

Investigating dependency code reuse using callgraphs

Master's Thesis

Algirdas Jokūbauskas

Investigating dependency code reuse using callgraphs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Algirdas Jokūbauskas



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Investigating dependency code reuse using callgraphs

Author: Algirdas Jokūbauskas
Student id: 4744233
Email: a.jokubauskas@student.tudelft.nl

Abstract

Software development is more and more reliant on external code. This external code is developed, bundled into packages and shared using package repositories like crates.io or npm. Reusing shared code bundles greatly improves development speed but without proper care and knowledge of included external code it can cause issues as well. Over-reliance on simple trivial packages that can be easily implemented locally can cause severe consequences like not being able to build if the package is no longer available. Furthermore, including large packages when only a small amount of it is being used can cause software projects become bloated with unnecessary code. This thesis proposes several metrics: leanness index, software composition index and utilization index that quantify how software projects reuse their dependencies or how much of the dependencies are utilized by their dependents. All three of the metrics are based on full function callgraph of the applications. Computation of these metrics was implemented for Rust and applied on every package in Rust package repository crates.io. General findings showed that dependency leanness was rather low: 21.7% of all packages in crates.io used less than 5% of their included dependencies while 95% of all crates used less than 62.4%. Another discovery revealed that packages in crates.io tend to consist mostly of code from external dependencies as opposed to local code: on average 91% of lines of code come from external dependencies. Lastly, using utilization index functions that were never used by any other crate in crates.io were identified: in 95% of packages, 71% of their callgraphs are never used.

Thesis Committee:

Chair:	Prof. Dr. Georgios Gousios, Faculty EEMCS, TU Delft
University supervisor:	Joseph Hejderup, Faculty EEMCS, TU Delft
Committee Member:	Dr. Asterios Katsifodimos, Faculty EEMCS, TU Delft

Preface

When I started writing this thesis I did not even know how to code in Rust. It was a challenging language to learn but I'm glad I did it. Majority of the project was done in pure rust and it made me feel more confident in its durability. I used it to develop many tools and infrastructure that ended up not being used for the final result which gave me quite a bit of new knowledge, like writing kubernetes controllers or graph database queries.

Many people helped me throughout the project. It all started with having a Skype interview with Georgios Gousios who introduced me to the thesis and later provided guidance and general advice. I've had several discussions with Anand Ashok Sawant, Marco di Biase and Moritz Beller who helped me with finding relevant research to base my work on. However, I'd like to thank Joseph Hejderup whose previous paper was the inspiration and the starting point for this thesis, and who also helped me tremendously throughout, by having regular meetings and providing valuable feedback.

Algirdas Jokūbauskas
Delft, the Netherlands

Contents

Preface	iii
Contents	v
List of Figures	vii
1 Introduction	1
1.1 Problem Statement	3
2 Background	5
2.1 Dependencies	5
2.2 Package Registry	6
2.3 Callgraph	7
2.4 Code Analysis	7
2.5 Commonly used terms in this thesis	9
3 Proposed code reuse metrics	11
3.1 Leanness index	11
3.2 Software Composition index	15
3.3 Utilization index	16
4 Analysis Pipeline	21
4.1 Retrieving the callgraph dataset	22
4.2 Format Preprocessor	22
4.3 Graph Traversal	22
4.4 Dependency Analysis	24
4.5 Implementation	26
5 Results	27
5.1 RQ1: Leanness index	27
5.2 RQ2: Software Composition index	31

CONTENTS

5.3	RQ3: Utilization index	32
5.4	Qualitative example of several packages	33
6	Discussion	39
6.1	RQ1: Leanness Index	39
6.2	RQ2: Software Composition Index	40
6.3	RQ3: Utilization Index	41
6.4	Threats to Validity	41
7	Related Work	43
7.1	Dependency Networks	43
7.2	API Research	44
7.3	Unused API code analysis	45
7.4	Software metrics	46
8	Conclusions and Future Work	49
8.1	Conclusions	49
8.2	Future work	49
	Bibliography	51

List of Figures

2.1	Example of a dependencies section in Cargo.toml file from a Rust project . . .	6
3.1	Example callgraph for package	12
3.2	Example of an extended callgraph for package	13
3.3	Several callgraphs containing Package A and Package B as a dependency . . .	17
4.1	Pipeline of the metrics analysis	21
4.2	Callgraph data format	22
4.3	Sample callgraph	24
4.4	Sample callgraph of dependency analysis	25
5.1	Graph of leanness index distribution using two different methods	27
5.2	Box plots for leanness indices calculated using nodes and lines of code	28
5.3	Share of functions from package dependencies that contain Lines of Code in- formation	29
5.4	Graphs comparing leanness index when calculating ratio of used versus unused public functions of a dependency versus analyzing its entire callgraph	29
5.5	Box plots leanness index when calculating ratio of used versus unused public function LOC of a dependency versus analyzing its entire callgraph	30
5.6	Graph of leanness index distribution comparison to public API usage method .	31
5.7	Graph of software composition index in log scale.	31
5.8	Box plot of software composition index.	32
5.9	Graph of utilization index distribution	33
5.10	Box plot of utilization index distribution	33
5.11	Dependency use in Pathfinder - 0.6.5	34
5.12	Share of different dependencies in Pathfinder 0.6.5	35
5.13	Dependency use in Rodio - 0.10.0	36
5.14	Share of different dependencies in Rodio 0.10.0	37
8.1	IDE plugin example to visualize leanness index	50

Chapter 1

Introduction

These days nothing in software development is done from scratch. Developers constantly use each others' code to speed up work and make it better. This sharing is usually done through packages that encapsulate libraries. They are collections of code that serve one specific purpose and can be included in any project to add functionality. Doing so creates a dependency in that project and makes the project a dependent of said package. When running or compiling a project, these dependencies need to be resolved, which involves figuring out the correct version of the dependency, then either downloading its source code and compiling it or downloading the already compiled artefacts.

To speed up package development developers can include other packages into it as a dependency. Those packages as a result can include others as well. Therefore, including several dependencies in a project can create a large dependency tree that the project needs to resolve before running or building. To easily manage these dependencies, most programming languages nowadays, have package repositories like Maven [17] or crates.io [4]. They contain a large number of user created libraries that can be easily added to any project.

Using these package managers and external libraries in general can be a great improvement to the software development process. Firstly, it reduces time to minimal viable product by not requiring every team develop redundant code that has already been implemented by others. Security can also be greatly improved when using well supported external code. Such libraries would get constant bug and vulnerability fixes that a small team might not even know about. Furthermore, new developers using these libraries can learn a great deal about good code design.

However, over-reliance of external code, bad practices of popular library developers or vulnerabilities can cause many issues as well. There have been various incidents in the past like the left-pad incident [8] where a developer has unpublished a popular package which caused many other packages and even projects down the line, fail to run. Another example was a malicious attack called the event-stream incident [24]. In this case, single point of failure issue was apparent when an attacker took over a popular npm package called "event-stream" and injected malicious code which quickly propagated through its dependents.

Besides famous incidents, like the ones mentioned, a general aspect of software development is code bloat. Bloat can come in many forms, like dead, unused functions, large amounts of duplicate code, inefficiently compiled code, and inclusion of many unused or

outdated dependencies. The last one is what this thesis will focus on. Having large amounts of dependencies is not inherently bad, however there are generally good practices a developer can use to ensure it. Ensuring good dependency hygiene in a project requires adding packages only if they are necessary, making sure the packages that are in use are not abandoned and understanding the entire dependency tree that results in adding one dependency. With current tooling available in IDEs (Integrated Development Environments) these things are not obvious or easily understood. In most cases developers need to explicitly investigate all dependencies that are included to see if any of them have security issues, are out of date, or are not necessary and could easily be re-implemented in the project itself.

There has been a lot of research in the field of dependency usage. Over 40% of included dependencies and around 70% of public APIs [9] are never used. However, there seems to be a lack of studies that try to investigate entire callgraphs of dependency trees, which could provide valuable insight in how developers behave when it comes to dependency use and how this behaviour is reflected in the obfuscated callgraph that results from it. For example, a callgraph would reveal how including one package and calling one out of ten of its public APIs utilizes the entire dependency tree and may encourage the developer to reconsider such use.

In order to fill this gap and evoke more discussion on dependency research this thesis will focus on quantifying code reuse on a callgraph level. This quantification would be evaluated by implementing it for Rust programming language [21] and running it on Rust's package manager crates.io [20] ecosystem. To quantify code reuse I will define 3 metrics: leanness index, which describes how much code is actually being used from the included dependency tree, software composition index that defines a ratio of code coming from dependencies compared to total code in the project, and lastly, utilization index which provides insight into how much each dependency is being utilized by its dependents in crates.io. Metrics were calculated by first downloading and compiling every crate from crates.io, then the compilation artefacts analyzed and one callgraph generated, creating descriptive json file for each crate. Afterwards, the callgraphs were traversed, their properties measured and saved into a database for the empirical study.

General findings were that 21.7% of all crates used less than 5% of code from their dependencies and 95% of all crates used less than 62.4% of dependency code. Another interesting finding was that once a project starts using dependencies, its local code is overwhelmed by the amount of code that gets included externally. On average, crates that include dependencies have 91% of their lines of code come externally. And lastly, it would seem that many packages are not utilized much by its dependents: in over half (57%) of all crates, only 10% of their callgraphs are in use.

In the following chapter I will provide background information on concepts and terms used in the thesis. Afterwards I will define and justify the proposed metrics in detail including several examples. Then the analysis pipeline will be discussed, explaining how the metrics get generated from the source code of a crate. After this, the results are presented and later discussed. Then a related works section provides an overview of research that was done in the field. Lastly, the thesis will conclude, giving various suggestions for future work and utilization of the metrics.

1.1 Problem Statement

The following are the research questions that the thesis will try to answer.

1.1.1 RQ1: How much of the code, included by external dependencies, is actually being used by a project?

Many current approaches that look into code reuse through dependencies end up analyzing only the public APIs exposed by the top level direct dependency. This question is here to probe the dependency tree of packages and all private functions within the direct and transitive dependencies. Knowing this information a developer would have an understanding on how bloated their code is.

1.1.2 RQ2: How much of the crate's code comes from the crate itself and its external dependencies respectively?

The first question needs to have context to have an impact, and the second one provides that context. If it is known that a software project does not utilize its dependencies much, it is important to also know how that compares to the codebase of the project itself. If it only uses a small part of a small package, it might not be a severe issue. This question would provide a ratio of code coming from dependencies and local code.

1.1.3 RQ3: How much is each package utilized by its dependents?

Last question, flips the focus around, looking at packages and not projects that include them. It would be valuable and interesting to know how other projects that include the dependency use it. With this information, package maintainers would have more to work with when designing their packages and possibly splintering them into smaller ones to extract the most used core functionality.

Chapter 2

Background

In this chapter I will try to go over the core concepts that need to be understood to read the rest of the thesis. This includes how dependencies work generally and in Rust, the functionality, purpose, and creation of package managers and explaining the concept of callgraphs.

2.1 Dependencies

Project dependencies are described in metadata files. The metadata file is located in the root of the project, which lists its every dependency and the version of that dependency which follows the semver[23] range definition described below. In case of Rust, this data is stored in Cargo.toml file in the root of the project. An example of this file can be seen in Figure 2.1. These semver range definitions of dependencies can potentially resolve to different results depending on available releases in the package repository and the current timestamp of the compilation. For example, if a version requirement is set to `^1.0.2`, when the project is built it will retrieve the latest version of the package between 1.0.2 (inclusively) and 2.0.0 (exclusively). During the compilation of a Rust project, exact versions are resolved and saved in Cargo.lock file. In any future compilations, Cargo.lock file is used to choose a version of every dependency that was used before, but making sure that it still fits the range definitions defined in Cargo.toml file. If it no longer fits, the dependency is updated. Cargo.lock file can be considered as a definition of the entire dependency tree that starts at the root application.

