (Mar 16'18)	55,651 (23.0%)
`a4bc79d`	3 months (May 11'18)	82,960 (38.0%)
`75f1ff7`	6 months (Aug 3'18)	97,509 (48.7%)

B. Generating Call Graphs

To generate call graphs, we use the LLVM call graph generator, version `4.0.0`, which works by analyzing a program's LLVM Intermediate Bytecode Representation (IR).

Peril 4. Not being Rust-specific, LLVM may miss Rust-specific calling conventions, leading to a potentially incomplete Rust CDN. To evaluate its shortcomings, we consider all possible ways [25] to define or call a function in Rust (Table III). We then construct examples that exercise a specific function call or definition, and generate the call graph representing these cases. After inspecting the generated call graph, we document the support of each feature in Table III. Overall, we can identify that the LLVM call graph generator is not able to infer non-static dispatch calls or macro invocations.

TABLE III: LLVM call graphs and Rust call mechanisms.

Call Mechanism	Support
Standard function definition [28]	✓
Generic function definition [28]	✗
External function definition (e.g., FFI) [28]	✓
Standard method call (e.g., `Foo::m();`) [29]	✓
Standard method call with receiver (e.g., `Foo.m();`) [30]	✓
Statically dispatched method call (+/- receiver) [31]	✓
Dynamically dispatched method call (+/- receiver) [31]	✗
Macros (e.g., `print!("hello");`) [32]	✗

TABLE IV: Round-by-round build statistics.

Build Round	#Releases	#Packages	Time (hrs)
CRATES.IO	72,947	12,307	—
1. Rustc stable	40,366 (55%)	9,376 (76%)	33.8
2. Rustc nightly	+4,972 (+7%)	+976 (+8%)	+13.5
3. Cargo.toml fixes	+2,644 (+4%)	+244 (+2%)	+5.8
4. Native dependencies	+1,862 (+3%)	+235 (+2%)	+16.5
Σ	49,844 (69%)	10,831 (88%)	69.6

Furthermore, it is only able to infer generic function definitions if instantiations of it exist.

To generate the LLVM IR of the Rust packages, we need to compile our set of $72,947$ package versions. Compiling such a large set of packages is a challenging task because many environmental factors influence it. It is also an important step because compile failures affect the completeness of the Rust CDN.

We perform the build step in several compilation rounds to achieve maximum completeness. We first use a stable version of the compiler, and then iteratively analyze compilation logs to tackle the common failure reasons. The compilation itself ran for almost three days in parallel using a build server with an Intel Xeon E5-2690 v4 CPU with 14 hyper-threaded cores clocked at 2.6GHz, 128GB RAM and seven 2TB SSDs formatted with ZFS in RAIDZ-1 (RAID-5 equivalent) mode, on Ubuntu 16.04.3 LTS.

Table IV shows the number of successful compilations along with the total time for each compilation round. In the first round, we successfully compile 51% of our set using the `rustc stable 1.22.1 (2017-11-22)` compiler. Unfortunately, a Rust package's build manifest does not specify the compatible compiler versions; a package may use unstable features from a nightly compiler release, which are not backwards compatible with the stable compiler. By swapping the stable compiler version for the nightly version `rustc 1.24.0-nightly (2017-12-06)`, we compile an additional $4,972$ package version releases. Analysis of the remaining compilation errors reveals that a large number of builds fail due to missing path-based dependencies. These dependencies are incorrectly pointing the download source of a dependency to a local directory instead of CRATES.IO. To resolve this, we use CARGO's internal dependency source rewrite feature in and compile $2,644$ additional package versions. Finally, we observe that several package releases are using native library dependencies which are not installed on Ubuntu 16.04; with them installed, we compile an extra $1,862$ package releases.

TABLE V: Build failure reasons for package versions that did not build after installing native dependencies.

