REM: Visualizing the Ripple Effect on Dependencies Using Metrics of Health

by

Zhe Chen
B.Sc., University of Victoria, 2019

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

REM: Visualizing the Ripple Effect on Dependencies Using Metrics of Health

by

Zhe Chen
B.Sc., University of Victoria, 2019

Supervisory Committee

Dr. Daniel M. German, Supervisor
(Department of Computer Science)

Dr. Miguel Nacenta, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Daniel M. German, Supervisor
(Department of Computer Science)

Dr. Miguel Nacenta, Departmental Member
(Department of Computer Science)

ABSTRACT

In recent years, free and open-source software (FOSS) components have become common dependencies in the development of software, both open source and proprietary. As the complexity of software increases, so does the number of software components they depend upon; in addition, software components are also depending on other components. Thus, their dependency graphs are growing in size and complexity. One of the current challenges in software development is that it is not trivial to know the full dependency graph of a software application and the vulnerabilities in it. Developers are usually aware of the direct dependencies their application requires, but might not be fully aware of the dependencies that those dependencies require (the transitive dependencies). Open-source software components, each as individual project, have health, a condition that is measured by the quality in different aspects of the development. Unfortunately, unhealthy software components as transitive dependencies can break any software application; therefore, software application developers need tools, methods and visualizations to inspect the health of these transitive dependencies and their potential impacts on the dependency graph of their software application. In this thesis, I first proposed and presented the Ripple Effect of Metrics (REM), a visualization of dependency graphs that leverages metrics of the health of dependencies. The two main features of the REM are: first, to display, and potentially summarize, the full dependency graph of an software application based on the health of each of its dependencies; and second, to display the ripple effect of vulnerable dependencies on the rest of the dependency graph. I then enhanced the features of an existing automated dependency tool in GitHub, open-source Dependabot, with the

REMs to become the REM-Dependabot. The REM-Dependabot helps maintainers of software applications to inspect the health of all of its dependencies, especially the vulnerable dependencies with CVE security advisories, and also explore the impact that some of these dependencies might have. I designed a case study on one of the most popular NPM JavaScript applications with three use cases each with a goal that the maintainers of that application need to achieve. The case study demonstrates and argues that the REM can be beneficial to developers as an aid to a more effective dependency management with its visualization of the ripple effect on the dependency graph using health metrics activity. A major portion of this thesis has been published at 2020 Working Conference on Software Visualization (VISSOFT)[10].

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

First of all, I would like to thank everyone who has been in my college life, as well as those who has contributed to the completion of my masters degree.
I would like to express my deepest appreciation to:

my parents, for their support, love and encouragement. Without them, I would never have enjoyed so many opportunities including pursuing a masters degree.

Daniel M. German, for guidance, support, and patience. Through out my life in the university, Daniel has not only patiently helped me to solve problems, but has also helped me to think differently, which will be beneficial for my future career.

Lulu Ke, Dandan (my lovely cat), and my friends, for supporting me in the low moments during the COVID-19 pandemic.

Chapter 1

Introduction

Software development relies on the reuse of software components, many of them open-source. Each of these components —a dependency—might have an independent software development process, with its own developer's team and release cycles, etc. Reusing one or more software component can result in **the dependency party**, which is a situation where dependencies are nearly impossible to be identified due to its incredible number when a software application uses a handful of software components as its dependencies. Many of those components, as individual projects, bring their dependencies over, and same to those dependencies of dependencies, repeatedly. While only a few "guests" are on "the guest list" of the "host", the actual number of "guests" presented could be significantly larger due to the "friends" that those "guests" and other "friends" brought over. In the world of dependency management, if we say the "host" to be the software application, then the "guests on the list" are the *direct dependencies* of the software application, and "friends" are seen as the *transitive dependencies*.

Using open-source software (OSS) components as dependencies is the current trend of the software development process and have received significant attention from industry (i.e. companies and organization) in recent years. In fact, according to Synopsys 2020 "Open Source Security and Risk Analysis" report[1], 99% of the codebases audited in 2019 contained open-source components. Developers tend to favor open-source software because it is readily and freely available and can be easily accessed as open-source software has all the information public, such as source code, changelogs, maintainers list, issue trackers, so that it is developed collaboratively and maintained by worldwide community members. By using open-source software components as dependencies, applications can achieve a much shorter development process with less

maintenance cost, and a much faster release cycle.

Over the years, open-source software have grown and expanded rapidly in the ecosystems which is a place where software cooperate and evolve together with communities that developers contribute and discuss. According to the recent report on Software Supply Chain (known as the dependency graph) from Sonatype[2], open-source software (including containerized application) has received 1.5 trillion download requests in 2020. For the world's largest software registry, Node Package Manager (NPM), the average monthly download traffic has grown more than 100% over the last year (2019), and the number of total packages(components) is reaching the 1.5 million mark, which is around 63% more than the number from one year ago. Furthermore, the more complex the ecosystem is, the more complex in size and depth the dependency graphs can become. Decan et al. computed the ratio of the number of dependencies over the number of packages in NPM software ecosystems for five years (from 2012 to 2017) and found out an increasing complexity relative to the number of packages[13].

1.1 Motivation

Since most components in the ecosystems are individual open-source projects, each project can have its own development teams. The development process for an open-source software component is similar to a company's development process regarding the different roles developers have. Where there usually is a combination of one or more project owners (the project manager) responsible for reviewing and accepting (by merging) others' code and many other online contributors (software developers) work on the codebase.

Open-source software projects have health, a condition that is measured by the quality in different aspects of the development. Different parties try to quantify the quality of the projects to have a health metric. For example, Community Health Analytics Open Source Software(CHAOSS) tried to understand and define health (of open-source projects) by creating different metrics[4]. NPM also introduced search scores[1] on every software package (each as individual project). Since many aspects of the development such as code coverage and release cycle can vary significantly among open-source software projects, it is often hard for developers to follow and

[1]https://docs.npmjs.com/searching-for-and-choosing-packages-to-download

track all the activities that happen to the software components they are using, a metric regarding the health of a project can help developers to know whether they should use an open-source software project as a dependency.

Open-source software project health can be affected by the software development process of a project, such as code coverage, unit testing, etc. Developers frequently create new open-source software, some are out of interest. And when developers' interest shifts to other domains or eventually fades away, the project will lose the attention and stopped being maintained, which could lead to a maintenance issue such as incompatibilities to the newer environments.

Security risks can also have strong impact on open-source software. (Common Vulnerabilities and Exposures) CVE[3] and (GitHub Security Advisory Database) GHSA[6] are two of the systems that publish and maintain lists of publicly known security vulnerabilities regarding software applications, including open-source. Open-source developers sometimes need to work on bug fixes related to security advisories released about these sources. All of those issues can impact the health of the open-source software projects. Therefore, by fixing them, the project can improve its health.

The health of the application is affected by the health of those open-source software projects that it depends on. The result from Cox et al. showed that (software) systems using outdated dependencies are four times more likely to have security issues as opposed to systems with up-to-date dependencies [11]. And more recent studies in 2019 suggested a significant percentage (up to 40%) of all NPM packages depend on code with at least one publicly known vulnerability [34]. For example, in 2018, *event-stream*, a popular NPM software package with 1.5 million weekly downloads and is depended on by nearly 1,600 software packages, was compromised after being transferred to a malicious user who then secretly injected malicious code into next release version. When the software was compromised and exposed, millions of applications that depend on this software package, whether directly or transitively, were immediately exposed to the security issue[2]. The *event-stream* incident demonstrated how millions of software applications' health can be exposed to a security risk and all of their health be negatively affected by one single compromised NPM package that they directly or transitively depend on. When a software package breaks its dependant software which further breaks many of their own dependant software packages, then this compromised software package is creating a ripple that affects every soft-

[2]https://medium.com/intrinsic/compromised-npm-package-event-stream-d47d08605502

ware package and application that depends on it. This is defined as the *ripple effect*, a propagation of an event from one software to every other software that is in the paths of the dependency relationship. A ripple effect of the problematic dependency in a dependency graph shows how the problematic one impact the health of the software application that relies on it.