2.1.1 Semver

Semver (semantic version) defines a way to version software, so that every developer can follow the same logic when releasing packages. The main component of semver are three numbers divided by dots like so: 1.3.10. The first number in this string is called a major version. It is incremented when a dependency introduces API changes that may be incompatible with previous releases. The second number is called minor. It is incremented when the API is updated in a backwards compatible way. Last number is called a patch. It is incremented when applying bug fixes in a way that does not change the API. A version is

2. BACKGROUND

higher when the left most number that is not the equal is higher than another version. Besides these three numbers, there can also be extensions specifying pre-release versions or other metadata. Every release of a dependency to a public registry should get a new semver that follows these definitions.

When including dependencies with a semver, a developer can ask for a specific version that matches one specific release. However it is also possible to define a range. The follow are several examples of range definitions:

1. `>1.2.3` - The highest version that is higher than the one specified.
2. `>1.2.3 <2.3.4` - The highest version that fits in between 1.2.3 and 2.3.4 not including them in this case.
3. `~1.2.3` - The highest patch version up to, but not including the next minor. In this example it would fit 1.2.10, but not 1.3.0.
4. `^1.2.3` - The highest minor version up to, but not including the next major. In this example it would fit 1.5.10, but not 2.0.0.

```
[dependencies]
semver = "0.9.0"
semver-parser = "0.9.0"
serde = "^1.0.90"
serde_derive = "1.0.90"
serde_json = ">1.0.39"
```

Figure 2.1: Example of a dependencies section in Cargo.toml file from a Rust project

2.2 Package Registry

Most modern programming languages have one or more package repositories that are used to freely publish, update and retrieve packages. Java has Maven and Groove, NodeJs has npm and Rust has crates.io. Any developer can create their own package and publish it to these repositories, which can either require source code or only compiled artefacts. These published packages can later be updated with new bugfixes and releases. Most repositories also provide their own package managers that provide software tools which help in performing these tasks. For Rust, developers can use Cargo. When using Cargo to build or run an application, various actions are performed in the background. For example, after editing the previously mentioned Cargo.toml file seen in Figure 2.1, and running the `Cargo build` command, Cargo will download and install or update all required dependencies based on the requirements in the file and the current state described in Cargo.lock file.

2.3 Callgraph

A callgraph is a representation of all functions and function calls in a software project. In this graph, every node represents a function. Each of the nodes may also contain metadata information about the function, which can include its signature, the package and version it belongs to and its accessibility. The edges in this graph represent function calls. They are directional. Such a graph would in many cases be acyclic, unless the code contains any direct or indirect recursive functions. This graph could only be generated by analysing the entire codebase.

2.4 Code Analysis

There has been a lot of research in the field of code analysis and parsing. This section describes the most common approaches and the one chosen for the project. A similar study on Java packages is performed by Anand Ashok Sawant and Alberto Bacchelli [22]. Their goal was to create a fine-grained approach to API usage analysis. The paper also has an overview on most popular approaches to analyzing code usage of client libraries: text matching, analyzing compilation artefacts, partial program analysis, dynamic analysis and abstract syntax tree.

2.4.1 Text matching

The first approach is matching text in the code like it has been used in [18] and [26]. This approach involves matching the imports of external libraries and looking for their invocations inside the text file itself. The main drawback of this approach is that import names may have conflicts in cases where the same method name is used. Furthermore, when analysing statically typed languages, this method loses the ability to use defined types to gather information.

2.4.2 Analyzing compilation artefacts

Another approach is to compile the code and analyze compilation artifacts for precise method usage. This approach is used by [15] and Präzi [10]. Such an approach would provide very precise type and invocation information in statically typed languages like Java or Rust, but it also comes with its downsides. Analyzing an entire dependency network requires access to source code of all dependencies which also need to be compilable. The problem was also encountered by Hejderup et al [10], where they could not compile 31% of package releases in crates.io. Furthermore, analyzing such a network requires retrieving and compiling code for every version of every package since several versions can be included in each project. This requires a large amount of resources and error handling in case of failures.

2.4.3 Partial program analysis

API usage can also be analyzed by partial program analysis. This approach combines the analysis of compiled artefacts and text matching to eliminate drawbacks of those two approaches. With partial program analysis, compiling every dependency is no longer needed since the artefacts of downloaded public APIs are enough. It takes types gathered from these artefacts and matches them to the uncompiled files calling them. Such an approach was developed and analyzed by Dagenais et. al. [5]. They have created an Eclipse plugin that parses partial files and matches those to compiled external API artifacts. The result proved to be very accurate and did not require full compilation like in bytecode analysis. However, it requires using the Eclipse environment and is very Java specific. The implementation used Eclipse's context to gather information, thus analyzing an application would require loading it into a running instance which hinders scalability.

2.4.4 Dynamic Analysis

Dynamic Analysis could also be used to generate a callgraph. It involves analyzing the code as it is being executed. Such an approach is the most accurate one, since it also takes code logic into account to determine whether an API is being called, not just referenced. However, there are several problems with this approach. First of all, it is highly resource intensive. It would be unfeasible to perform it on such a large scale as an entire dependency network. Another problem is the requirement of external resources, like databases or specific hardware, in order to run it. Lastly, it is hard to have full coverage of all code since the callpaths are so dependent on incoming data. It is likely that the resulting graph would be incomplete.

2.4.5 Abstract Syntax Tree

Lastly, AST analysis involves transforming code files into an Abstract Syntax Tree (AST). This tree is a representation of the code where each variable, statement and invocation is a node. Parsing it can also provide accurate type information which helps when trying to ensure accuracy in API calls. This was also the approach Sawant et. al. [22] used in their paper.

2.4.6 Chosen callgraph method

This project started off by using the Präzi [10] approach, since it was the only one available for Rust. However, later on a better and more accurate approach was developed by Konstantinos Triantafyllou, a master student at ETH Zürich [14] (currently under development and private). His benchmark [13] showed that the LLVM artefact approach used by Präzi, misses all dynamic dispatches. These issues are alleviated when using the his callgraph generator. It is currently the most sound callgraph generator for Rust.

2.5 Commonly used terms in this thesis

This section contains an overview of domain-specific terms used in this thesis and their definitions.

Table 2.1: Definitions

Term	Definition
Dependency	External package added to a software project required to compile or run
Direct dependency	A dependency added to a software project explicitly by the developer
Transitive dependency	A dependency that was added to a software project as a result of adding a direct dependency
Dependent	A software project (application or package) that depends on some package
Dependency tree	A tree graph of all dependencies needed to compile or run a software project
Dependency Network	A large graph that describes relations of all packages in a package repository

Chapter 3

Proposed code reuse metrics

To answer the research questions, three metrics for measuring reuse in any project with dependencies are proposed.

3.1 Leanness index

The name leanness for this index was chosen to reflect how much excess, unnecessary code there is in an application. The term lean could also be used when describing a budget that does not have wasted expenditures or a human body that has low amount of fat.

3.1.1 Goals of the Leanness Index

The index needs to reflect several aspects of dependency trees in software projects. First, it needs to account for the amount of public API functions that are available versus those that are in use. This would provide a simple surface level usage metric. However, it is not enough, it also needs to account for all the used private functions inside the dependency. In Figure 3.1, the box represents the entire code base of a package and calls into it from its dependent. Green nodes are functions from the main package that make calls into the dependency, blue nodes are public functions that can be called by dependent packages, red nodes are private functions for internal use. In the example, we can see that a blue public function number 1, called by the green dependent number 1 uses basically the entire codebase of the package. Its call path contains 6 out of 7 internal functions. Blue node number 2 calls only 1 internal function while 3, 4 and 5 are surface level functions that do not call any internal functions.

Just looking at this example, we could state that any application or package that uses this library as a dependency and only calls functions 3, 4 or 5, like the green node number 2, does not utilize it very much and adds a lot of bloat and excess weight. However, if it calls only function 1, like green node 1, it will utilize the majority of code added, and thus, be leaner.

Furthermore, this package may have its own dependencies. In this case the leanness index needs to reflect how this package utilizes its dependencies and those dependencies' dependencies. An example can be seen in Figure 3.2. The previous example was modified

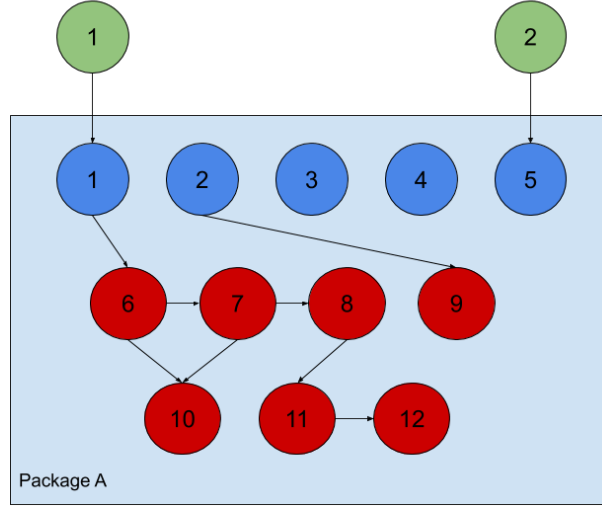


Figure 3.1: Example callgraph for package

to add an additional lower level dependency. Now function 5 uses 1 of the dependencies' public functions, which uses all private functions of that dependency. The situation changes for the leanness index too. Compared to the previous example, the index should no longer be very high if an application uses only function one of a library because more than half of the included functions are no longer in use. Using only function 5 on the other hand should have increased the leanness index.

3.1.2 Definition

In this section two versions of the leanness index are defined. In the end both will be compared and evaluated. One of them is based on lines of code per function, another is the number of functions in a callgraph.

Number of functions

Amount of functions are being measured because they reflect the underlying topology of a software system and each function can be strictly assigned to either a specific package or local code. Every call to a function could be considered as a unit of reuse. This data is easily extracted in a callgraph, by using nodes for functions and edges for function calls. This approach has issues with accuracy since function count will depend on how the developer decided to implement the package. Some developers or domains might have a large amount of very simple small functions, while another (database controller for example) might have very large ones. We calculate leanness the following way:

$$L_f = \frac{u_f}{n_f} \quad (3.1)$$

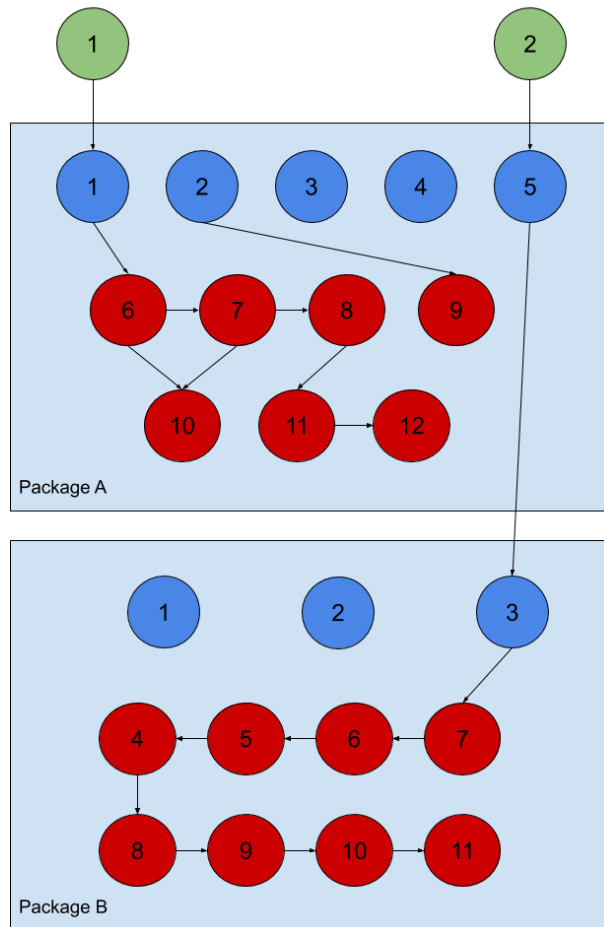


Figure 3.2: Example of an extended callgraph for package

where:

L_f = Leanness index based on number of functions

u_f = total number of functions in all included packages in the dependency tree
originating from the root application

n_f = number of function in statically analyzed dependency tree that are in the callpath

Lines of code

A possible way to resolve the issue of large and small function footprints being treated the same is to use some sort of metrics for each of the nodes as a weight instead. Doing so, should counteract excessive splintering of simple logic, and provide a better representation of the "amount of code" in any sub-graph. The easiest way to get such a weight is to sum the lines of code in each function. This approach is computationally simple and possible to

do at scale in callgraph generators. While still not perfect, it could improve on inaccuracies from the previous method at a low computational cost.

$$L_l = \frac{u_l}{n_l} \quad (3.2)$$

where:

L_l = lines of code based Leanness index

u_l = total number of lines of code in all included packages in the dependency tree
originating from the root application

n_l = number of lines code in statically analyzed dependency tree that are in the callpath

3.1.3 Leanness Computation

This chapter describes how to calculate the index for any application. This is a language and package manager agnostic method. Later sections will define the implementation for Cargo package manager. Before analyzing an application, a callgraph of every package in the ecosystem needs to be generated and number of lines of code per function saved. When analyzing an application, the following steps are performed:

1. A callgraph is generated for the source code of any project. This callgraph has to contain functions of every direct and transitive dependency. This step might be different for every programming language, but the output json file should be the same.
2. Traverse the path of every highest-level external node (entry point of the application which could either be a main function or a public API function) and label all nodes touched as "used".
3. Count every dependency node that is in the callgraph and sum all lines of code.
4. Count dependency nodes that have been labeled as used. Also sum lines of code for these nodes
5. Calculate the ratio of the two previous sums to get two indices.