Failure reason	#Builds
Compile error w. error code	13,509 (58%)
Compile error wo. error code, of which	7,272 (31%)
. . . code parsing errors	1,486
. . . conditional compilation errors	1,058
. . . dependency resolution errors	719
. . . type checking errors	278
. . . other errors	3,711
Custom build script failure	2,127 (9%)
Missing system dependencies	137 (< 1%)
Miscellaneous errors	18 (< 1%)
Σ	23,063

Peril 5. Despite our best efforts, we could not compile $23,063$ (31%) package releases. To understand why they fail to compile, we analyze the compiler errors and classify them into five categories in Table V. The majority seem to relate to actual programming faults in the packages, in particular the Rust type checker (e.g., `E0277`, `E0599`, `E0425`), syntactical errors and invalid specifications for conditional compilation. A common reason for these error messages is the improper use of Traits. Overall, we can compile 69% of total releases and at least one release for 88% of packages, roughly double the ratio of previous attempts [33].

C. Generating Unique Function Identifiers

For each constructed call graph in our set of packages, we parse the function names to append it with OPR and package-specific information. The Rust compiler mangles function identifiers in source code to flat C++ namespace-like representations [34]. To construct a UFI from a Rust-mangled identifier, we prepend namespaces in the identifier with appropriate package names and versions, and also with an OPR-qualifier (i.e., CRATES.IO). Due to nested type structures in functions, portions of an identifier can have nested namespaces which complicates the UFI construction. To untangle the nested namespaces in an identifier, we develop a parser that produces syntax tree representations from Rust-mangled function names. With a syntax tree, in a simplified way, we can access and prepend individual namespaces in an identifier.

The Rust mangled function identifiers share some similarities with Rust syntax. To build the parser, we use the parser combinator framework, syn [35] for parsing Rust source code. To validate our parser implementation, we create a corpus containing all function identifiers from our set of constructed call graphs, and then attempt to parse each identifier into a syntax tree. The constructed corpus consists of $111,876,036$ function identifiers. In the first attempt, we use the default Rust parser to process our corpus. In total, $6,030,730$ identifiers generate a parse error. After adding support for brackets (i.e., `<...>`), we reduce the parse errors to 4 million errors. After inspecting the error logs, we identify that the parser fails for namespaces that contain `Impl` structures and anonymous closures. After resolving these cases, we reduce the parse error rate to 0.24%. Therefore, we are able to annotate multiple namespaces with package information in a single identifier with high confidence.

D. Unifying Call Graphs

To generate a single CDN, we merge nodes with the same UFI (from different call graphs) into a single node. Without merging nodes, a naïve concatenation of all individual call graphs resulted in a graph with $60,410,714$ nodes and $178,308,144$ edges. After merging nodes on the function name, we reduced the graph to $7,034,536$ nodes and $19,511,485$ edges, merging on average 8.6 nodes onto a single node. In total, 6,983,046 function nodes have an origin in CRATES.IO. The 51,490 remaining nodes are standard library functions from `core`, `std`, or packages which not hosted on CRATES.IO. Overall, we have a definition for 6,882,760 nodes, 97.8% of all nodes, i.e., we could expand their internal call flow similar to `io::crates::Lib2::used` in Figure 4. This high percentage demonstrates the internal completeness of the buildable part of CRATES.IO and that merging nodes, core to PRÄZI, is useful and occurs often.

E. Evaluation of the Rust CDN

One purpose of the Rust CDN (and PRÄZI at large) is to present a more precise view of the dependency relationships between software components in comparison to a traditional PDN. Since both networks operate on a different abstraction level, we cannot directly compare them. We can, however, reverse-engineer a new, possibly more accurate PDN called $\widehat{\text{PCDN}} \subset$ PDN by "uplifting" the CDN. Only then can we compare it to the original PDN. We construct $\widehat{\text{PCDN}}$ by including a package dependency $\langle \texttt{A}, \texttt{B} \rangle$ in $\widehat{\text{PCDN}}$'s edge set if there is at least one call from a function in package A to one in B in the CDN. This, of course, sacrifices precision in the CDN and thus is an absolute lower bound for possible improvements.