Maintaining a good health for every dependency is challenging for dependency management; it often requires constantly checking for the availability of new software updates, verifying their safety, and addressing any migration efforts needed when upgrading a dependency [26]. However, managing dependencies is not a trivial task due to the reducing visibility of the dependencies. In a dependency graph, identifying the packages that are affected by the ripple effect of a vulnerable package can sometimes be very hard to detect when one dependency is used by many other dependencies so that it hides in a deeper layer of the dependency graph. Developers responsible for the maintenance of the dependencies are better informed of the direct dependencies because they are explicitly chosen to be added to the application. And because each of these dependencies might also rely on other dependencies for the functioning (the transitive dependencies), developers will gradually lose awareness and control over their dependencies and have trouble understanding the dependency graph. Fig. 1.1 represents an overview of development's visibility of the a dependency graph of a software application. The figure shows a ripple effect from a vulnerable dependency to the application as highlighted in red colour. When maintainers try to identify an unhealthy dependency, the further that dependency is located, the less obvious it is to maintainers, and the more difficult it is to identify that unhealthy dependency and observe its ripple effect. Usually, developers regularly update their direct dependencies manually to try to avoid vulnerabilities present in old and insecure versions. But this method will not traverse the entire dependency graph, thus leave the hidden unhealthy dependencies remained in the deeper layers of the dependency graph.

A growing number of software applications use continuous practices (i.e., continuous integration, delivery, and deployment) for dependency management since continuous practices have become an important area of software engineering research and have been successfully applied to both greenfield and maintenance projects [29]. As GitHub has become the most popular platform for people to host and participate in open-source projects, developers start to enable continuous practices into their development pipeline for fast and secure product releases. Dependency management is no exception. GitHub Marketplace offers automated dependency updates bots that

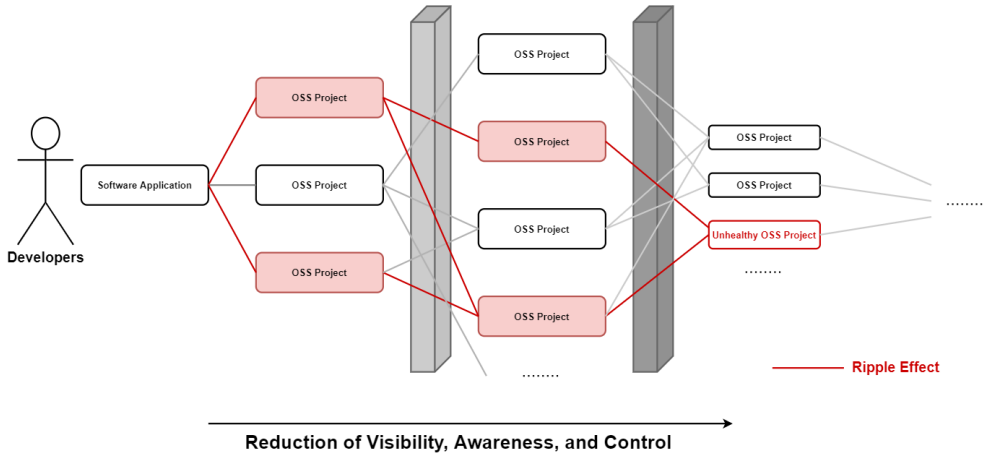


Figure 1.1: Developer's visibility, awareness and control of the application's dependency graph and the propagation as the ripple effect from the unhealthy software project to the application

can be fitted into one of those pipelines. For instance, one of the most popular bots, Dependabot[5], is integrated into GitHub to help monitor the freshness (i.e., the use of recent versions) and vulnerability of their dependencies. The essential task of Dependabot is to notify developers once a newer or vulnerable dependency is found and create a pull request (a local change that is submitted to a remote project waiting for reviews before it gets merged into the production code) that automatically bumps up to the latest secure version available.

Depending on the dependency management system's configuration, with automated dependency updates bots such as Dependabot, developers sometimes are limited to only update the dependencies explicitly defined (the direct ones). To enable transitive dependency updates (only on some programming languages such as JavaScript), developers must maintain a dependency-tree file auto-generated from the current configuration file, namely, the lockfile. However, lockfile maintains the dependency-tree that is a full hierarchy on how the dependencies are required and defined(Fig. 1.2). As the lockfile is well-known for machine-generated and not very reader-friendly due to the size and lack of information on the dependencies, developers can be easily overwhelmed.

Thousands of lines are a typical size for a lockfile. For example, `Metrics`[3], a popular NPM open-source application with more than 2,200 stars received, contains a lockfile with more than 7,000 source-lines-of-code) SLOC. Therefore, when it is diffi-

[3]https://github.com/lowlighter/metrics

```
{
  "name": "example",
  "version": "3.2.0-beta",
  "lockfileVersion": 1,
  "requires": true,
  "dependencies": {
    "@actions/core": {
      "version": "1.2.6",
      "resolved": "https://registry.npmjs.org/@actions/core/...",
      "integrity": "sha512-ZQYitnqiyBc3D+k7LsgSBmMDVkOVidaagD..."
    },
    "@actions/github": {
      "version": "4.0.0",
      "resolved": "https://registry.npmjs.org/@actions/github/...",
      "integrity": "sha512-Ej/Y2E+VV6sR9X7pWL5F3VgEWrABaT292...",
      "requires": {
        "@actions/http-client": "^1.0.8",
        "@octokit/core": "^3.0.0",
        "@octokit/plugin-paginate-rest": "^2.2.3",
        "@octokit/plugin-rest-endpoint-methods": "^4.0.0"
      }
    },
...
```

Figure 1.2: `package-lock.json`, a lockfile example in JavaScript/NPM

cult for developers to maintain the dependencies in the lockfile, they use automated tools such as Dependabot. When there is a newer desired version found on transitive dependency after Dependabot scanned the lockfile, Dependabot will create a pull request regarding the version bump with changes made only to the lockfile. The pull request shows the information regarding the updating vulnerable dependency, such as the severity level of the vulnerability, commit history of the newer version and maintainers' information.

The pull request that Dependabot creates provides information regarding the dependency update, such as commit history and file changes for project developers to review. However, automated tools such as Dependabot lack the information to show where the vulnerable dependency is located (in the dependency graph) and how the vulnerability affects the application so that developers can have a better understanding of the exposure of the vulnerability when reviewing the dependency-related pull requests.

1.2 Problems with Using Dependencies

Application developers need to deal with the entire software application's dependency graph, which can become large enough to hardly understand due to transitive dependencies. We used GitHub data collected by GHTorrent[19] of up to June, 2020. From the data, around 100,000 NPM applications were selected that have been using dependencies and also received attention from the public (i.e. number of stars and forks). Fig. 1.3 shows the scatter plots of runtime and development dependencies of the 100,000 NPM applications collected. And as these figures show, the number of direct runtime dependencies tends to be small with median of 7, but the number of total transitive dependencies can be very large with median number of 80, meaning that more than half of the applications have only required as low as 7 dependencies but the actual number explodes to more than 80. The numbers for development dependencies are even more skewed, with a median of 9 direct development dependencies and 408 transitive ones, meaning the task of managing dependencies requires a thorough understanding of the dependency graph.

Relying on unhealthy dependencies can break or cause problems to the application being developed. Problems regarding using open-source software projects as dependencies is usually because of the large dependency graph of the software application with the reduction to the development's visibility, awareness and control of the dependencies (as explained previously using Fig. 1.1). It is often difficult to identify the unhealthy ones in the entire dependency graph of the application and how it affects the application. Therefore, specifically, the fundamental problems with using software projects as dependencies in a large dependency graph are the followings:

1. **Identifying what and how dependencies are used in application is difficult**. Identifying dependencies is important to the software project management because when transitive dependencies are brought into the dependency graph, it can be difficult to know how these dependencies are used by other dependencies or the application and whether they have any security risk. Also, the application and its dependencies can sometimes rely on the dependencies in a situation where they are same project but with different versions which distinguish them as different dependencies. Hence, it significantly increases the difficulty of identification. For example, by manually scanning through the lockfile of `Motrix`[4], software component `ansi-regex` has appeared 14 times as

[4]https://github.com/agalwood/Motrix

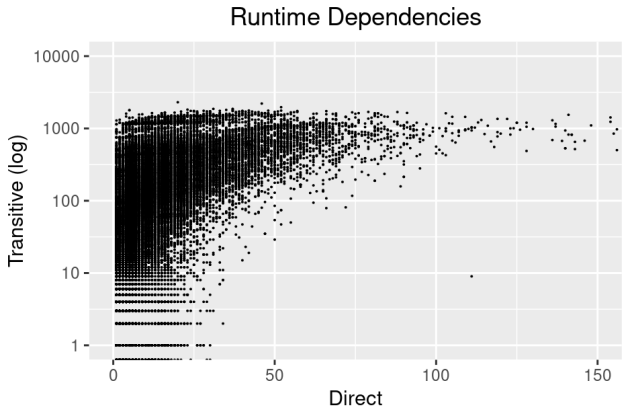
a transitive dependency in the dependency graph with four different versions. However, due to the large size of the lockfile (over 14,000 SLOC), it remains mysterious when it comes to which dependency has required it, at what version, and what is its relation with the application (how it is used).