This is the leanness index of an application. It represents the amount of code included by dependencies that is being used. This index would answer the first research question defined previously. An example of this calculation can be done by using Figure 3.2. Suppose that this is the entire callgraph generated by step one of the computation process. Main project has only 2 simple functions 1 and 2. It adds one dependency called Package A, and that dependency has another transitive dependency called Package B which is not explicitly stated by the developer of the main package (green). During step two of the computation, we traverse the graph by starting at green node 1, then after that is done doing the same for green node 2. This process will label nodes 1, 5, 6, 7, 8, 10, 11 and 12 from Package A. It would also label nodes 3, 4, 5, 6, 7, 8, 9, 10, 11 from Package B. Step 3 would count all nodes that come from dependencies which in this case is 23. Step 4 would count all

nodes that have been labeled as "used", which is 17. Then a ratio between used and all dependency nodes is calculated which ends up being $17/23 \approx 0.74$. The index means that 74% of included dependency functions are being used by the application

This can be compared to analyzing only public APIs. Taking the example in Figure 3.2, it could be easily said that the application uses 2 out of 5 of available APIs. That is 40%, compared to 74% when considering the entire callgraph.

3.2 Software Composition index

3.2.1 Goals of the Software Composition Index

Software composition index is used to answer RQ2. While the main purpose of leanness index is to show how much of included dependencies is actually being used disregarding the code in the main project, software composition index's main purpose is to reflect how much of the program's code comes from the developers themselves and how much comes from external dependencies. Secondary purpose of this metric is to explain and ignore packages with very low leanness index. After all, if there are two packages that have a 0.05 leanness indices due to very low utilization of their dependencies, there is a big difference between the two if one has only a small portion of its logic come from dependencies, while the other is comprised mostly of dependency logic.

3.2.2 Definition

$$D = \frac{u}{n} \quad (3.3)$$

where:

D = lines of code based Software Composition index

u = total number of lines of code in the main package's callgraph.

n = total number of lines of code in the project and its dependencies combined.

3.2.3 Composition Index Computation

Computing the Software Composition index uses the data that was generated for leanness computation. Therefore, it involves following the first 2 steps of leanness computation described in section 3.1.3. Afterwards these steps are performed:

1. Count total nodes and lines of code for all nodes in the graph.
2. Count total nodes and lines of code for all nodes that come from dependencies.
3. Calculate the ratio of the two previous sums to get two indices.

This is the software composition index. Figure 3.2 can be used as an example again. Count the total number of nodes which is 25. Then count nodes that belong to the included dependencies which would be 23. Lastly, divide the numbers and get the ratio - $23/25 \approx 0.92$. This means that 92% of the functions in the codebase for this project come from dependencies.

3.3 Utilization index

3.3.1 Goals of the Utilization Index

If enough crates are analyzed, data about how other packages use each crate can be generated. Previous questions were concerned with a crate's dependencies and analyzed how it uses them. Utilization index will analyze dependents and how other crates use the crate in question.

For example, let's say we wanted to get a utilization index for warp [25], a web server crate. If we look for this crate on crates.io and go to its dependents, we can see that there are 54 packages that depend on warp. If those 54 crates are analyzed and every function in warp labeled as in use or not, functions that are never used would be identified.

3.3.2 Definition

$$U = \frac{u}{n} \quad (3.4)$$

where:

U = lines of code based Utilization index

n = number of functions that belong to crate in question. This includes functions from its dependencies

u = number of functions that are used by at least one dependent of the crate in question.

3.3.3 Utilization Computation

Utilization index calculation process is described in detail in this section. Unlike previous metrics, this one involves several logical loops. First, data is gathered by looping over all crates in crates.io. Afterwards, each crate can be calculated by aggregating collected data. Data gathering process for each crates is as follows:

1. First step is to generate a callgraph with all dependency functions in it. This is the same step as in leanness index computation
2. Find the direct dependencies of the crate that is being analyzed.
3. For each of the direct dependencies perform the following steps:
 - a) Create a unique copy of the graph for this dependency only.
 - b) Generate a list of transitive dependencies that originate for this direct dependency.
 - c) Traverse the callgraph, starting from each public function of the dependency that is being called by the main dependent and label the nodes as "used".
 - d) Loop through all the nodes in the graph checking if they are part of the transitive dependency list and if they have not been labeled yet, label them as "unused".

- e) Store the results of the analysis in database, adding a row for each function and assigning a use counter to it, then either increment it if it was previously there or add a new one with use counter set to 1.

Once these steps are run through every package in crates.io, the actual index can be calculated. Perform the following steps to get the index for a dependency.

1. Retrieve the data previously calculated for this crate. It consists of a row for every function that results in being added to a dependent if this crate is included. This row contains a function identifier and an integer which represents in how many dependents the function is used.
2. Calculate the ratio of functions that have the use counter at more than 0 and all functions. This is the utilization index for the crate.

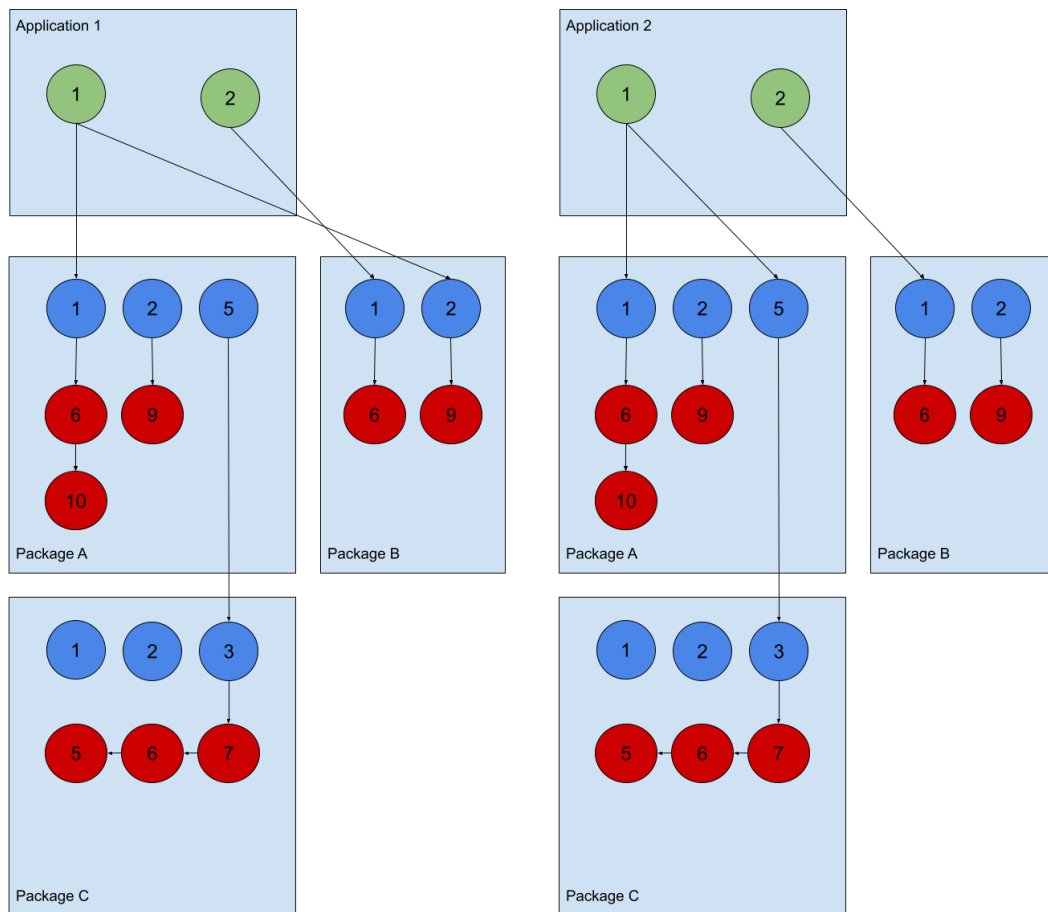


Figure 3.3: Several callgraphs containing Package A and Package B as a dependency

As an example for this calculation take Figure 3.3. There are 2 completely separate callgraphs. Each one originating from the analysis of the applications displayed at the

3. PROPOSED CODE REUSE METRICS

top. For the ease of example both **Application 1** and **Application 2** have the same exact dependencies, in most cases this would not be true. The first loop described in this section would go over these 2 graphs one a time, starting with the one for **Application 1**.

First step is to retrieve all direct dependencies, in this case that would be **Package A** and **Package B**. Then we start an inner loop for these direct dependencies, starting with **Package A**. A copy of the entire graph is created for **Package A**. Then transitive dependencies that originate from **Package A** are retrieved. In this case that would only be **Package C**. Then we start traversing the graph. Traversal only starts from public nodes of **Package A** that have an inward edge from Application 1. In this case it would be node 1 in **Package A**. After traversal, nodes 1, 6 and 10 are marked as "used". Then a loop is initiated for all the nodes in the subgraph that originates from **Package A**, in this case it would be all **Package A** nodes and nodes from its transitive dependency we retrieved earlier: **Package C**. In this loop we label all nodes that have not been labeled already as "unused". In this example, nodes 2, 5, 9 in **Package A** and all nodes in **Package C** get labeled as "unused".

Now this data is saved in the database. For every node a row is inserted since it is the first time **Package A** is encountered. Those that were used get the use counter field set to 1 and all those that were not used get the field set to 0. Afterwards, **Application 2** is analyzed. Similarly start with the first direct dependency **Package A**, traversing it, labeling, and counting nodes. After all nodes are labelled the database entries that were previously inserted are updated. Rows corresponding to used functions get their use counter field incremented. Unused nodes do not affect the database.

The output data after these two runs can be seen in Table 3.1. The entire table is meant for the analysis of **Package A**. **Package C** functions are also included in **Package A's** analysis due to the fact that it is a transitive dependency. Inspecting the data leads to 3 functions never being used. Since there are a total of 12 functions, the utilization index ends up being $9/12=0.75$. Similar procedure can be performed on **Package B**. The data resulting from the analysis would point to functions 2 and 9, out of 4 in total, not being used. Thus, the utilization index would be equal to $2/4=0.5$.