As a caveat, the PRÄZI perils may result in missing calls across packages in the CDN, which in turn makes the $\widehat{\text{PCDN}}$ miss dependency relationships. Therefore, our evaluation focuses on quantifying the differences between PDN and $\widehat{\text{PCDN}}$ and qualitatively exploring their causes. To make the comparison fair, we remove from the PDN all packages that could not be compiled because those could never appear in the $\widehat{\text{PCDN}}$. We then extract the subset of dependency links present in the PDN, but absent in the $\widehat{\text{PCDN}}$. For each dependency, we determine through manual code inspection whether it is correctly or incorrectly absent in the $\widehat{\text{PCDN}}$.

The $\widehat{\text{PCDN}}$ contains $42,827$ nodes and $110,762$ edges. The PDN contains the same number of nodes but has $129,535$ edges. A set difference on the edges of the two networks shows that $18,042$ edges (i.e., 14%) are not in the $\widehat{\text{PCDN}}$. Qualitative evaluation of all 18,042 different edges is practically infeasible, as it relies on manual work. Instead, we select a statistically representative subset of its edges using Cochran's sample size formula [36]. Selecting from a homogeneous set of edges, at a 95% confidence level with a confidence level interval of 5%, we need a sample of $n = 381$ edges that the first author investigated. In Table VI, we break down the results into dependencies that i) should be and are absent in the $\widehat{\text{PCDN}}$ and ii) should be present in it, but are not.

TABLE VI: Manual inspection and classification of 381 different dependency relationships between the PDN and the $\widehat{\text{PCDN}}$.

Categorization	**#Samples**
i) Dependencies absent in $\widehat{\text{PCDN}}$ (and should be)	**133** (35%)
… dependency not declared	73
… invoked in modules but not exported in the library	53
… invoked in test code but not part of the library	7
ii) Dependencies absent in $\widehat{\text{PCDN}}$ (but should not be)	**248** (65%)
ii.1) Call graph generator	114 (46%)
… call inside a generic function	72
… dynamic function call	12
… missing generic definition	8
… C method invocations	22
ii.2) Type-only dependencies	50 (20%)
… imported `Trait` or `Struct`, no function call	50
ii.3) Preprocessor	84 (34%)
… part of functions with conditional compilation	58
… use of macro functionality	26
Σ	**381**

Our qualitative evaluation shows that our Rust CDN can identify several cases (35%) where a regular PDN would definitely lead to false positives reported to developers. At the same time, there is a substantial amount of dependencies that the $\widehat{\text{PCDN}}$ seems to fail to capture. While 65% sounds discouraging for RUSTPRÄZI at first, in the following, we will characterize and explain them in detail, and show that none are a theoretical limitation of PRÄZI.

Of the missing dependency links, almost half are due to shortcomings in the LLVM call graph generator. They come as no surprise, as we found out that we can only claim soundness for static dispatch (see Section V-B). Plugging a better call graph generator in, switching to a language with better generators (such as Java), or not using these language features thus resolves the problem, which is not inherent to PRÄZI. The remaining cases represent possible future improvements to PRÄZI. A fifth of the missing cases are data-type only dependencies that a call graph cannot capture, e.g., importing a struct of another package. This suggests going beyond call graphs for PRÄZI. Another third of the missing dependencies are due to conditional compilation: a function is only compiled-in if the appropriate feature toggles are on. We mitigate this issue when, we rerun the call graph generation process for every possible feature toggle combination. We have thus shown that none of the missing cases is a principal shortcoming of PRÄZI.

To verify the generalizability of this evaluation, the first two authors conducted an inter-rater reliability study. We cross-validated 20 randomly selected pairs of dependencies. After an independent assessment and comparison of the results, both raters agreed that 19 ratings of the main rater were correct ($p_0 = \frac{19}{20} = 0.95$). The naïve likelihood of a random agreement is $p_c = 0.5$. This gives us a Cohen's κ of $\frac{p_0 - p_c}{1 - p_c} = \frac{0.95 - 0.5}{1 - 0.5} = 0.9$ [37], which signifies (almost) perfect agreement [38], increasing trust in the correctness and generalizability of the manual inspection.