2. **Identifying the propagation of unhealthy dependencies is difficult**. In modern ecosystems such as NPM, one way to manage dependencies is to update the vulnerable dependency version in a temporary dependency file(the lockfile). which further change the content of the dependency under a directory so it can build and run the application using the patched version instead of the vulnerable one. Therefore, it is important to identify the precise location of a dependency when the dependency becomes risky to know where to replace. As more dependencies are presented in the greater depth of the dependency graph, developers lose visibility, awareness, and control of their dependencies. Current dependency management tools lack the ability to pinpoint the precise location where the health of dependencies is problematic, and fail to identify how the problematic dependencies propagate its risk to the rest of dependencies including the application (this is also demonstrated later in Chapter 6).

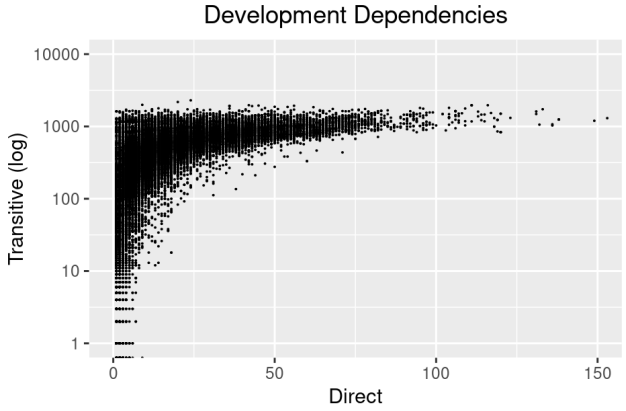
1.3 Proposed Solution

When unhealthy software projects can cause problems to its dependant applications, both developers and users of a software application should have means to evaluate the health of the application that is being developed or used. One of the methods to achieve this is to have a visualization that displays the health of the dependencies of the software application, and highlights the vulnerable ones if necessary. When an automation tool such as Dependabot is enhanced with such visualization (showing the information on vulnerable dependencies), developers can make a better decision when maintaining the dependencies of the application. Application users can also decide to whether or not use the application based on the dependency graph showing the vulnerable dependencies.

With the goal of helping developers and users gain a better understanding of the impact the vulnerable dependency can have on the software application they are building or using, in this research, I therefore propose a dependency graph visualization, namely the Ripple Effect of Metrics (REM). The REM is a dependency graph



(a) direct and transitive runtime dependencies distribution



(b) direct and transitive development dependencies distribution

Figure 1.3: Two scatter plots that show distributions of the relationship between direct and transitive dependencies across over 100,000 popular GitHub NPM/JavaScript applications

with vertices and edges annotated in a way to show the health of dependencies and highlight the risky ones and the ripple effect of problematic ones. The REM uses open-source software projects' metrics of health (for example, those that measure aspects of the development process, such as quality, development activity) and whether a dependency is vulnerable and shows their ripple effect in the dependency graph. To achieve this, the REM improves traditional dependency graph visualizations by adding two features. First, as shown in Fig. 1.4, the REM contains annotations on the dependencies that show different colours according to their metrics of health to show the risky dependencies that have lower value. And second, the REM highlights the

ripple effect of the vulnerable dependencies (the red node in the example figure) to reveal the propagation of problematic dependencies determined by the vulnerability metric.

The REM can be used in the software development process where dependencies from a software ecosystem of modern programming languages are used. By using the REM on any software application using dependencies from open-source ecosystems, developers should be able to identify potentially vulnerable dependencies based upon their metrics of health and understand how these potential vulnerabilities propagate through the dependency graph.

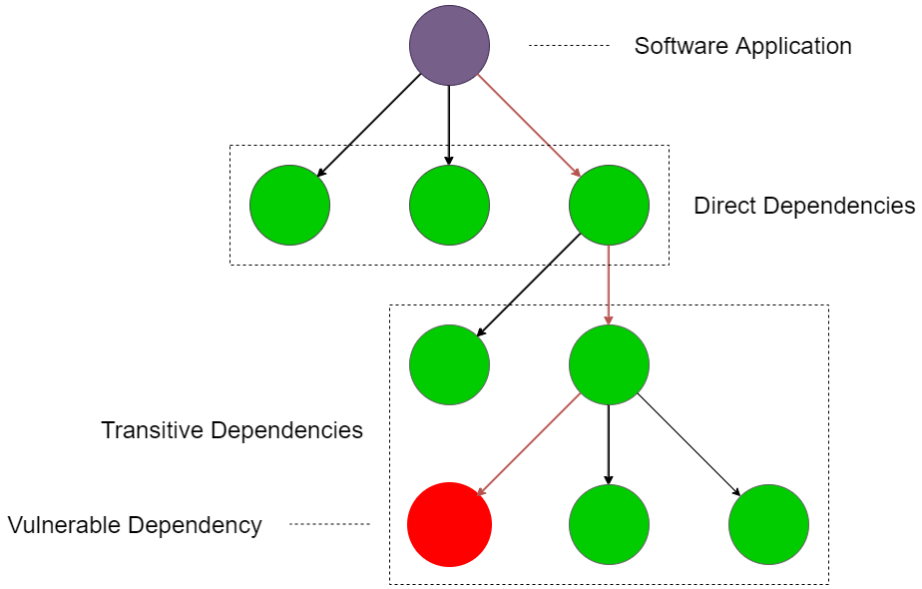


Figure 1.4: Ripple Effect of Metrics (REM) as a solution for revealing dependencies' health and propagation of the vulnerability

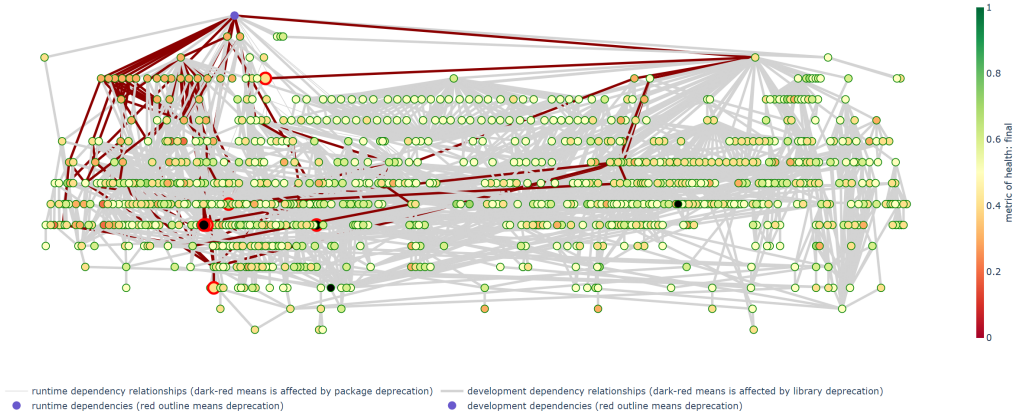
1.4 Contributions

The main contributions of this thesis can be summarized as the followings:

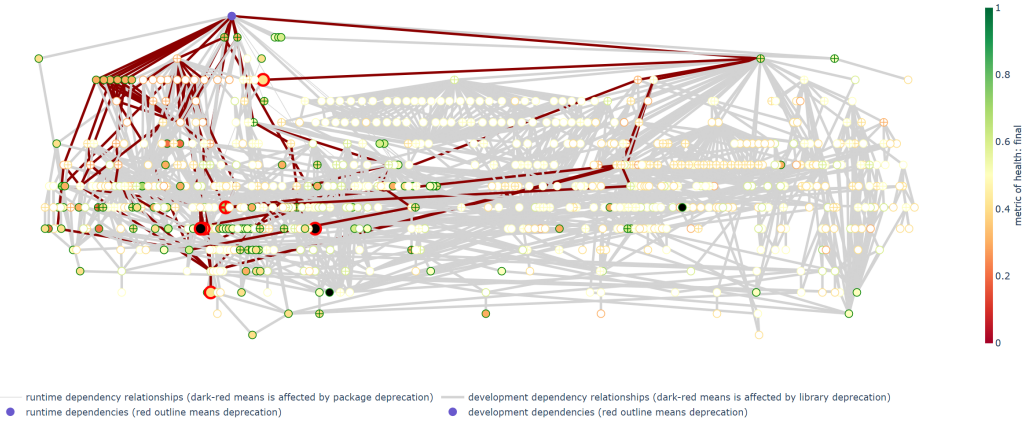
1. **The Ripple Effect of Metrics (REM) visualization**. The REM is an improved graph visualization over the hierarchical dependency graph to effectively show the extra information regarding the health metrics of dependencies and their potential ripple effects on the software application that uses them.