Table 3.1: Utilization analysis example table

Package	Function	UseCounter
Package A	1	2
Package A	2	0
Package A	5	1
Package A	6	2
Package A	9	0
Package A	10	2
Package C	1	0
Package C	2	0
Package C	3	1
Package C	5	1
Package C	6	1
Package C	7	1

Chapter 4

Analysis Pipeline

In order to find the sought data about the Rust package repository most crates in the crates.io registry need to be analyzed. Figure 4.1 represents the process involved from retrieving the source code to saving the extracted data into the database. Part of this process is language specific, meaning that if the same metrics were to be extracted for Java, this part would have to be implemented for it. The general purpose of this is to extract and output generic callgraph files. For Rust, the first step in this section is to retrieve the source code. Afterwards, it is compiled and a callgraph is generated. These steps were done using the callgraph generator developed by Konstantinos Triantafyllou [14]. Afterwards, callgraph files were reformatted into Json files that the algorithm accepts.

Language agnostic section is language independent. It takes in a specific format described below and outputs data into an SQL database. The algorithm can be split into two sections: one that traverses the entire graph and one that traverses subgraphs for each direct dependency. Lastly, data is saved in an SQL database.

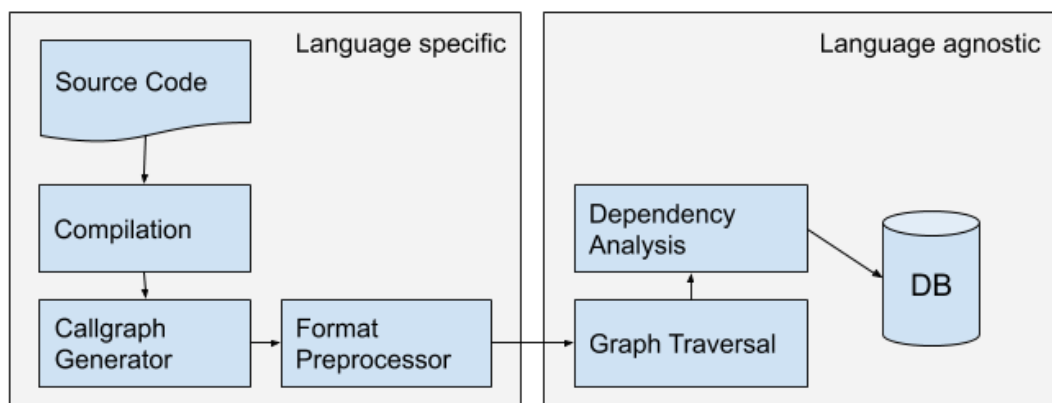


Figure 4.1: Pipeline of the metrics analysis

4.1 Retrieving the callgraph dataset

The entire crates.io registry was downloaded and a callgraph generated for every crate. For each crate, if compiled successfully, a callgraph file was created in a Json format, which defined every node and edge in the graph. Cargo.lock was also outputted which specified what versions of dependencies did the callgraph analyzer resolve to during the compilation process. The Cargo.lock and the callgraph Json files were used in the following steps.

4.2 Format Preprocessor

The purpose of this step is to bridge the gap between different file formats that come out of callgraph generators and the format that is the most efficient for the metrics extraction algorithm. During this step the graphs are turned into Json files that are later easily parsed into Rust data structs which allow for a low complexity graph traversal. Due to simplicity in handling mixed data types python was used to convert the callgraph generator's output into a format seen in Figure 4.2. The final Json file contains an array where each element in it represents a node seen in the figure. It contains several descriptive fields, such as if it's public or which package/version it belongs to. This node object also has two arrays that define edges: one for inward edges coming from its parents and one for outward edges which point to its children. These are here to speed up graph processing during the analysis. Target numbers are indices to other elements (nodes) in the array. This Json is later parsed into a Rust struct.

```
{
  "id": 0,
  "package_name": "ring",
  "package_version": "0.16.12",
  "crate_name": "ring",
  "relative_def_id": "ring[3909]::arithmetic[0]::bigint[0]::elem_exp_consttime[0]::entry[0]",
  "is_externally_visible": false,
  "num_lines": 3,
  "source_location": "/opt/rustwide/cargo-home/registry/src/github.com-1ecc6299db9ec823/ring-0.16.12/src/arithmetic/bigint.rs:960:5: 962:6",
  "inward_edges": [
    {"target": 3674},
    {"target": 1893},
    {"target": 3773},
    {"target": 3773},
    {"target": 2961},
    {"target": 991},
    {"target": 991},
    {"target": 991}
  ],
  "outward_edges": [
    {"target": 4030},
    {"target": 4030}
  ]
},
```

Figure 4.2: Callgraph data format

4.3 Graph Traversal

Next step is to investigate the graph itself and determine used and unused nodes. Starting at every public function of a crate, traverse downwards through the graph labeling every node along the way. A label is applied to every node that is visited denoting the fact that it is a possible callpath from the root crate. If traversal finds a node that has already been labeled

Algorithm 1 Graph Traversal

```

1: procedure TRAVERSEFROMNODE(graph, index)
2:   level  $\leftarrow$  0
3:   current  $\leftarrow$  index
4:   next  $\leftarrow$   $\emptyset$ 
5:   while current.len() > level do
6:     i  $\leftarrow$  current[level]
7:     node  $\leftarrow$  graph[i]
8:     if node.type = null then
9:       node.type = "used"
10:    next  $\leftarrow$  next  $\cup$  node.outward_edges
11:    end if
12:    if level + 1 = current.len() then
13:      current  $\leftarrow$  next
14:      level  $\leftarrow$  0
15:      next  $\leftarrow$   $\emptyset$ 
16:    else
17:      level  $\leftarrow$  level + 1
18:    end if
19:  end while
20: end procedure

```

the algorithm ignores it and does not investigate its children. This step helps ignore recursive callpaths and speeds up the process preventing investigation of the same tree branches.

In order to achieve a much higher efficiency, the algorithm does not scrape the graph along the edges back and forth, instead it moves downwards one level of depth at a time. In this context, level of depth is path length between the root node and the node in question. The algorithm starts with the root node's children as level 1, labels them and adds their children to a hashset of the next level. Once these are analyzed, it moves on to the next level and loop continues until the next level has no nodes in it. Since duplicate hashset insertions are not allowed this method ensures an $O(n)$ complexity of graph traversal.

Pseudo code for this procedure can be seen in Algorithm 1. This procedure is called for every public function of the main crate. Graph argument is a reference to the entire graph structure which is modified during the procedure. Index argument points to the root node in the graph. That is where the traversal algorithm starts.

4.3.1 Graph Traversal Example

For example, see the callgraph in Figure 4.3. In this example nodes 15, 16 and 14 end up not getting labeled and are considered unused. The algorithm would start at Node 1 and label it as used since it's just a public function. It would then proceed as follows:

1. Add all children of node 1 to the pool of nodes to investigate once current pool is done.

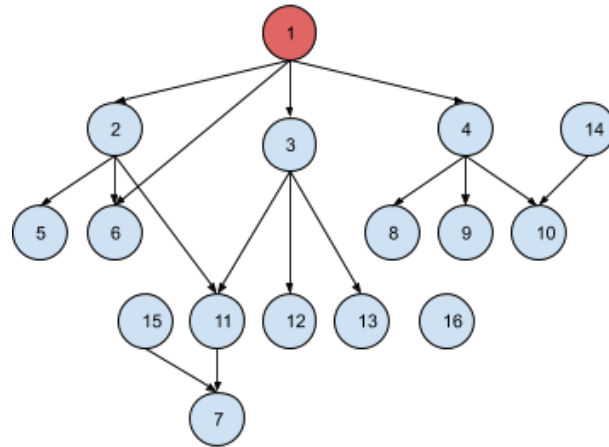


Figure 4.3: Sample callgraph

2. Move to the next node pool since first is done (only contained node 1).
3. Investigate node 2, add its children to the next node pool including 6 and 11.
4. Investigate node 6, no children to add.
5. Investigate node 3, add its children to the next node pool except 11 since it has already been added once.
6. Investigate node 4, add its children to the next node pool.
7. Current node pool investigated, moving to the next one, starting from 5. No children, moving on.
8. Investigate node 6, since it has already been labeled before, it is ignored.
9. Investigate node 11, add children to the next node pool.
10. Investigate the rest of the nodes in this pool - 12, 13, 8, 9, 10.
11. Move to the next pool, which only contains node 7 and investigate it.

4.4 Dependency Analysis

The steps above are enough to get data for RQ1 and RQ2. However, to answer RQ3 (Utilization index), each direct dependency needs to be analyzed in more depth. To explain the following algorithm several concepts need to be established. The dependencies are not analyzed separately using their respective graphs. Instead they are analyzed as part of the main graph that was used to answer RQ1 and RQ2. Dependency relations are determined by analyzing the Cargo.lock file that was an artefact of the compiler/callgraph generator step. This file contains all crates and their versions that were used in the compilation of the main

package and their relations to one another. A graph could be constructed using it, where the root node is the main package.

The algorithm, after performing the previous steps, generates a list of direct dependencies from the Cargo.lock file. Then, for each of those dependencies, it finds other dependencies that originate from them. Meaning that, by adding this direct dependency, they were indirectly added to the dependency graph. These indirect dependencies might appear more than once for each direct dependency and that is adequate for this purpose. For the purpose of this explanation look at Figure 4.4. In this figure, green nodes are functions in the original main package, red ones are public functions in the direct dependency that is being analyzed in the current iteration, blue nodes are private functions of the direct dependency that is being analyzed, yellow nodes are public functions of a transitive dependency, purple nodes are private functions of the transitive dependency.

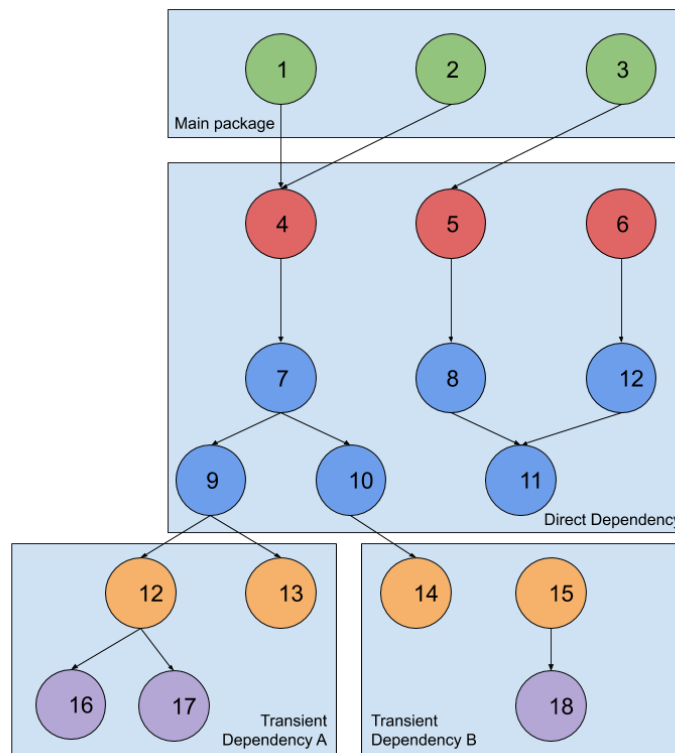


Figure 4.4: Sample callgraph of dependency analysis

After finding these transitive dependencies the algorithm performs a similar graph traversal to the one described in section above. Starting from each public function of the dependency, in the example these are red nodes. The algorithm determines if it is being used by the main package or not, by checking its inward edges. In the example nodes 4 and 5 are used. If it is being called from the main package, then a graph traversal algorithm is initiated with that public function node as root. All nodes that are traversed, are labeled as "used"

by the algorithm. Once all public functions of the dependency that are called from the main package have been traversed, the algorithm performs the same traversal on public functions that are not called by the main package. In the example it starts at node 6. This time assigning a label "unused". If traversal encounters a node that has already been labeled "used" (in this example it would occur at node 11) it is ignored and its children are not traversed.