VI. Case Studies

To demonstrate the effectiveness of our approach, we present two cases studies, namely security vulnerability propagation and function deprecation.

A. Security Vulnerability Propagation

Perhaps the most common application of dependency networks is the study of the spread of security vulnerabilities [4], [7], [39]. Companies, such as BLACKDUCK [13], TIDELIFT [11], and GITHUB help projects identify whether they are affected by publicly disclosed vulnerabilities in their dependencies.

With our first case study, we aim to assess whether the call-based representation of an OPR could yield higher precision in the security vulnerability propagation case and how severe its soundness issues would be in such a real-world test. We compute how nine security advisories from Rust's security advisory database, RUSTSEC [40], affect other packages, using both our Rust CDN and PDN. Table VII presents a detailed overview of the advisories we examined along with the number of affected package versions per network. From the initial set of nine advisories, we could not analyze the `security-framework` advisory because it is macOS-specific, `sodiumoxide` because its vulnerable versions generate build failures, and the `openssl` advisory because it is related to configuration rather than vulnerable code. From the six remaining ones, we skip `cookie` because it does not have any callers and `smallvec`, whose vulnerability is in a generic function, which the current version of the Rust CDN cannot cover due to limitations in the call graph generator (see Section V-E).

By construction, our Rust CDN is precise, but could miss function calls to dependent packages, while the Rust PDN is sound, but overapproximates the number of used packages (see Section V-E). To compute the accuracy of the two networks, we need to establish a ground truth: for each security advisory, we collect the direct dependents of the vulnerable package; then, we manually investigate whether there exists a function call from a dependent to any function of the vulnerable package. We compare the Rust PDN and the Rust CDN (converted to a $\widehat{\text{PCDN}}$, as in Section V-E) against the ground truth and create a confusion matrix:

i) *True Positive* (TP) means correctly flagging a package as vulnerable when a vulnerable function call exists.
ii) *False Positive* (FP) means falsely flagging a package as vulnerable when there is no invocation of a vulnerable function.
iii) *False Negative* (FN) means flagging a package as *not* vulnerable when there is at least one evident call of a vulnerable function.
iv) All remaining cases are *True Negatives* (TN).

Finally, we use the standard binary classification metrics precision, recall, and accuracy to compare their performance.

From the results in Table VII we observe that the $\widehat{\text{PCDN}}$ reports a much lower number of affected package versions, on average, 83% fewer affected packages than the PDN. By analyzing the direct dependencies of the vulnerable packages, we establish that a high percentage of these affected packages are in fact false positives in the PDN. This score would be even more in the advantage of our $\widehat{\text{PCDN}}$ were we to look at the full transitive closure (which we have not done because of the high manual workload associated with it). Although our $\widehat{\text{PCDN}}$ has a lower recall score than a PDN, our $\widehat{\text{PCDN}}$ has an accuracy which is three times higher than the state-of-the-art PDN, signifying that CDNs yields very high precision benefits over traditional PDNs, even in real case scenarios with suboptimal tools (see Section V).

B. Deprecation Impact Analysis

As packages evolve, their public API changes to accommodate improved functionality. As a consequence, functions can become obsolete. Several programming languages, have a special mechanism to annotate obsolete functions, either in the API documentation (e.g., in Python) or as a language feature (e.g., in Java). While annotating functions as deprecated is a common practice among developers, cleaning up deprecated code is a far more challenging task. In a qualitative study, Sawant et al. report that API producers are "wary about removing deprecated features from their API" and "mostly have no preset protocol for removal" [41]. The reason is that developers cannot know the impact of such cleanups.