2. **Implemented the REM on NPM software applications**. In this thesis, I implemented the NPM-REM, that focuses on the software applications that

use open-source software packages from the NPM ecosystem (example shown in Fig. 1.5a). The implementation details include data collection, processing, and the creation of the REM in HTML format. The NPM-REM uses metrics of health defined and made available by NPM for their search result rank and the vulnerability metric from the GitHub security advisory database.



(a) Full REM Dependency Graph

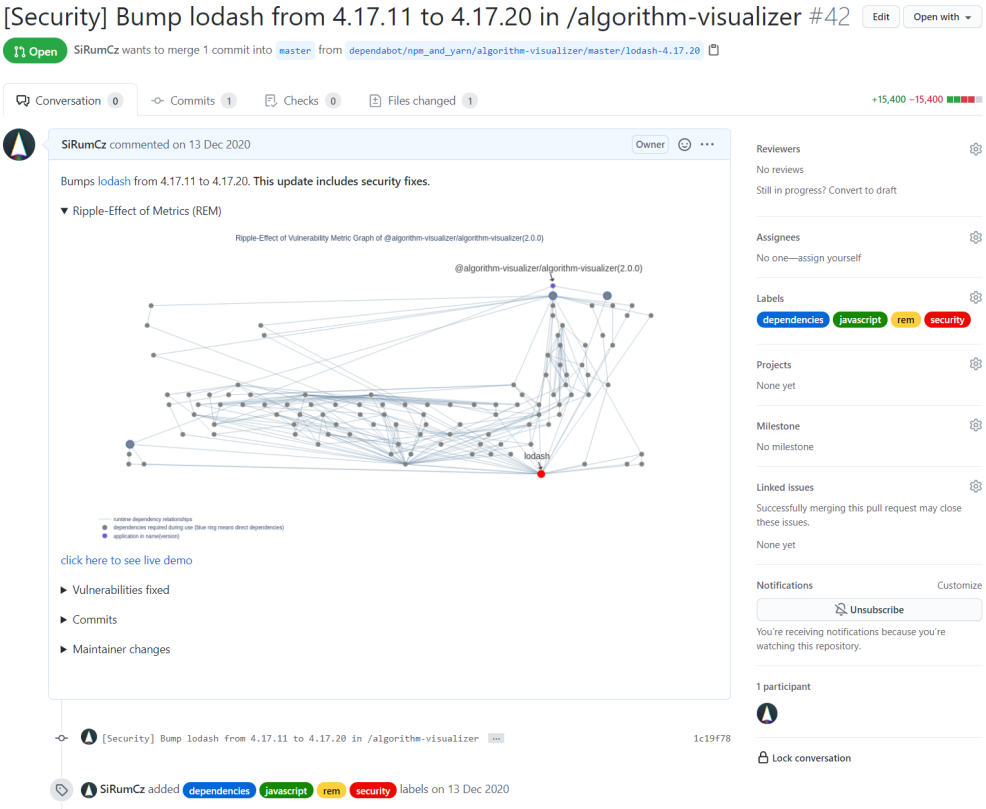


(b) Reduced REM Dependency Graph using *Filtering* and *Grey-out*

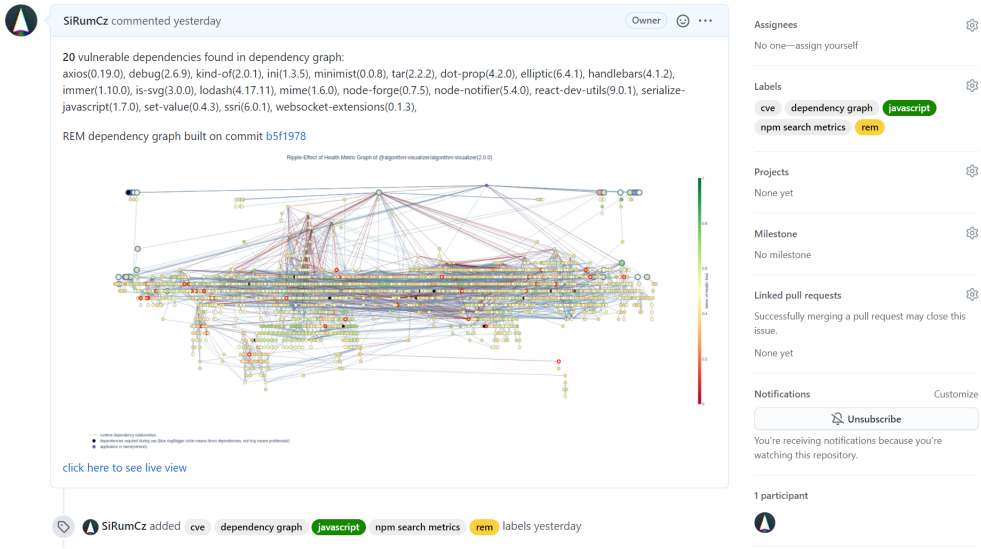
Figure 1.5: Two REM examples using NPM `final` score as dependencies' metric of health and component `deprecation` state as ripple effect metric

3. **Implemented two filtering algorithms on the REM**. In the NPM-REM, two algorithms are also included to help improve the readability of large dependency graphs of the REM: *Collapsing* and *Grey-out* (example shown in Fig. 1.5b, which is the filtered version of Fig. 1.5a).

4. **REM-Dependabot: an extended open-source Dependabot that uses the NPM-REM visualization**. This thesis extends the open-source Dependabot, an automated dependency management tool hosted in GitHub, with the NPM-REM. By extending the Dependabot, the REM helps in providing visual supports on the highlighting of the vulnerable dependency and its ripple effects to the application and providing a set of interactive REMs for understanding and exploring the weaknesses in the dependency graph of software application. This REM-Dependabot has two functions: 1) enhances the GitHub Pull Requests created by Dependabot (example shown in Fig. 1.6a) on the dependencies' CVE security vulnerability with a REM customized to only highlight the problematic dependency and its ripple effect, and 2) the creation of a new feature that creates GitHub Issues (an example is shown in Fig. 1.6b). In the Issues created by REM-Dependabot, dependencies with CVEs are listed, followed by a preview of a REM and a link to a set of REMs that use score metrics from NPM as metrics of health and CVE as vulnerability metric. This set of REMs includes a complete and a filtered version of the dependency graph with the ripple effects from the dependencies with CVEs for every metric of health. Both REM (in pull requests and issues) include a preview image of the REM and a link to the live-view with interactions such as zoom, tooltip, and toggle to show different types of dependencies.



(a) Pull Request example using `security vulnerability` as Ripple Effect metric



(b) Issues example using NPM `final` score as dependencies' metric of health

Figure 1.6: Two examples of the REM-integrated Dependabot

5. **Case Study**. To further demonstrate the features and usefulness of integrated NPM-REM and REM-Dependabot, this thesis includes a case study on a selected NPM software application with two use cases one shows the GitHub pull request and anther shows the GitHub issue, both using REM-Dependabot. Comparisons between with and without the REMs are also given to show that REM in dependency update pull request is effective in pinpointing the vulnerable dependency and showing how the application is affected by the vulnerability, and the set of complete REMs can provide the opportunity to identify weaknesses in the dependency graph.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 contains background information about existing visualizations regarding dependency updates, software ecosystem, dependency management, and description on why I choose NPM and Dependabot in this research.

Chapter 3 introduces the REM. This chapter includes a description of the design of the REM model, metrics of health and the vulnerability metric used in the REM, and two filtering algorithms that improve the readability of the REM by reducing the size of the graph to focus on unhealthy and risky dependencies.