This method is guaranteed to find nodes that are used by the main package, but it still does not cover unused nodes that are transitive dependencies. In the Figure 4.4 it would be node 15 which is an available public API method that is not used by its dependent package. To solve this, the algorithm loops through every node in the graph and if it belongs to any of the transitive dependencies that were added by the direct dependency that is being analyzed, it assigns an "unused" label to each of these, finalizing the graph.

4.5 Implementation

Majority of the code was implemented in Rust, this includes the algorithms and call graph generation. Format preprocessor was the only thing that was implemented in python. Due to varying datatypes in Json arrays it was much simpler and modular to use something like python that can change the format into a strict data structure.

The entire application was packaged into a docker container and run on a server rack at TU Delft. The container itself did not utilize all cores and memory available on such a rack since it was not needed for this dataset. The algorithm itself has low complexity and crates.io is not an extremely large registry. Total run took 2 days. It could easily be parallelized if for example npm (with its over 1.4 million packages) was to be analyzed with this method.

Final results were saved in an SQLite database file which was 49.5GB in size. 27295 packages were successfully analyzed, while 6320 could not be compiled for analysis.

The crate analysis implementation can be found open sourced on github [2].

Chapter 5

Results

In this chapter results of the thesis are presented and analyzed.

5.1 RQ1: Leanness index

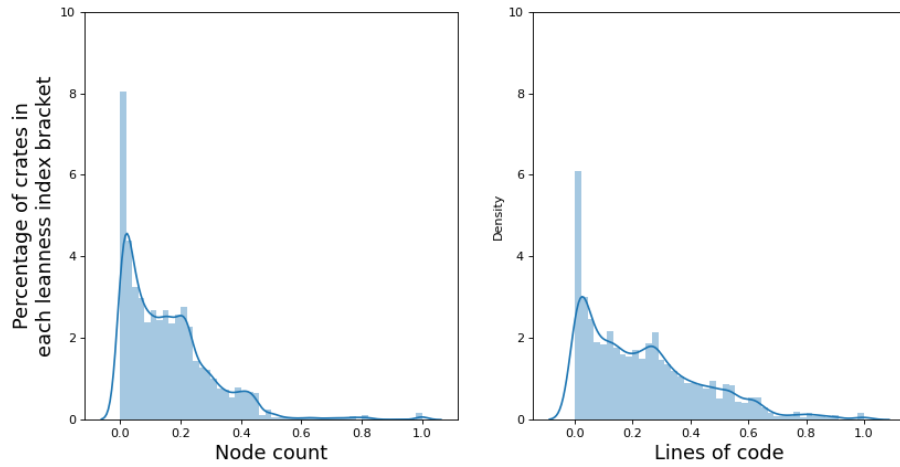


Figure 5.1: Graph of leanness index distribution using two different methods

5.1.1 Lines of Code versus Node Count

The graphs in this section were generated the way it was described in leanness index formulas 3.2 and 3.1. Additional post processing was done. Crates with no dependencies included at all, have been removed. Only crates with dependencies are relevant. Furthermore, crates that use exactly zero lines of code from their dependencies have also been removed. It is justified by the fact that the callgraph generator has faults recognizing certain aspects of callgraphs and that this metric is mostly concerned with code reuse and not whether certain

5. RESULTS

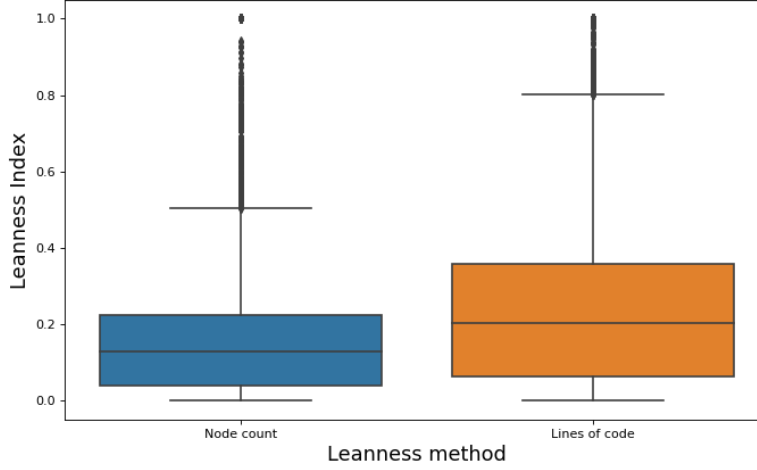


Figure 5.2: Box plots for leanness indices calculated using nodes and lines of code

dependencies are unused at all. Figure 5.1 shows the distribution of the leanness indices that have been calculated using two different methods. Box plots of the same data is shown in Figure 5.2.

Both methods result in a large amount of crates that end up in the bin close to zero. When measured using node count, 28% of crates have used less than 5% of their included dependencies' functions. There is a slight difference when comparing against counting lines of code (LOC): 21.7% of crates use less than 5% of LOC from dependencies. Similarly, when counting nodes 79% of crates use less than 25% of functions while lines of code based metric shows that 57.6% use the same share of lines of code. Furthermore, 95th percentile for node count is a rather low 42.5%, while lines of code count reaches 62.4%. Lastly, mean for node count is 0.16, with a median of 0.13, while LOC count mean is 0.24, with a median of 0.21. Standard deviation 0.14 and 0.2 respectively.

From the numbers and as can be seen in the graphs, LOC method has a much more even spread than just counting nodes. There are several reasons for this: having many simple superficial functions that are not needed, having majority of the functionality in several core functions and possibly other reasons. Looking at this comparison it could be stated that using LOC method is more representative of what the leanness index is trying to show. Using LOC as a weight for each function makes it so not all functions are treated equally which is correct, since in reality all functions are not equally important. Therefore, any further results will be presented using only LOC count.

One more thing to consider regarding using LOC count is that LOC is not available for all nodes in the generated graphs. Functions that originate from C code appear in the graph with their lines of code set to 0. Figure 5.3 shows how many functions from dependencies contain LOC information. This graph shows the distribution of the ratio of each analyzed package's count of functions with LOC information that come from dependencies and all

functions from the same package that come from dependencies. 95% of all crates have at least 67.16% of dependency functions with LOC information. This lack of precision is insignificant when analyzing the entire repository. However, for qualitative analysis of each package this could prove useful in determining bloated dependencies.

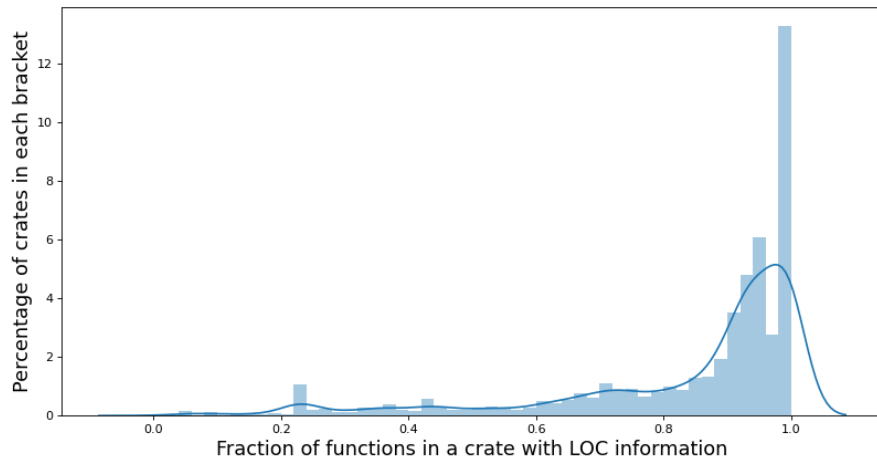


Figure 5.3: Share of functions from package dependencies that contain Lines of Code information

5.1.2 Count Public APIs vs Entire Callgraph

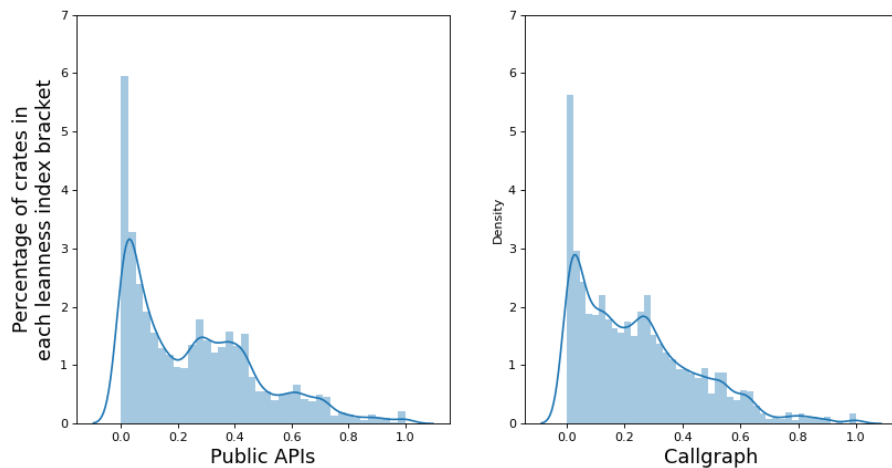


Figure 5.4: Graphs comparing leanness index when calculating ratio of used versus unused public functions of a dependency versus analyzing its entire callgraph

5. RESULTS

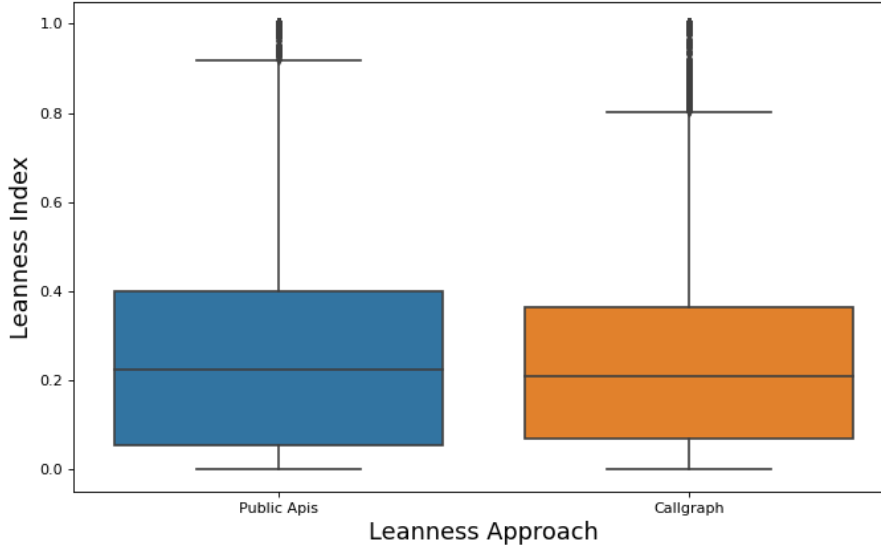


Figure 5.5: Box plots leanness index when calculating ratio of used versus unused public function LOC of a dependency versus analyzing its entire callgraph

This section presents and describes the difference between calculating code reuse with the leanness index which uses the entire callgraph of a dependency and counting only its publicly available APIs. Callgraph method follows the definition described in section 3.1.2 under the leanness index. When using the public API method, all publicly available functions that have been used at least once are considered as used. All other publicly available functions are considered unused. The ratio of used ones and all publicly available functions is what the leanness index was compared to. This comparison is drawn in a graph in Figure 5.4 and boxplot in Figure 5.5. The dataset and data trimming is the same as shown in the leanness index graph previously. It is obvious that both graphs are very similar. However, this is most likely due to comparing distributions of the entire registry of crates.io and not individual packages, since a callgraph based method can change the leanness index both ways: making it more or less lean. Over a large enough dataset this simply evens out. Which means that replacing the method does not make much of a difference when analyzing the whole dependency network.

However, the data can still be very valuable when looking at individual packages instead. Figure 5.6 was produced by calculating the difference in leanness index between the two methods for each package and plotting its distribution. There is still a large amount of crates with a difference close to 0, but there is also a significant amount where this more accurate information provides meaning. The mean value of absolute difference of all crates in cargo is 0.08. Also in 30% of all packages the difference is more than 10%.