By linking dependent functions together, PRÄZI enables us to perform change impact analysis at the OPR level. Using an PRÄZI CDN, developers can estimate the impact of removing deprecated functions, both to direct API clients and transitively. To demonstrate this, we calculate the impact of function deprecation within CRATES.IO.

In Rust, deprecated functions can be annotated with a `#[deprecated]` attribute: the Rust compiler will fail compilation if a program links to a deprecated function (unless a `#[allow(deprecated)]` is specified). To find deprecated functions, we extract function signatures prepended with a `#[deprecated]` attribute. In total, we find 721 deprecated function signatures from 190 package versions in 43 unique packages. We only consider deprecated functions and their callers, i.e., functions who call the deprecated functions directly. Among the 190 package versions, only 42 package versions have callers, reducing our search space to 43 deprecated functions. Then, we manually match the deprecated function to its UFI in the Rust CDN. We are able to find a UFI for 24 deprecated functions; the remaining ones are missing due to the reasons identified in Table VI. We perform a propagation analysis (similar to Section VI-A) for the 24 cases. In total, 13 of 24 deprecated functions have calling functions in other packages and together affect 163 package versions,

TABLE VII: Results for the security advisory propagation analysis.

Package	Function	#Packages PDN	#Packages $\widehat{\text{PCDN}}$	#Cases (Weight)	Precision PDN	Precision $\widehat{\text{PCDN}}$	Recall PDN	Recall $\widehat{\text{PCDN}}$	Accuracy PDN	Accuracy $\widehat{\text{PCDN}}$
`base64`	`encode_config_buf`	257	51	128	0.25	1	1	0.68	0.25	0.92
`cookie`	`parse_inner, max_age`	0	0	0	-	-	-	-	-	-
`hyper`	`Headers::set`	21	3	2	1.00	1	1	0.5	1	0.5
`smallvec`	`insert_many`	1,581	0	325	-	-	-	-	-	-
`tar`	`unpack_in`	502	31	61	0.62	1	1	0.71	0.62	0.82
`untrusted`	`skip_and_get_input`	5,655	564	291	0.25	1	1	0.43	0.25	0.86
Σ (or weighted average)		8,016	649	482	0.30	1	1	0.53	0.30	0.87

both directly and indirectly. This amounts to $\frac{163}{42{,}827} = 0.38\%$ of the $\widehat{\text{PCDN}}$.

Table VIII shows an overview of the results. In its first column, we show which deprecated functions belong to which package version in an encoded format. For example, the `platform_{window/display}` represents the two functions `platform_window` and `platform_display`. We observe for the dependent $\widehat{\text{PCDN}}$ sub-graph of each respective package version that, on average 48% of dependents call directly or indirectly the deprecated functions. In other words, almost half of their callers break if the deprecated function is removed. This form of information can aid library maintainers with vital information about whether it is already safe to remove a deprecated function, particularly when a large number of transitive consumers are calling it potentially unknowingly, due to a transitive call chain.

VII. Implications

While researchers and practitioners widely use PDNs meta-information for various decision making tasks, e.g. related to updates [7] or security [39], it seems that their precision is a largely overlooked aspect. Indicatively, in the case of CRATES.IO, 35% of the dependency links included in a dependency network extracted from metadata should not be there. The work we present in this paper uncovers several implications in the way that both researchers and practitioners use PDNs, which we briefly present below.

A. Implications for Researchers

Our study showed that the generation of sound, yet precise call graphs remains an important research problem for the feature-richness of modern languages. The quality of a PRÄZI CDN is directly dependent on the quality of the call graph generator. In the Rust case, a Rust-specific call graph generator needs to be developed as a plug-in to the compiler in order to i) accommodate for features such as macros that are only visible during compilation, and ii) be always up to date with the latest language features.

TABLE VIII: All called deprecated functions.