Chapter 4 describes the implementation details on the NPM-REM, which is a research prototype that aims at generating the REM for NPM software applications. This chapter describes the process to build the dependency graph, collect and annotate the example metrics of health from NPM search engine in the dependency graph, and collect and use the package deprecation as the Ripple Effect metric on the dependency graph.

Chapter 5 is the integration of the REM into the open-source Dependabot. This chapter contains the integration process, implementation details and a general walk through on how REM-Dependabot works toward the vulnerable dependency fix and show the dependency vulnerabilities in the REM.

Chapter 6 contains a case study with one selected NPM software application to argue that the REM can effectively help developers maintain their dependencies by showing two use cases and comparing the scenarios with and without the presence of the REM.

Chapter 7 discusses different aspects of the building process of the REM, providing

more insight on the effectiveness of the REM. The different aspects include data reliability, the processing of dependency information and concerns regarding visual elements. The chapter discusses the limitations of the REM and the future work. Finally, the chapter concludes the thesis.

Chapter 2

Background and Related Work

This chapter includes a description of software ecosystems, related work, dependency management in general, and an explanation of why this work focuses on the NPM software ecosystem and Dependabot.

2.1 Software Ecosystems

According to Wikipedia[1], an ecosystem is a community of living organisms in conjunction with the nonliving components of their environment, interacting as a system. A software ecosystem is very similar in nature and has been defined as a collection of software projects which are developed and evolve together in the same environment [25]. Traditionally, software engineering research has focused on individual software systems [13]. The development of the tools for distributed version control system such as Git[2] and the online repository platform, GitHub, has brought the attention of online collaborative development to organizations and developers' communities (especially open source), which then created the environment where developers contribute to software packages by participating in the publishing and maintenance of the software written for specific programming languages or running under certain system environment. Common examples of the described software ecosystems can be the Debian distribution for Linux operating system and CRAN software ecosystem for R programming language.

In recent years, software ecosystems for programming languages has gained a wide attention by researchers. German et al. [16] and Kula et al. [23] has studied

[1] https://en.wikipedia.org/wiki/Ecosystem
[2] https://git-scm.com/

the software ecosystem for R, Decan et al. [12, 13] and Witten et al. [31] has studied the NPM software ecosystem in many of their works, and Raemaekers et al. [27] has successfully studied Maven software ecosystem.

Software ecosystems usually come with at least one centralized registry that stores most of the published software projects for other software to use and software ecosystems evolve with the growing numbers of open source software packages published to their centralized registries by the developers' communities. Those open-source packages are mostly created with the goals of providing functionalities then can be used as dependencies by both, other software packages inside the ecosystem and software applications outside of the software ecosystem. Some centralized registries in software ecosystems contain hundreds of thousands packages, each with many different versions, from backward-compatible minor or patch versions to the major releases with breaking changes that can easily break the software, and all of which bring the challenge to the task of dependency management.

2.2 Dependency Management

Assume A is a software component in NPM—an NPM package—that provides some functionalities in helping build another software B within the NPM software ecosystem. A is usually referred to as a **dependency** of B, and B is referred to as a **dependant package** of A. Dependencies can have different purpose of use. In contemporary software development process, developers usually separate runtime from development dependencies. The runtime dependencies are software packages used to run the application (e.g. frameworks) and/or when the application is running; while development dependencies are software packages used during the development process or build time (e.g. testers). Therefore, configuration files used by package managers usually allow developers to define different types of dependencies using keywords documented. While researches such as the work of German et al.[17] that studied the dependency graph of Debian GNU/Linux ecosystem using building and running dependency information defined in a single *Stage* classification, they have also clarified that each dependency type has its very own purpose. In fact, the Debian management system separates run-time dependencies (`Depends`) from build dependencies (`Build-Depends`) using different keywords[3]. Similarly, in the NPM software

[3]https://www.debian.org/doc/debian-policy/ch-relationships.html

ecosystem, the running and building dependencies are separated into two different types: `dependencies` and `devDependencies`, respectively.

Depending on the software ecosystem, developers often need to maintain a configuration file that contains the information such as the name and the version (or the version constraint, a range of versions) about the dependency. For examples, to define the dependencies, JavaScript/NPM software applications use `package.json` file (Fig. 2.1), Java/Maven software applications use `pom.xml` file (Fig. 2.2), and Ruby/Bundler software applications use `Gemfile` (Fig. 2.3).

```
{
    "name": "example",
    "version": "1.0.0",
    "dependencies": {
        "vue": "^2.5.13",
        "vue-material": "^1.0.0-beta-7"
    }
}
```

Figure 2.1: `package.json` example in JavaScript/NPM

```
...
<artifactId>example</artifactId>
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit-dep</artifactId>
        <version>4.10</version>
    </dependency>
</dependencies>
...
```

Figure 2.2: `pom.xml` example in Java/Maven

Developers should always use the latest compatible versions of every dependency they use in their applications. Consider the Javascript/NPM software ecosystem as an example. To update dependency management file, developers need to first identify which dependency is outdated. This can be done via NPM CLI (Command Line Interface) using `npm outdated` command (result shown in Fig. 2.4 as an example).

```
source 'https://rubygems.org'

gem 'nokogiri'
gem 'rack', '~> 2.0.1'
gem 'rspec'
```

Figure 2.3: `Gemfile` example in Ruby/Bundler

NPM also offers the CLI command `npm update` to update to the latest wanted version, a up-to-date version that is compatible with what the application specified as the version constraint in the dependency file. In the example of Fig. 2.4, since the wanted version is same as current one, there will be no changes made to the current dependency setup. That is because, while the newer version (2.0.0) is newer than the current one (1.2.0), according to the Semantic Versioning[4] (a three-bit versioning schema for communicating stability) that NPM recommends developers to follow[5], the newer version, with a bump in its major version, is likely backward-incompatible (i.e. break the software functions with incompatible components).

```
> npm outdated
Package              Current  Wanted  Latest  Location
vue-prism-component    1.2.0   1.2.0   2.0.0  metrics
```

Figure 2.4: example of the outdated dependency list from `npm outdated` on **metrics**, a JavaScript software application

An empirical study from Bogart et al. showed that developers in two software ecosystems (R/CRAN and JavaScript/Node.js) hold different attitudes towards dependency management and struggle with changes because of reasons such as a flood of information from the mailing list and dependency activity [9]. Results from Kula et al. showed that more than 60% of the interviewees were unaware of their vulnerable dependencies, more than 80% of studied (software) systems still keep their outdated dependencies, and developers affected by vulnerable dependencies are not likely to respond to a security advisory [24]. One explanation for developers to neglect to

[4]https://semver.org/

[5]https://docs.npmjs.com/about-semantic-versioning

update software dependencies is the difficulty of constantly checking for the availability of new software updates, verifying their safety, and addressing any migration efforts needed when upgrading a dependency [26]. This explanation can be used to describe the traditional methods (NPM CLI commands, for example). While it usually updates your dependencies, it requires developers to regularly check for the new updates, and developers need to manually verify its safety and address any migration effort.

2.3 Related Work

Many researchers have investigated the modeling and visualization of dependency graphs[21, 17, 18, 22, 30, 31, 15, 23, 14, 20]. For example, Falke et al. suggested a hierarchical graph layout for visualizing the functional dependency graph of software applications; by using a hierarchical layout, users can view the top-level nodes as a first overview and step-wise unfold the nodes on demand to look into details [15]. Dias et al. implemented Hunter, a visual tool that includes node-link diagrams as dependency graphs for understanding JavaScript source code [14]. German used package management files to visualize the dependency graph in the hierarchical layout of three popular applications in Fink, one of the software distribution for macOS, and attempted to identify the use of applications in terms of different types of success [18]. The REM in this research is extended from the concept of German's work by adapting the technique of creating dependency graph and adding the visual annotations that highlight specific aspects of the dependencies' information to have a more meaningful dependency graph.

Ever since the *event-stream* incident in 2018, Developers' communities has expressed concerns about the dependency health. In recent years, researchers have studied problems associated with dependency relationships using historical data [13, 21]. Decan et al. have adopted the use of external sources [12, 13]. Researchers conducted an empirical study on library (dependency) migration and showed that most of the studied projects keep outdated dependencies [24]. Yau et al. investigated the ripple effect from the location of a modification to other parts of the system that are affected by such modification [32]. While researchers have been broadly studying the dependency management, two areas have not gained much attention from researchers: package deprecation as a vulnerability, and the use of NPM package scores as information linked to the health of a dependency. The former area can have a significant

impact on the ecosystem [28], and package scores provide valuable information regarding the way a package is perceived by its development process and community. Robbes et al. used the ripple effects of deprecation to study the propagation of changes in the ecosystem as a whole [28].