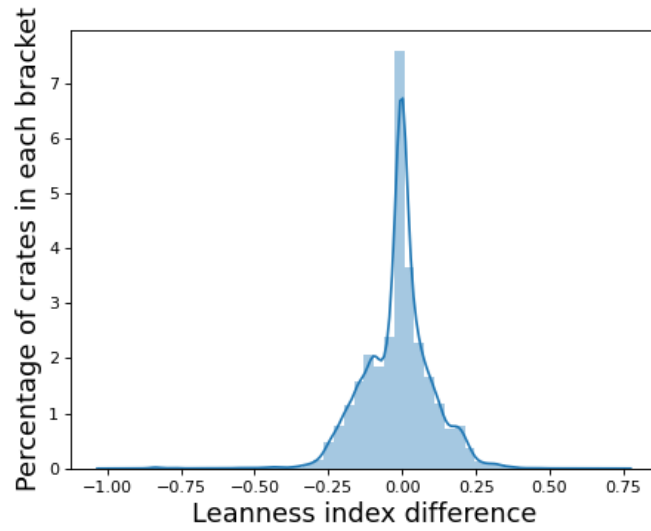


Figure 5.6: Graph of leanness index distribution comparison to public API usage method

5.2 RQ2: Software Composition index

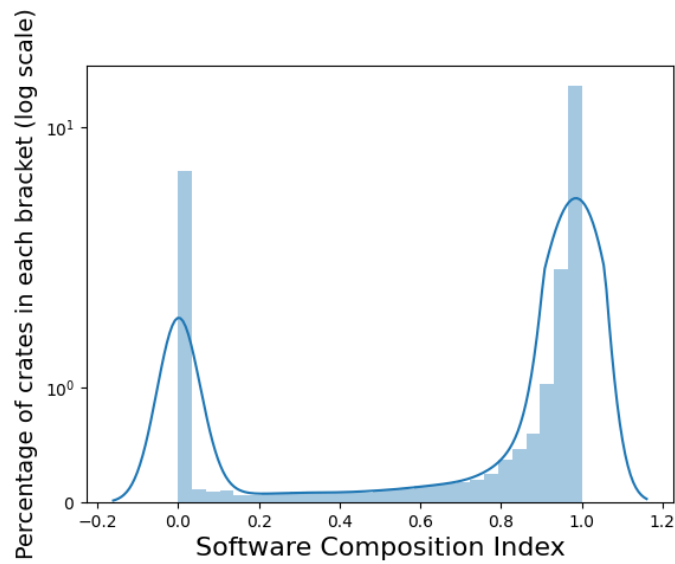


Figure 5.7: Graph of software composition index in log scale.

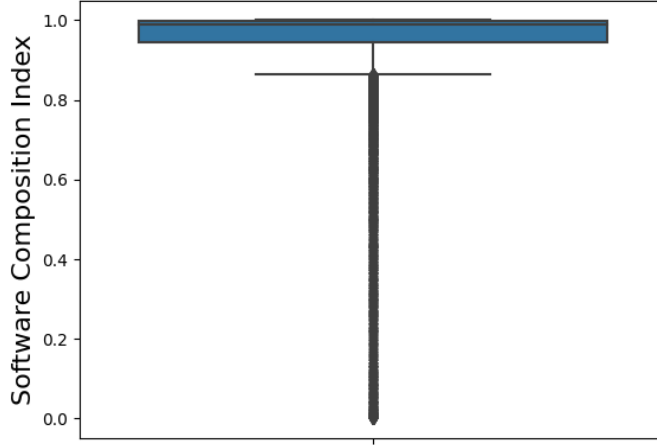


Figure 5.8: Box plot of software composition index.

Software composition index aims to show how much of the code in a software project comes from external dependencies as opposed to local code. The distribution of this metric for the entire crates.io repository is visible in Figure 5.7 and 5.8. The y axis of the graph uses a log scale. There is a significant amount of crates that have no dependency code - 14%. This is not an error like it was when analyzing the leanness. Edges of a graph are not part of the calculation for this plot. It only counts the existence of nodes in the final compiled file. Thus, the amount of zeroes is accurate. This is due to the fact that there are quite a few packages that just do not include any dependencies. However, once dependencies are added to a crate they overwhelm the code of the project itself in most cases. The mean value for non-zero dependency crates is 0.91, so on average crates that include dependencies have 91% of their compiled lines of code come from dependencies. This was the reason to represent the data using a log scale. Data is also visualized as a box plot in Figure 5.8. Data used for the box plot does not include crates that have exactly zero dependency LOC. It is not relevant when showing the distribution of dependency vs local LOC. The plot clearly shows how almost the entire dataset has a very high software composition index, median being 0.988%.

5.3 RQ3: Utilization index

Utilization index distribution can be seen in Figure 5.9. About half of the packages have a very low utilization: 57% of all crates have only 10% of their callgraphs used by other crates in crates.io. Furthermore in the case of 95% of crates, 71% of their callgraphs are never used. The mean value for the utilization index is 0.12, while median is a much lower 0.055.

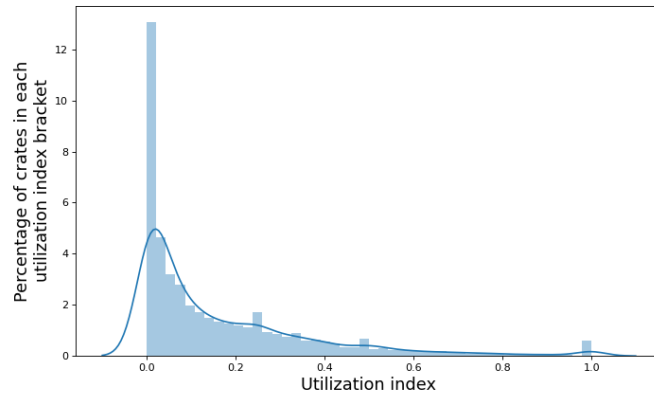


Figure 5.9: Graph of utilization index distribution

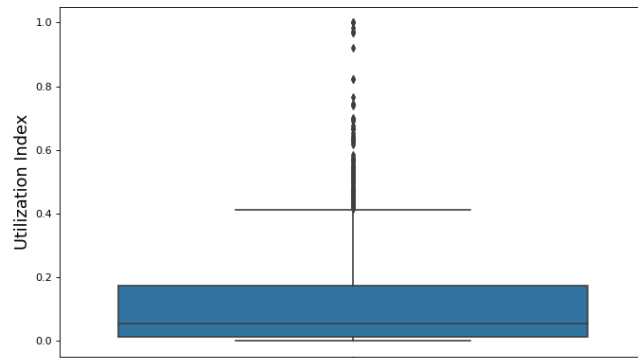


Figure 5.10: Box plot of utilization index distribution

However, this is only representative of the usage by other crates.io dependents. Some of these crates are intended more for use in final applications or non library Rust projects. For a better result, this metric could use a larger study of open source Rust applications.

5.4 Qualitative example of several packages

To highlight some features of these metrics that are not visible when analyzing large datasets, several arbitrary crates were chosen to analyze in depth using the generated data. The chosen packages are pathfinder and rodio.

5.4.1 Pathfinder

Pathfinder is a crate that can be used to generate images with large amounts of objects or nodes. The version analyzed here is the latest at the time - 0.6.5. This crate has 5 direct dependencies included in its Cargo.toml file. First piece of data is the software composition index. A pie chart was chosen to visualize the share of local code versus dependency code. It can be seen in Figure 5.11. Pathfinder seems to consist primarily of dependency code, since 98.1% of LOC comes from crates.io. This 98.1% can be further split into different dependencies that are included. This visualization can be seen in Figure 5.12. Lastly, a table of leanness indices per direct dependency is presented. This data comes as an artefact of utilization index calculation and can be seen in table 5.1. Leanness indices range from 0.11 in case of image crate to 1.0 with pythagoras. Low leanness of an image crate is to be expected since such a crate would contain a large amount of utility functions for image manipulation which are not all needed for this dependent. However, it could indicate that the image crate might benefit from being split into smaller subcrates instead. Pythagoras on the other hand is a very minimal trivial crate that only implements functions for calculating pythagoras theorem. It could be argued that such crate is unnecessary and should be reimplemented by the developer. If the developer chose to remove the image crate and reimplement it themselves the leanness index of Pathfinder would increase from 0.15 to 0.59.

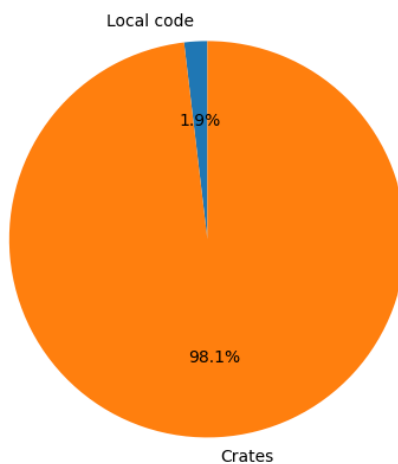


Figure 5.11: Dependency use in Pathfinder - 0.6.5

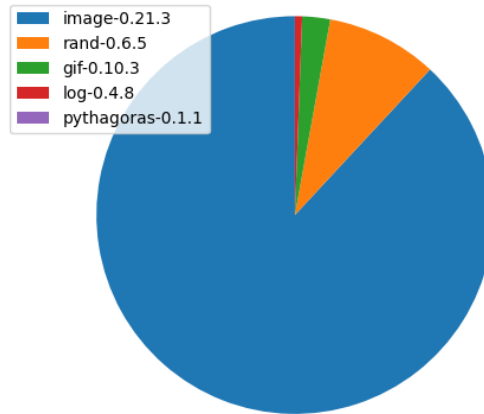


Figure 5.12: Share of different dependencies in Pathfinder 0.6.5

Table 5.1: Leanness indices per dependency in Pathfinder 0.6.5

Direct Dependency	Leanness	Functions with LOC
gif	0.36	0.99
image	0.11	0.92
log	0.23	1.0
pythagoras	1.0	1.0
rand	0.48	0.62

5.4.2 Rodio

Rodio is an audio playback library with over 200,000 downloads. The version analyzed here was 0.10.0. This crate has 6 direct dependencies in its Cargo.toml. Similarly to pathfinder, software composition index can be seen in Figure 5.13. It is clear that a large majority of code in Rodio, like in pathfinder, comes from external dependencies. That 96.4% can be split further to find the distribution of origin for each of the direct crates. Keep in mind that these are only direct dependencies and not transitive ones. Transitive ones that are added as a result of direct dependencies are included in the calculation. The result is in Figure 5.14. A table of leanness indices per dependency can be see in Figure 5.2. One interesting number in this table is minimp3's share of functions with LOC. It seems that less than half of all

5. RESULTS

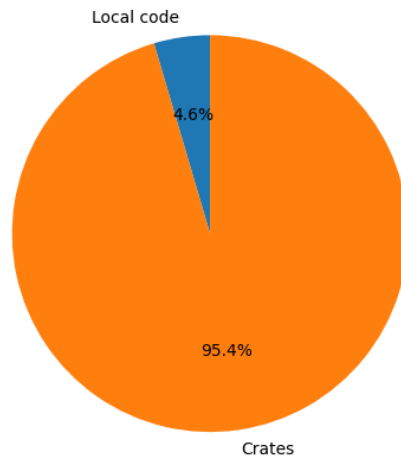


Figure 5.13: Dependency use in Rodio - 0.10.0

functions in this crate had LOC information due to the fact that it provides Rust bindings to a C minimp3 library. Thus, it is hard to judge the provided leanness index. Another low leanness index library is cpal. It provides low level functions for handling audio input and output. It is a rather large library with over 32,000 lines of code with all of its dependencies, since it allows playback on many different hosts and formats. Only 763 lines of code are in the callpath originating from rodio. It is hard to judge its usefulness for rodio, but this could point to a consideration for the developers to replace or reimplement required functions. If the developer chose to remove the image crate and reimplement it themselves the leanness index of Rodio would increase from 0.15 to 0.57.