Function	Package	$\widehat{\text{PCDN}}$	#Affected by Dep. Fn.s
`OwnedKVList::{new/id/root}`	`slog:::1.7.1`	93	63
`platform_{window/display}`	`winit::0.7.6`	91	50
`platform_{window/display}`	`winit::0.9.0`	31	16
`platform_{window/display}`	`winit::0.8.3`	44	14
`platform_{window/display}`	`winit::0.6.4`	36	12
`get_formats_list, get_name`	`cpal::0.4.6`	16	8
Σ 13	6	311	163

We also demonstrated the volatility and fast-paced nature of dependency resolution in OPRs by comparing timed snapshots of CRATES.IO. It seems crucial to include this aspect in future studies on OPRs.

Finally, we showed the granularity and precision that PRÄZI offers opens the door to many new analyses not possible before, for example change impact analysis across an OPR. There are several strands of research opportunities to improve PRÄZI internally (which we list in Section V-E) or externally, e.g., by applying it to a different programming language.

B. Implications for Practitioners

With PRÄZI, we have described and implemented a technique that can reduce the number of false positive in a wide array of current applications that suffer from bad precision. Practitioners could improve our prototypical RUSTPRÄZI implementation to make it sound in all cases relevant to them.

Due to the large number of broken packages on CRATES.IO, builders of OPRs should consider validating packages before publishing them, similar to CPAN [42] or CRAN [43]. Moreover, to make the entire build chain setup of packages more reproducible, builders of OPRs should perhaps have similar goals to what Debian wants to achieve [44].

Many IDEs support a “remove unused imports” analysis. PRÄZI allows the development of a “remove unused library” feature, cleaning and optimizing the dependency set of projects.

C. Threats to Validity

In the case of security, it is important for PRÄZI to guarantee that no false negatives exist, i.e., that the analysis is fully sound. The PRÄZI technique is only susceptible to this threat insofar as the used call graph generator is sound, which lies outside the scope of this paper. Our initial Rust implementation guarantees soundness for statically dispatched method calls, but is only “soundy” otherwise [45]. If Rust programmers avoid dynamic dispatch, our prototype is sound by construction (we have specified and measured the exceptions in Table VI). Moreover, security warnings only serve as an example application for meta-warnings attached to a program. Similarly, bug, performance, deprecation, or other advisories exist, e.g., in addition to RUSTSEC’s security advisories, similar such advisories exist for performance and semantical bugs [46], [47]. Conceivably, achieving a very high precision for such advisories while missing rare cases is an excellent trade off for many developers. In fact, as Livshits et al. argue, sacrificing soundness for precision is not just a common tradeoff for static analysis tools, even for vulnerability detection programs such as Fortify [48], but it is actually necessary for practical use [45].

VIII. Related Work

In this section, we briefly present previous PDN-based analyses and compare them to our work.

The aftermath of the leftpad incident has led to a surge of studies around OPRs. Researchers have constructed dependency networks of OPRs to trace the impact of security problems [4], [7], [39], to study the evolution of language ecosystems [4], [5], [49], [50], or to recommend update paths for projects [6]. In the area of security, notably, Kikas et al. [4] have shown there exist packages that can break up to 30% of packages in both NPM and RUBYGEMS. Moreover, Kula et al. [7] and Decan et al. [39] also indicate a large percentage of affected packages. However, our security case study has shown that these studies may grossly overapproximate the risk.

To the best of our knowledge, ours is the first attempt to construct a dependency network on a function call level. The closest to our work is an initial vulnerability study by Zapata et al. [51] where the authors manually established that 73.3% of 60 JavaScript projects marked as vulnerable by a dependency checker are not. The results are on similar lines as ours, saying vulnerable functions in many cases are not called.

IX. Future Work

PRÄZI is a novel technique that is first to apply lightweight static analysis on whole OPRs. While it is comprehensive and can already provide valuable insights, our evaluation (Section V-E) showed that it is not complete. In this section, we present how PRÄZI can be improved and practically exploited.