As Bergle et al. [8] suggested, adequately visualizing software dependencies as graph is a non-trivial problem due to the multi-dimensional nature of software, the technique of dependency visualization varies from different aspect to aspect of the software. Isaacs et al. [20] developed an interactive dependency graph visualization that utilizes ASCII and brings a full dependency graph through the command line with certain degree of interactions, Fig. 2.5 shown as example on `dia`[6] specified in Spack management tool. While ASCII dependency graph visualization can be easy to integrate and have better cross-platform compatibility, the limitation of it can also be noticeable, such as the large graph presentation problem with the limited space on a typical command line interface.

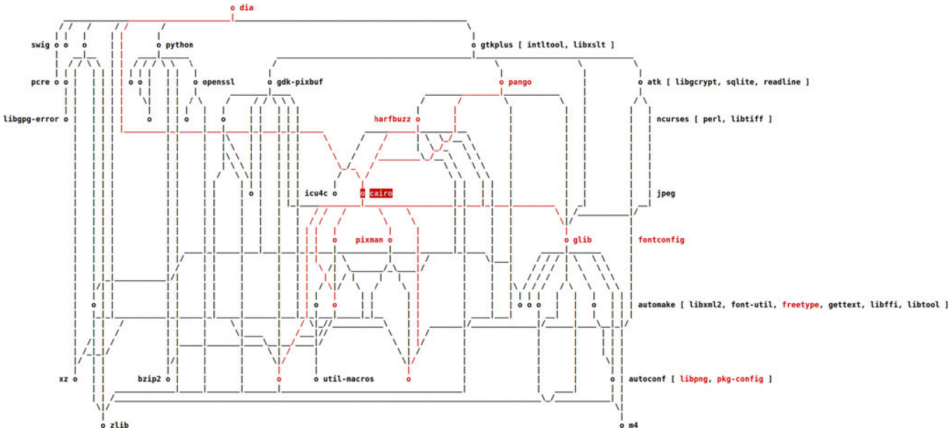


Figure 2.5: Example of ASCII depiction of the package dependency graph of `dia`[20]

Many Existing tools investigate on identifying dependency update opportunities. Kula et al. defined a set of visualizations to show how the dependency relation between a system (software application) and its dependencies evolves, one of which is the (system-centric dependency plots) SDP. SDP uses a radial layout and heat-map to show the change in dependencies on the package's release history. Their visualization shows the dependency version usage and dependency change by assigning different colours and shapes to the dependency nodes [22]. Fig. 2.6 is an example

[6]https://wiki.gnome.org/Apps/Dia

of System-centric Dependency Plots used on FindBugs software[7]. While their work showed a history of package release on direct dependencies along with annotations to show the attributes of dependencies (i.e. type and version usage), the visualization did not continue to show any connection among these dependencies. SoL Mantra[8] is a tool developed by Todorov et al. [30] to show the libraries' update opportunities based on the work of Kula et al. [23]. Another one is SoL Mantra, which is an orbital layout visualization that adapted the Wisdom of the Crowd in software ecosystem and developed the the coexistence coefficient (cc), shown in Fig. 2.7 and Fig. 2.8 as demonstration of the orbital layout visualization and coexistence coefficient. While both tools provided information on the dependencies and helped in dependency update, neither tool is designed to with the goal of identifying problems not only with the direct dependencies, but also with the transitive ones.

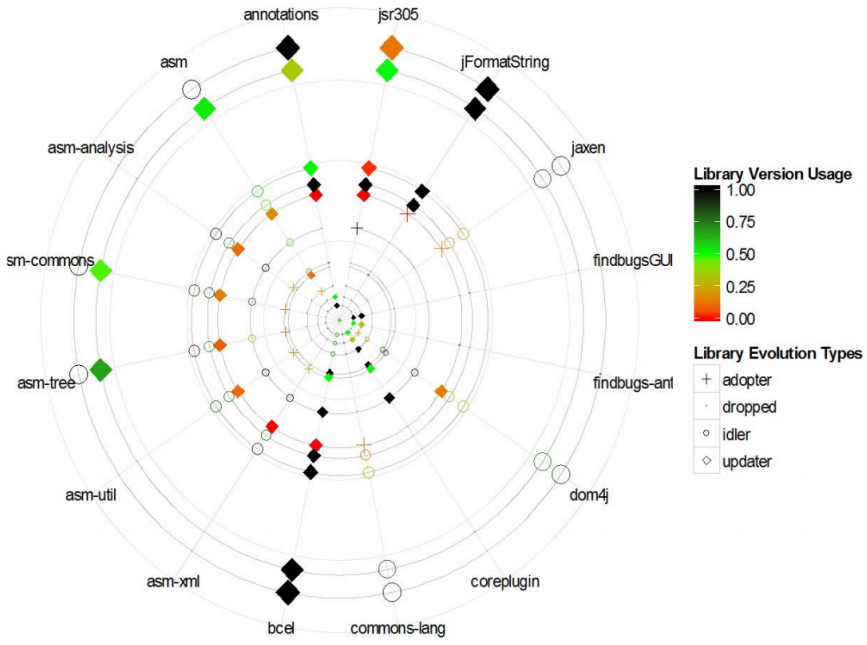


Figure 2.6: Example of System-centric Dependency Plots for FindBugs system[22]

[7]http://findbugs.sourceforge.net/
[8]https://sel.ist.osaka-u.ac.jp/people/boris-t/

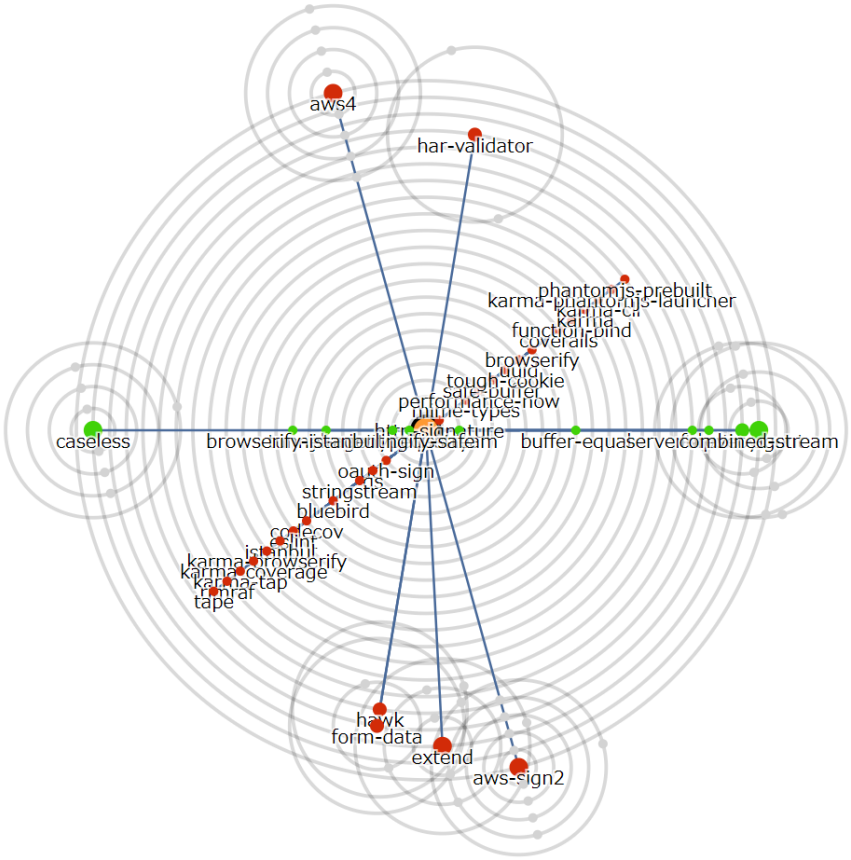


Figure 2.7: Example of SoL Mantra for **request**, an NPM package[30]

2.4 NPM and Dependabot

To investigate a better solution to manage dependency, this thesis focuses on the NPM software ecosystem for JavaScript. JavaScript is one of the most popular programming languages in recent years. Node Package Manager (NPM) is a package management system specifically for Node.js platform for writing server-side applications in JavaScript programming language. Table. 2.1 shows the statistics recently retrieved from NPM official website. The total number of packages hosted in NPM registry is significantly greater than many other software package managers. For example, at the same time the NPM statistics was collected, (The Python Package Index) PyPI hosts 284,303 projects[9], and according to libraries.io [7], Maven hosts 378,730 packages[10].