Table 5.2: Leanness indices per dependency in Rodio 0.10.0

Direct Dependency	Leanness	Functions with LOC
claxon	0.68	0.99
cpal	0.02	0.77
hound	0.36	0.98
lazy_static	0.5	0.85
newton	0.58	0.99
minimp3	0.09	0.47

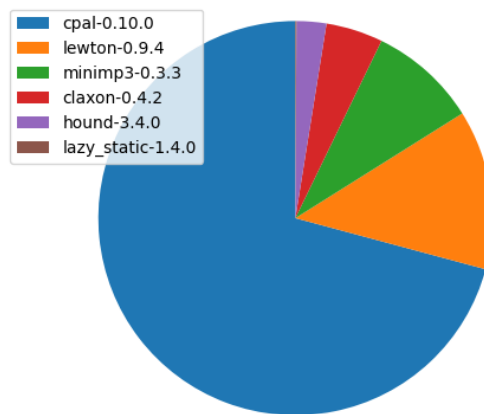


Figure 5.14: Share of different dependencies in Rodio 0.10.0

Chapter 6

Discussion

This chapter discusses results and their implications for future researchers and practitioners by revisiting research questions and threats to validity.

6.1 RQ1: Leanness Index

Calculating the leanness index using the entire callgraph of an application was supposed to be an improvement over traditional methods of only looking at the publicly available APIs in packages. Reviewing the results, it could be said that this approach proved to be situationally better. Analyzing the index distribution over the entire package registry showed no significant difference due to deviation in index appearing as both positive and negative. However, it seems to have a clear difference in a significant amount of cases when analyzing it per crate basis. The algorithm itself is not computationally expensive and as a result it could be stated that analyzing the whole callgraph, if possible, should be the default approach when trying to investigate how packages use their dependencies.

General conclusion of the leanness index analysis was that most developers do not utilize majority of the code that they include. This does not necessarily mean that it is a bad practice, it is always a case by case basis. Sometimes the amount of code that is used (even though majority of included lines are not) is just not worth the effort. For example, if the included package contains type definitions for some web API, like Amazon Web Service, a developer will highly likely not use a very large chunk of the code that comes with such a crate. However, re-implementing it is very tedious work and if in the future the developer would like to start using additional API endpoints they would have to perform additional work again, and keep it up to date in case things change. Including such a package and forgetting it is the most maintainable approach. In other cases it is better to have highly peer reviewed and secure functions instead of relying on local development team replicating such process. But having reuse data more available and visible to developers could improve dependency hygiene and maintainability. An example of a case where removal of a dependency is the best course of action would be if several simple date parsing functions are used, but the library itself has thousands of functions for different types of date manipulation. If the developer knows that their project does not need extensive date parsing and only has a

few use cases for it, re-implementation would have strong arguments for consideration.

Seeing such a large amount of crates with a very low leanness, could potentially point to developers not considering or understanding what happens to their project landscape. Currently most package managers, including cargo, obfuscate processes that are being performed in the background. Adding a single line of code that defines a version range for a dependency, ends in large amounts of code being downloaded, compiled and added to a possible very small project. Some of this code comes from transitive dependencies and are thus even more hidden. As a result it is highly probable that not all included code is reviewed and verified. It could lead to potential security, performance or maintainability issues.

Using callgraphs to analyze these issues for specific packages proved to be a more accurate approach than looking at public API usage or simply investigating the inclusion in the metadata files. With this approach concrete callpath vulnerabilities becomes identifiable and having proper dependency hygiene easier to achieve. In this case, dependency hygiene is understanding what code is included in a project, how well maintained it is, how much of it is being used by the developer's project, how hard it would be to re-implement it and having a general understanding of the dependency landscape instead of blindly adding new ones to the project every time a new function is needed.

6.2 RQ2: Software Composition Index

Software Composition Index was the simplest one to calculate but provided the most shocking results. As can be seen in graph 5.7, in almost every case, when a dependency is added to a project, majority of the lines of code in that project come from external sources and not the project itself. General trend towards this was expected, but the numbers are more extreme than the hypotheses. The conclusion here is that programmed logic present in a dependency network is highly splintered. Packages tend to include dependencies which contain large amounts of code and development time, adding comparatively small amount of logic on top. Such interconnectedness comes with both benefits and downsides. First of all, due to ease of use and no requirement to implement most logic, developers' ideas can be implemented and iterated on, at a much faster pace than if majority of the code had to be done from scratch.

However, it can also cause problems like single point of failure. When interconnectedness is high, events such as leftpad incident cause a lot of damage [8]. Furthermore, bad implementations of popular packages could spread performance issues across many projects. However, this sort of issue can be alleviated by keeping the sources of available packages open (which is the case in Rust). The community in most cases is very eager to find and solve these problems, so any significantly popular package would end up under the scrutiny of many developers.

The index by itself does not provide a lot of valuable information to the developer. However, when used in conjunction with the leanness index could provide hints on when to take action. For example if the leanness of a project is very low, it might pale in comparison to how much dependency code there is to begin with. When the leanness index is 0.1% it

would be much more concerning if a project has a software composition index of 90% as opposed to 10%. The later would point to extreme bloat of unused code in a project which might take up a lot of space when compiled into binaries or inside a container.

6.3 RQ3: Utilization Index

The goal of utilization index was to quantify how developers use each crate. The empirical data showed that most of the packages are not utilized much, indicating that it might be a good idea to splinter its functionality into more packages. Many of the most popular crates, like `rand`, already do this. However, this index does not provide any good insight into how this could be splintered. To do so, a deeper analysis and graph clustering could be employed. When clustering a graph, utilization of different parts of code could be categorized by its popularity. Splintering based on that result could highly reduce dependents' compiled code size and maintainability.

6.4 Threats to Validity

There are two main threats to the empirical data gathered and presented before. Firstly, the callgraph generator is not perfect. There were packages with 0% of their code used. This occurs when structs defined in a package gets initialized in the dependent. The callgraph generator does not pick it up and it appears as unused in the graph. Improving on this would provide more accurate results.

Furthermore, the dataset used in the thesis could be enhanced if non package projects get included. Many of the dependencies like webserver, visual algorithms and others are meant to be used in user facing applications which do not end up on crates.io. A great approach would be to perform an analysis of the most popular Rust open source projects on github and combining the data with crates.io output. Another approach would be to categorize all the crates on crates.io based on their intended functionality. Data structure packs, API bindings, threading, webserver and many other could be analyzed separately to better reflect developer intentions.

Chapter 7

Related Work

Dependency networks and APIs are well explored topics in computer science. This chapter discusses latest research in software dependency networks. Afterwards, API usage will be investigated, in particular finding out how much of the included code actually gets used by dependents. And lastly, it will look into research covering proposed metrics in software maintainability.

7.1 Dependency Networks

A. Decan et al. [6] performed a recent study on security vulnerabilities in the npm dependency network. They have analyzed all known vulnerabilities in the history of npm using snyk.io dataset and extracted several statistics. They found that the number of vulnerabilities and corresponding distinct packages has been steadily increasing. In 2015 there were around 100 known cases and in 2018, close to 400. They have also found that *three out of four vulnerable packages have more than 90% of their releases affected by the vulnerability at discovery time*. Furthermore, it seems that it takes most vulnerabilities more than 28 months to discover them, On the other hand most of them are fixed soon after discovery. The biggest issue seems to be that more than half of vulnerable package dependents take a long time to apply the fixed version of the update even after it is fixed.

Another paper by A. Decan et al. [7] investigates evolution of various package managers include crates.io. Various metrics are provided like, growth over time (Cargo had a linear growth for both metrics: number of packages and number of dependencies) or package update frequency (45% of packages constitute to 80% of all updates). However the other two research questions are more relevant to this thesis. The paper looked into which extent do packages depend on other packages and how prevalent are transitive dependencies. They find that regardless of ecosystem, large majority of packages are connected (have a dependency or a dependent), this number varies between 60% and 80%. The transitive dependency analysis showed that while most packages have few dependencies, they have a much higher number of transitive dependencies. In Cargo for example, half of dependent packages have at least 41 transitive dependencies while having a median of 2 direct dependencies. This paper also proposes a similar metric to the one covered in this thesis. They

try to evaluate a reusability index over the entire package repository over time. They define the index at any point in time as the maximal value of n such that there exist n required packages in the ecosystem, and at the same time, having at least n dependent packages.

Similar results were found by R. Kikas et al. [12]. They have also looked into a number of dependencies over time and growth of the network itself in npm, RubyGems and Crates.io ecosystems. However, they have additionally looked into a network's vulnerability to removal of a single project. They define it as number of nodes (packages) that are affected by a removal of one packages. Each of their researched ecosystems have packages which if removed would affect up to 30% of other packages and applications. This also indicates the speed of bug spreads if they are introduced in these popular packages.

A paper by Rabe Abdalkareem et al. [1] looks into why developers use trivial packages in their projects. The paper notes that a serious discussion about the values of trivial packages has started since the occurrence of the left-pad incident [8]. It then tries to investigate the causes of such use. First of all they quantitatively define what is a trivial package, which was done by a developer survey where they were asked what indicates if a package is trivial and to rate a list of package examples. The survey only involved 22 developers of various levels of experience which is rather concerning. Nevertheless, they concluded that packages that are less than 35 lines of code and have a less than 10 McCabe's complexity. The paper provides various empirical metrics on npm dependency network's trivial packages and ultimately answers the main question. Two main reasons why developers use trivial packages are that they improve productivity and that they provide well implemented and tested code. However, further investigation by the researchers found that only 45.2% of trivial packages have any tests at all.

7.2 API Research

Open source libraries and APIs have been researched extensively in many ways, their benefits, their downsides, various empirical data and vulnerabilities. Here are a few of the more interesting related works in the field.

General reasons for why people start using libraries instead of their own already existing implementations and vice versa is researched in [27]. It would seem that in majority of cases where developers replace their own code with an existing library is because they were not aware of the existence of such a library to begin with. Once they find out about it and if it is a well tested and documented library, they start using it instead. On the other hand, developers also re-implement some libraries with their own code. This part of the paper is more interesting for this project because according to their research, one of the most common answers was that the library was too heavy for the light functionality the developer needed, or that the library was too hard to use.

API usage trends over time in Java project was analyzed by [18]. They have created a tool that mines a few hundred open source project repositories over time and provides usage trend information on imports that they use. The tool can be used to see if the package a developer wants to use has a decreasing usage trend, signaling that other projects are phasing it out for some reason. The authors also suggest that it can be used by developers to improve

their APIs. It is a somewhat similar research to this thesis, however they have performed the analysis by only looking at import statements in 200 different projects. Such an approach provides a rather superficial view of APIs usage and does not explain it. Furthermore, 200 projects is not a very representative dataset. This thesis will implement find trend data for each public function of API, for each version of API and over the entire crates.io repository.

A more recent study on java APIs was performed by C. Lima and A. Hora [16]. This paper investigates a large number of characteristics of API usage in the most popular java open source projects on github.com. First set of characteristics was regarding code of the chosen set of APIs. These include size, complexity, legibility, documentation and others. Second set was about the evolutionary properties of APIs. These were changeability, contribution and stability. Lastly they investigated client adoption of APIs. They have chosen 3 groups of APIs to investigate separately: top 10% of the most used ones, bottom 10% and a randomly selected set from the remaining ones called 'ordinary'. The APIs were furthermore grouped into java for those starting with 'java*', android for those starting with 'android.*' and others. The findings generally point to a significant statistical difference in most metrics between different both API categories (popular, ordinary) and ecosystems (java, android, others). This is an important finding to consider for the results of this thesis.

7.3 Unused API code analysis

In this section I will look at papers that have researched similar aspects of API usage, which is - unused code. For each of them I will look into how they are different from the work in this thesis.