First, the buildability of an OPR plays a crucial role in the completeness of the PRÄZI CDN; researchers could setup several build environments (operating system, compilers, compile flags) to generate a maximally complete call graph. PRÄZI can also be combined with product line research to help mitigate the issues arising from conditional compilation.

The soundness and completeness of the PRÄZI CDN can be significantly improved through language-specific call graph generators. In several mainstream languages, e.g. Java, precise call graph generators with few constraints exist, such as Soot [52] and Wala [53]. For the purposes of PRÄZI, researchers could work on enriching the static with dynamic call graphs obtained from running a project's tests.

To make a full PRÄZI implementation practically useful, the size of the generated CDN must be tamed to cope with OPRs the size of MAVEN or NPM. One way to do this would be to "compress" the CDN with the realization that only a small part of a library changes between versions; therefore, the existing "identical" nodes (and their subtrees) could be annotated with a version range and only a possibly small part of new nodes would need to be created. A PRÄZI-specific database would be able to aggressively compress the CDN graph by exploiting its structural and temporal properties: its edges are immutable, it only grows by appending nodes and edges, while the degree distributions for nodes follow power laws.

Apart from our two case studies, CDNs enable a rich set of applications that researchers can work on. A potentially interesting question would be to explore the application of change impact analysis for aspects other than deprecation, e.g.: "If I change the semantics of this function, who will have to be notified?" Researchers can also investigate issues such as dependency health and quality, by coupling the CDN information with analytics from the projects source code repository. Furthermore, when we annotate function nodes with file and license information, the CDN could be used to study module-level licensing incompatibilities.

X. Conclusion

We presented a generic approach, PRÄZI, to construct fine-grained dependency networks at the function call level. With PRÄZI, we implemented RUSTPRÄZI, a precise call-based dependency network of Rust's CRATES.IO package repository to demonstrate improvements over state-of-the-art package-based dependency networks (PDNs). After building all releases of CRATES.IO packages, RUSTPRÄZI is able to represent 69% of CRATES.IO. In our evaluation, we qualitatively analyzed 381 cases to understand how accurately RUSTPRÄZI represents dependency relationships of CRATES.IO. In 35% of the cases, a PDN overapproximates the dependency relationships. Despite the fact that we encountered a number of soundness issues in the remaining cases, we showed that none of them is a fundamental limitation of the PRÄZI technique. Moreover, our two case studies demonstrated the usefulness of the generated CDN even with these issues. In a case study of security propagation, we have shown that RUSTPRÄZI is three times more accurate than a state-of-the art dependency network, showing that the function-level granularity reduces the number of false positives. In our second case study, we investigate the propagation of deprecated functions and quantify the overall dependence on such functions in CRATES.IO.

This paper makes the following contributions:

i) A novel technique, PRÄZI, to create precise, call-based dependency networks.
ii) An open source Rust implementation, RUSTPRÄZI of the general PRÄZI technique to show its practical feasibility.
iii) An evaluation of RUSTPRÄZI and quantification of its shortcomings.
iv) Two case studies on security and deprecation warnings, that demonstrate PRÄZI's usefulness.
v) Derivative datasets including the CDN, its evaluation, and the case study data is available as a replication package. [1]

[1] https://DrNXs1ALFzzQth4r.github.io

REFERENCES

[1] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A large-scale empirical study on software reuse in mobile apps," *IEEE software*, vol. 31, no. 2, pp. 78–86, 2014.

[2] W. B. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE transactions on Software Engineering*, vol. 31, no. 7, pp. 529–536, 2005.

[3] V. R. Basili, L. C. Briand, and W. L. Melo, "How reuse influences productivity in object-oriented systems," *Communications of the ACM*, vol. 39, no. 10, pp. 104–116, 1996.

[4] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 102–112.

[5] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "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, 2017, pp. 385–395.

[6] R. G. Kula, C. De Roover, D. M. German, T. Ishio, and K. Inoue, "Modeling library dependencies and updates in large software repository universes," *arXiv preprint arXiv:1709.04626*, 2017.

[7] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[8] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 109–118.