[9]https://pypi.org/
[10]https://libraries.io/maven

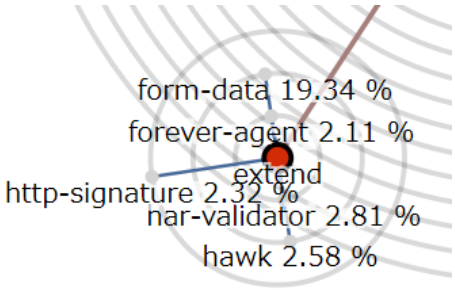


Figure 2.8: Example of coexistence coefficient (cc) on one of the dependencies in SoL Mantra for **request**, an NPM package

Table 2.1: statistics retrieved from NPM official website (as of January 18, 2021)

Total packages	**Weekly downloads**	**Monthly downloads**
1,493,231	26,068,305,652	85,237,282,447

Over the years, NPM has been encouraging package owners to use GitHub as the code repository for their packages and in fact, most of the NPM packages are now using GitHub as their official repositories for development collaboration. This enables an easy access between the two platforms to maintain the healthy ecosystem.

As more and more software in NPM ecosystem host their projects in GitHub, to avoid manually checking for the availability of new dependency updates, developers begin to use automated tools. Through a market section named GitHub Marketplace, GitHub offers great selections of plugin for developers to use on the Continuous Integration (CI) pipeline of their repositories. One of the categories that Marketplace has is dependency management[11]. It contains automated tools which provide information about dependency update and help shorten the dependency management process.

Dependency management bots are automated tools that are typically integrated by maintainers into their project repositories with the goal of helping keep dependencies secure and up-to-date. This research focuses on enhancing features of the Dependabot, a GitHub-native dependency management bot. While there are many other bots available such as Greenkeeper[12] and Snyk[13] that offer excellent experience at providing dependency update features and have open-source repository, the reasons why this research chooses Dependabot are the followings:

- **Dependabot has a GitHub-native version that is widely used by de-**

[11]https://github.com/marketplace/category/dependency-management
[12]https://github.com/apps/greenkeeper
[13]https://github.com/marketplace/snyk

velopers. Dependabot was directly integrated into GitHub in 2019 earlier as the official dependency management tool, and it supports many popular programming languages such as JavaScript, Python, Go, Java, and Ruby, making it the most popular dependency management bot,

- **Core features are open-source, makes Dependabot ideal for research prototyping**. open-source Dependabot provides the source code, written in Ruby, that implements the basic features and core functions in a public GitHub repository. In addition, offers a scripting repository for the purpose of demonstrating its use that is simple enough to understand, can run locally, and most importantly, fully customizable for the research purpose.

The operation process of Dependabot or any Dependency management bot can be generally described as the following three steps [5]. First, the automated bot pulls down dependency files and looks for any outdated or insecure requirements; if any dependencies are out-of-date, the automated bot opens individual pull requests to update each one. Once project owners check that their tests pass, verify the included changelog and release notes, then merge the pull request.

However, while many information are provided such as the information about the version release, the vulnerability information and maintainer changes, little information are given about the dependency relationships to the application or the impact of the risk on the application for not updating the dependency. This can cause trust issues since developers might not fully understand the consequences for not updating the dependency and can be reluctant to perform the update. In this thesis, the Dependabot will be extended to include the REMs with the goal in helping developers make better decisions regarding dependency updates with the visual support from the REM. The extended Dependabot can show the ripple effects of the updating dependency on to the application in the pull request, and it will also maintain an overview of the dependency graphs of the application in a separate GitHub Issues section using the REMs.

Chapter 3

Ripple Effect of Metrics Visualization

This chapter first describes the visualization model of the Ripple Effect of Metrics (REM), including two critical metrics, a definition of the ripple effect, and a definition of the REM. Then the chapter describes the visual rendering of the REM. Finally, two filtering algorithms on the REM are described with a demonstrated example.

3.1 The Ripple Effect of Metrics

Before defining the Ripple Effect of Metrics, two metrics that are playing important roles in the REM, *metric of health* and *vulnerability metric*, will be explained with examples.

3.1.1 Metric of Health

A *metric of health* measures the aspects related to the health of a dependency–such as its quality. In other words, a metric of health is a rating that a third party provides regarding the quality of the dependency, whether subjective or not.

This thesis focused on the analysis of NPM applications and their dependencies. For every NPM package (a dependency), NPM provides a set of health metrics to measure and rank different aspects of its quality, popularity and development process. These metrics can be observed when performing a search action on NPM packages. For example, Fig.3.1 is the snapshot of a search result from NPM. As the snapshot

shows, the metrics measured are shown on the right-hand side of each result package. Specifically, these metrics are[1]:

- **Popularity**. The *popularity* score is an indicator on how many times the package has been downloaded.
- **Quality**. The *quality* score includes considerations such as the presence of a README file, stability, tests, up-to-date dependencies, custom website, and code complexity.
- **Maintenance**. The *maintenance* score associates with the attention from developers where higher scored package usually is more frequently maintained in terms of release, commit, etc.
- **Final**. The *final* (or optimal) score combines all three score metrics in a meaningful way.

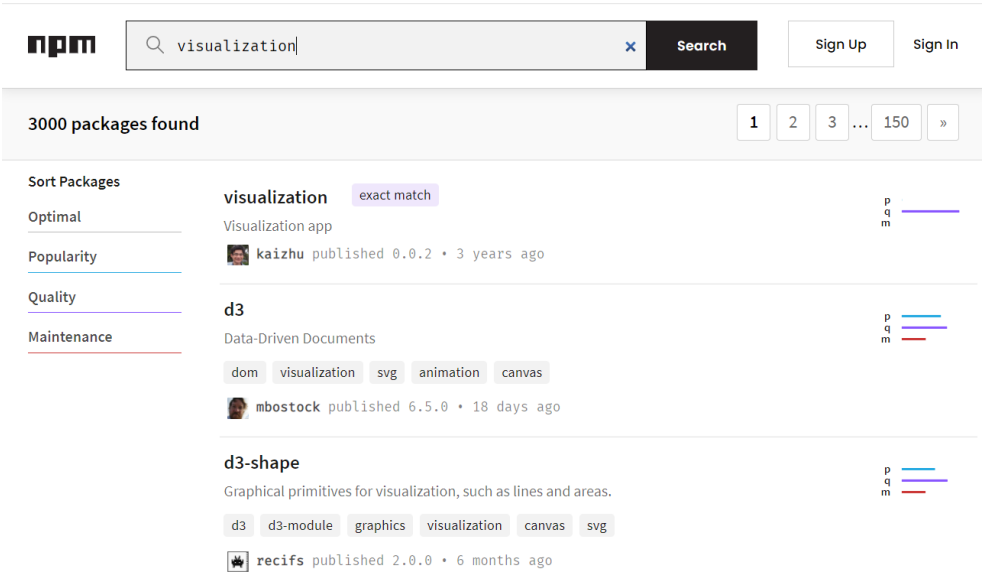


Figure 3.1: A snapshot of *"visualization"* search result from NPM that shows different metrics of NPM packages

Metrics of health can be used to represent the health of a NPM package and which then can be used to represent the health of a dependency graph in the REM. When a dependency has a low metric of health in a dependency graph, it becomes a risky dependency and there is a potential risk for applications that use it.

[1]https://docs.npmjs.com/searching-for-and-choosing-packages-to-download

3.1.2 The Ripple Effect

In general, the ripple effect is a concept of how an event affects another through propagation. The ripple effect can have either negative or positive impacts in different contexts. This thesis focuses on the ripple effect of unhealthy dependencies (i.e. with low metrics of health) that has a negative impact on dependencies including the application being developed that uses them.