Nicolas Harran et al.[9] have performed an analysis on 2.3 million Maven dependencies. The analysis was performed by choosing 99 most popular libraries in the Maven ecosystem, then picking 865,560 open source client programs that used these dependencies and looking at how much of the included dependencies do these clients actually use. According to their findings 43.5% of included dependencies are never used in bytecode. Furthermore they have found that on average, most of the API interfaces are not used. Average amount of unused interfaces from dependencies (ignoring the cases where they are not used at all) is 71.84%. In the end they provide a core index value for a median API. They propose that on average 17% of used types are enough to serve 83% of clients. The key problem with this paper is that it only considers top level API interfaces and their usages, ignoring the entire callgraph underneath it and inclusion of other dependencies. This thesis will expand on this greatly by quantifying reuse in a way that might help find out what amount of reuse is healthy.

Dong Qiu et al. [19] have done another large scale API usage analysis on the Maven ecosystem. The paper has a vast amount of empirical data. Some of the more relevant points to this thesis show similar results in terms unused packages: 15.3% of the classes, 41.2% of the methods and 41.6% of the fields are never used by any project in their dataset. Furthermore, the paper analyzes API coverage (whether a public method or class has been used at least once) for the core API (official java libraries). The results show that about 40% of all core classes have not been sufficiently used.

Anand Ashok Sawant et al. [22] have introduced a method for extracting fine grained API usage information using their new tool called fine-GRAPE. The information that they have extracted includes an analysis on how many API features have never been used by any client which is about 20% on average. The perspective of this statistic is rather different from what this thesis aims to investigate. The aim in the paper was looking for all clients that use API and looking for features that have not been used, while this thesis is more focused on analyzing the included dependencies of a project and see how many of the included features are actually used and how much is dead code.

7.4 Software metrics

In this section several papers about software metrics will be reviewed.

One of the first papers in the field of software metrics was written by S. R. Chidamber et al. [3] in 1994. They have set out to define, implement and collect a set of 6 metrics for object oriented software design. While this thesis, is focused on more general software metrics that is not pinned down by its design philosophy, it is good to review the groundwork that was laid. First proposed metric called Weighted Methods per Class. It is defined as a sum of complexities for every method in a class. Complexity itself is not defined. The purpose is to predict how much time and effort is required to develop and maintain this class. Another proposed metric is Depth of Inheritance Tree which is defined as the length of a path from a class to the root of its inheritance tree. This metric would identify classes that are deeply inherited and thus have too many methods. Such classes' behaviour would be too hard to predict. Third proposed metric was Number of Children, which simply counts how many classes inherit a specific class. This would identify how reused a class is and in case of high reuse it may need additional testing and thought put into it. One more metric is called Coupling between Object Classes and is defined as a number of other classes the class is coupled to. Coupling is defined as acting on methods or properties of another class. In classes where this metric is high would not be very modular, harder to encapsulate and also test. The fifth metric proposed by the paper is Response for a Class and it is defined as a length of a set of methods that can potentially be executed in response to a message received by an object of that class. Large number of such methods would make the class much harder to test and also increase its complexity. Last metric proposed in the paper is Lack of Cohesion in Methods. It is defined as a count of number of method pairs whose similarity is zero minus the count of method pairs that are not zero. Large amount of similar methods points to being able to have better encapsulation while lack of cohesion would point to the class needing to be split into separate classes. The metrics were chosen and designed to have a strong theoretical background explained in great depth in the paper and also intuitive for developers to understand. With these available for every class in a software project it could be used to pick areas to refactor in order to improve maintainability of the entire project.

Danail Hristov et al. [11] proposes a reusability metric framework for component-based software development. In this paper several reusability characteristics are distilled: availability, documentation, complexity, quality, maintainability, adaptability, reuse and price.

Each of these characteristics are defined in great detail, for example it is noted that reuse can be measured by frequency and amount of reuse. One of the conclusions in the paper was that software reuse is a context specific characteristic and that different methods need to be applied in order to provide valuable information in different application context, which is reflected well in this thesis as well. Computed metrics can point to varying consequences depending on context.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In conclusion, this thesis introduced 3 metrics that quantify code reuse of dependencies in crates.io, compared the results of these callgraph based metrics to the previous public API analysis, and applied the metrics for the entire crates.io package repository of Rust.

Leanness index provided information on how dependencies are used. Based on previous studies, the data gathered showed that usually only a small amount of code gets used from the included dependencies. 57.6% of all crates use less than 25% of LOC they include through dependencies. During the analysis of this data, lines of code versus node count was examined. While they share similar trends, two versions of the metric proved different. Lines of code based leanness metric had a more uniform spread of data over the entire crates.io.

Software Composition index enriched the leanness metric by providing supplemental information. It is a critical factor for developers when deciding if they need to reconsider their dependency use in the project. Empirical analysis of this index showed a very big share of crates that are overwhelmed by code from other dependencies. On average, if a dependency is included in a crate, 91% of its code comes from the dependencies.

Lastly, the utilization index showed how much each package gets utilized by its dependents. This metric could be used by crate developers to improve their APIs and possibly split code into several different packages to improve their dependents leanness and compiled code size. The analysis of the entire crates.io showed that 57% of all packages get utilized by only 10% of their code. These results could be skewed by the fact that only libraries were analyzed. Including open source applications would most likely improve the results.

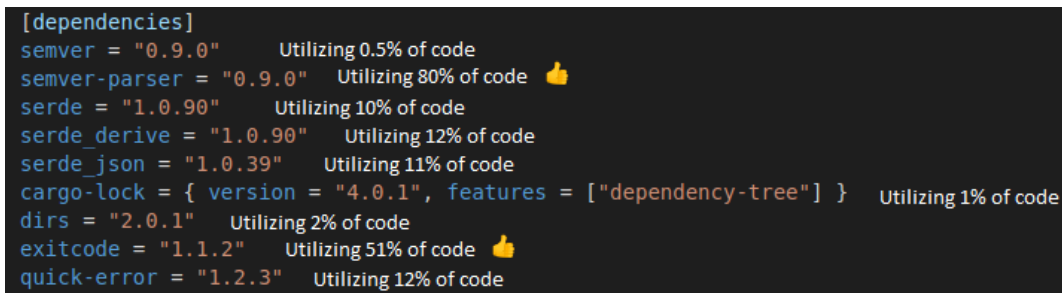
8.2 Future work

Future work regarding these proposed metrics would include two aspects. First is getting better data using the same concept. Primarily this would involve improving the callgraph generator. Current shortcoming of no edges for struct initializations can skew the data quite

8. CONCLUSIONS AND FUTURE WORK

drastically for certain packages that rely on large amount of different structs. Furthermore, additional rules in metric calculations could be included. For example, many packages are split into several sub packages that include each other. Such a package feature could be detected and handled differently. Lastly, improving the dataset by including open source applications would greatly help certain crates get a higher utilization index.

A different type of future work is to create tools and utilities that use these metrics in helpful ways. One of them could be to develop an IDE plugin that would generate this data on the fly. Providing a general software composition index and how each included dependency contributes to such a number in Cargo.toml file could help developers decide where to trim bloat. An example can be seen in Figure 8.1. The figure shows a section of Cargo.toml file in a Rust project. Included dependencies and their versions are defined here. Plugin shows a simple informational text to the right of each dependency. This example only has leanness information available, but it could also include others like total amount of code added by each dependency. Similar approach could be used to create tools that help package maintainers splinter their packages into smaller chunks, so that their users can include only what they need. Once simple implementations are created, they need to be investigated for their usability. A user study could be performed to see if developers find the metrics useful, whether the tools themselves visualize them well and make it easy to understand and if not, how it could be improved.

The image shows a screenshot of an IDE plugin displaying a list of dependencies from a Cargo.toml file. Each dependency is listed with its name, version, and a leanness index (percentage of code utilized). Some entries also include a thumbs-up emoji. The dependencies are: semver (0.9.0, 0.5% code), semver-parser (0.9.0, 80% code), serde (1.0.90, 10% code), serde_derive (1.0.90, 12% code), serde_json (1.0.39, 11% code), cargo-lock (4.0.1, 1% code), dirs (2.0.1, 2% code), exitcode (1.1.2, 51% code), and quick-error (1.2.3, 12% code).

```
[dependencies]
semver = "0.9.0"      Utilizing 0.5% of code
semver-parser = "0.9.0"  Utilizing 80% of code 🍑
serde = "1.0.90"      Utilizing 10% of code
serde_derive = "1.0.90" Utilizing 12% of code
serde_json = "1.0.39"  Utilizing 11% of code
cargo-lock = { version = "4.0.1", features = ["dependency-tree"] } Utilizing 1% of code
dirs = "2.0.1"        Utilizing 2% of code
exitcode = "1.1.2"    Utilizing 51% of code 🍑
quick-error = "1.2.3" Utilizing 12% of code
```

Figure 8.1: IDE plugin example to visualize leanness index

Bibliography

- [1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad 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*, pages 385–395, 2017.
- [2] Algirdyz/crate-analyzer, 2020. <https://github.com/Algirdyz/crate-analyzer>.
- [3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [4] crates.io: Rust Package Registry, 2020. <https://crates.io/>.
- [5] Barthélemy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. *SIGPLAN Not.*, 43(10):313–328, October 2008. ISSN 0362-1340. doi: 10.1145/1449955.1449790. <http://doi.acm.org/10.1145/1449955.1449790>.
- [6] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 181–191. IEEE, 2018.
- [7] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, 2019.
- [8] Sean Gallagher. Rage-quit: Coder unpublished 17 lines of javascript and broke the internet, 2016. <https://arstechnica.com/information-technology/2016/03/rage-quit-coder-unpublished-17-lines-of-javascript-and-broke-the-internet/>.
- [9] Nicolas Harrand, Amine Benelallam, César Soto-Valero, Olivier Barais, and Benoit Baudry. Analyzing 2.3 million maven dependencies to reveal an essential core in apis. *arXiv preprint arXiv:1908.09757*, 2019.

- [10] Joseph Hejderup, Moritz Beller, and Georgios Gousios. Pr azi: From package-based to precise call-based dependency network analyses, 2018.
- [11] Danail Hristov, Oliver Hummel, Mahmudul Huq, and Werner Janjic. Structuring software reusability metrics for component-based software development. In *Proceedings of Int. Conference on Software Engineering Advances (ICSEA)*, volume 226, 2012.
- [12] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE, 2017.
- [13] ktrianta/rust-callgraph-benchmark: A benchmark for Rust call-graph generators, 2020. <https://github.com/ktrianta/rust-callgraph-benchmark>.
- [14] ktrianta/rust-callgraphs, 2020. <https://github.com/ktrianta/rust-callgraphs>.
- [15] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1317–1324. ACM, 2011.
- [16] Caroline Lima and Andre Hora. What are the characteristics of popular apis? a large scale study on java, android, and 165 libraries, 2019.
- [17] Maven Repository, 2020. <https://mvnrepository.com/>.
- [18] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. Mining api popularity. In Leonardo Bottaci and Gordon Fraser, editors, *Testing – Practice and Research Techniques*, pages 173–180, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15585-7.
- [19] Dong Qiu, Bixin Li, and Hareton Leung. Understanding the api usage in java. *Information and software technology*, 73:81–100, 2016.
- [20] rust-lang/crates.io-index: Registry index for crates.io. <https://github.com/rust-lang/crates.io-index>, 2020. Accessed: 2019-12-04.
- [21] Rust programming language, 2020. <https://www.rust-lang.org/>.
- [22] Anand Ashok Sawant and Alberto Bacchelli. fine-grape: fine-grained api usage extractor—an approach and dataset to investigate api usage. *Empirical Software Engineering*, 22(3):1348–1371, 2017.
- [23] Semantic Versioning 2.0.0, 2020. <https://semver.org/>.
- [24] The npm Blog — Details about the event-stream incident, 2018. <https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident>.
- [25] warp-crates.io: Rust Package Registry, 2020. <https://crates.io/crates/warp>.

- [26] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06*, pages 54–57, New York, NY, USA, 2006. ACM. ISBN 1-59593-397-2. doi: 10.1145/1137983.1137997.
- [27] Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo. Why reinventing the wheels?, 2019. <http://www.bowenxu.me/publications/EMSE2019.pdf>.