[9] E. Constantinou and T. Mens, "An empirical comparison of developer retention in the RubyGems and NPM software ecosystems," *Innovations in Systems and Software Engineering*, vol. 13, no. 2-3, pp. 101–115, 2017.

[10] M. Han, "Introducing security alerts on GitHub," 2017, https://github.com/blog/2470-introducing-security-alerts-on-github. Accessed January 26, 2018.

[11] I. Tidelift, "Tidelift," 2017, . Accessed August 10, 2018.

[12] I. Justine Tunney, Google, "Operation rosehub, Google open source blog," 2017, . Accessed August 10, 2018.

[13] I. Black Duck Software, "Open source license compliance," 2017, . Accessed August 10, 2018.

[14] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "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, 2016, pp. 470–481.

[15] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 41–50.

[16] ——, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011.

[17] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.

[18] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 120–126.

[19] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, *Practical virtual method call resolution for Java*. ACM, 2000, vol. 35, no. 10.

[20] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for JavaScript IDE services," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 752–761.

[21] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini, "Call graph construction for Java libraries," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 474–486.

[22] T. R. C. Team, "The Rust programming language blog," 2015, https://blog.rust-lang.org/2015/05/15/Rust-1.0.html.

[23] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 158–191, 1998.

[24] M. Sulír and J. Porubän, "A quantitative study of Java software buildability," in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 2016, pp. 17–25.

[25] R. L. Documentation, "The Rust programming language," 2018, https://doc.rust-lang.org/book/first-edition/trait-objects.html.

[26] "Crates index," 2018, https://github.com/rust-lang/crates.io-index.

[27] "Crates api," 2018, https://crates.io/api/v1/crates/.

[28] "Functions," 2018. [Online]. Available: https://doc.rust-lang.org/stable/reference/items/functions.html

[29] "Call expressions," 2018. [Online]. Available: https://doc.rust-lang.org/stable/reference/expressions/call-expr.html

[30] "Method-call expressions," 2018. [Online]. Available: https://doc.rust-lang.org/stable/reference/expressions/method-call-expr.html

[31] "Types: Trait objects," 2018. [Online]. Available: https://doc.rust-lang.org/stable/reference/types.html#trait-objects

[32] "Macros," 2018. [Online]. Available: https://doc.rust-lang.org/stable/reference/macros.html

[33] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, 2017.

[34] rust lang/rust, "The rust linkage model and symbol names." [Online]. Available: https://github.com/rust-lang/rust/blob/0cf0691ea1879a84d09d53a19e0f0b06827cf95a/src/librustc_codegen_utils/symbol_names.rs

[35] "Syn," 2018, https://github.com/dtolnay/syn.

[36] W. G. Cochran, *Sampling techniques*. John Wiley & Sons, 2007.

[37] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[38] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[39] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *International Conference on Mining Software Repositories*, 2018.

[40] "Rustsec advisory database," 2018, https://github.com/RustSec/advisory-db.

[41] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding developers' needs on deprecation as a language feature," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 561–571. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180170

[42] "Comprehensive perl archive network," 2018. [Online]. Available: https://www.cpan.org/

[43] "Comprehensive r archive network," 2018. [Online]. Available: https://cran.r-project.org/

[44] "Debian: Reproducible builds," 2018. [Online]. Available: https://wiki.debian.org/ReproducibleBuilds

[45] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[46] "Rust performamnce warning," 2018. [Online]. Available: https://github.com/servo/servo/issues/17399

[47] "Rust semantical bug," 2018. [Online]. Available: https://github.com/crossbeam-rs/crossbeam-epoch/pull/53

[48] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis." in *USENIX Security Symposium*, vol. 14, 2005, pp. 18–18.

[49] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 2016, pp. 351–361.

[50] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, Feb 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9589-y

[51] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for NPM JavaScript packages," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*, 0 (to appear).

[52] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 214–224.

[53] "Wala," 2018. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page