3.1.3 Vulnerability Metric

Similar to the metric of health, a *vulnerability metric* is a flag used to show whether a package is vulnerable. Such vulnerable dependency has a negative impact on those dependencies that use it. A problem in this dependency can propagate (ripple) through the dependency graph all the way to the application being developed. Whether a dependency is vulnerable can be based on a third party report, or from the metric of health. The vulnerability metric is by definition binary. One example of such metrics is the **security advisory** reported by NPM. A security advisory identifies an NPM package as vulnerable due to its exposed security risk, and the vulnerability metric will be to `true`. An other example is **library deprecation**, such that it is considered to be vulnerable if the dependency is deprecated. For metrics that have a range of values (such as code coverage, number of downloads etc.), the user of the REM can define a value for the metric as a threshold: when the metric is above (or below, depending on the interpretation of the metric), the dependency is considered to be vulnerable. Consider the ripple effect of dependencies with **maintenance** score lower than 0.5 as an example. NPM packages with maintenance score higher or equal than 0.5 would be considered to be normal (non-vulnerable), and packages with score lower than 0.5 would be considered vulnerable.

The *vulnerability metric* is a flag that shows if a dependency is insecure for applications to use. When this flag is `true`, it means that the dependency is problematic and developers should take further actions to mitigate the vulnerability.

3.1.4 Definition of the REM

The Ripple Effect of Metrics (REM) is a dependency graph; that is, a directed acyclic graph (DAG) where the root of the graph represents the application of which the visualization corresponds, other vertices are its dependencies, and there exists an

edge from vertex A to vertex B if A explicitly requires the dependency B.

The REM is formally defined as a directed graph consists of vertices V and edges E as:

$$REM = (V,\ E)$$

A vertex is either a dependency or the application being developed (this vertex is the root of the graph), therefore, vertices V is denoted as:

$$V = \{Application\}\ +\ Dependencies$$

For each vertex, there are three basic attributes:

1. An *identity* attribute that identifies the vertex. For example, a combination of name, version and its location in the dependency graph can be an unique identity since there should not be more than two vertices at the same location with the same version and name.

2. A *metric of health*, that is a numeric value normalized between 0 and 1. Higher value corresponds to better health.

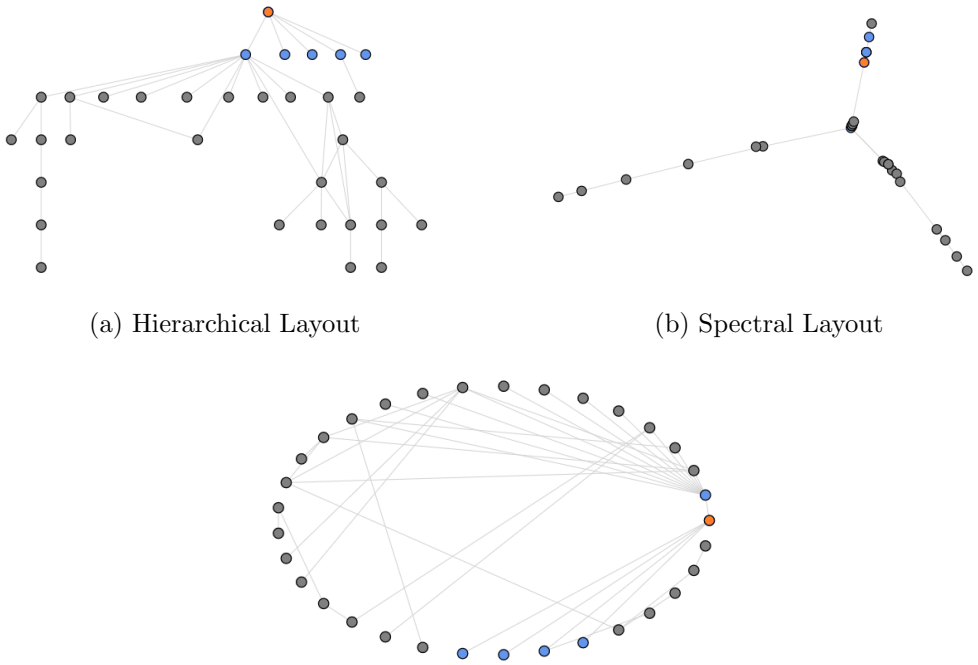
3. A *vulnerability metric*, that is a Boolean flag, when `true` it indicates that the dependency is vulnerable to a security risk.

An Edge is a pair of source and destination vertices representing that the source vertex requires the destination vertex. For each edge, there is a *type* attribute to identify the type of the dependency relationship. In this thesis, the type of edge can be either runtime (required when the application is running) or development (required when the application is being developed) to represent the different uses of the dependency relationship.

In the REM, the ripple effect of a vulnerable dependency vertex is a set of paths (i.e. set of groups of edges) between that vulnerable dependency and the application being developed (the root of the graph). The ripple effect is calculated using the vulnerable metric (a flag) of the dependency vertices, such that when there is a vulnerable dependency vertex, the paths, represented by groups of edges, are added to the ripple effect.

3.2 Rendering the REM

The REM is rendered using a hierarchical layout such that the root of the graph is at the first level, and upper-level vertices represent dependencies that require those to which they are connected in lower-level vertices. We compared popular graph layouts such as circular(Fig. 3.2c) and spectral layout(Fig. 3.2b), and we argue that the hierarchical layout(Fig. 3.2a) is a more effective way to visualize the dependency graph because it effectively shows dependency relationship, given that the main objective of the REM is to provide an intuitive visualization to help developers to understand and explore dependencies in the dependency graph.



(a) Hierarchical Layout

(b) Spectral Layout

(c) Circular Layout

Figure 3.2: Different layouts tested on an sample NPM application, where the orange vertex is the application, blue vertices are the direct dependencies, and grey vertices are the transitive dependencies

Vertices of the REM are annotated to highlight the health of their corresponding dependencies, and edges are annotated according to the ripple effect of vulnerable dependencies. The root of the graph is an exception since it has neither of the metrics, thus it should have a fixed colour that distinguishes itself from its dependencies. In the REM, the vertices are visually annotated in three ways:

1. Uses *metric of health* of a vertex to be the its filling colour. The colour scheme used to fill the vertices should be corresponding to their *metrics of health*.

2. The colour of the outline of vertex is according to their *vulnerability metric*. Since the vulnerability metric is a Boolean value, the colour choice on the outline must be from two different colours.

3. By default, the circle size is the same for every vertex across the dependency graph (as shown in the Figure. 3.3). However, the size of the vertex can vary (the implementation in this thesis used different sizes to highlight some of the vertices, examples will be shown in later sections). Since some of the dependencies play important roles (such as direct dependencies, and vulnerable dependencies) in a dependency graph, in the REM, the size of the vertex can be made larger for those that are direct dependencies or vulnerable dependencies so they are more obvious in the dependency graph.

And edges are also visually annotated:

1. Developers use dependencies for different situations. This is reflected as having different types of dependencies, such as runtime and development dependencies. To visually distinguish the different types of dependencies, they are displayed using different line thickness for their corresponding edges.

2. Edges can have different colours based on whether they are parts of the *ripple effect*. Whether or not an edge is in the ripple effect is a Boolean value, therefore, the colour choices are from two different colours. This value determines an edge being part of the ripple effect from a vulnerable dependency, the colour must be obvious so the users can distinguish from others.

Fig. 3.3 shows how a REM looks visually. First, there are two types of dependencies, development and runtime dependencies, each type has been described previously. The development dependencies are connected with thicker edges, and runtime dependencies are connected with thinner edges (this is also shown in the interactive buttons located at the bottom left corner in the figure). From the figure, the dependencies' filling-colour corresponds to the value of their metric of health, except for the software application being developed, which is shown as the orange circle. The range of values of the metric is normalized; the gradient colour-scale varies from red to green, representing the value range from the lowest to the highest, respectively. In Fig. 3.3,

dependency L3 and its transitive dependencies, dependency T1 and T2, are having lower values than dependency L1 and L2 based on the their filling-colours and a legend on the right-hand side of the figure. Therefore, low values in the metric of health in dependency T1 and T2 suggesting that there is a risk for the application being developed to depend on them. Fig. 3.3 also shows a problematic dependency; dependency T1 is considered vulnerable for having a red outline-colour with a thicker ring that is triggered by the vulnerability metric. Since dependency T1 is a problematic dependency, it creates a ripple effect on the dependency graph of the application A. Dependency T1, T2, L3, and the application A are all involved in the ripple effect; vertex A is the software application, T1 is the vulnerable dependency, T2 directly depends on vulnerable dependency T1, and dependency L3 is also the direct dependency of software application A that depends on T2. This ripple effect is visually represented by the dependency T1 annotated with red outline-colour, and every path (i.e. the edges) from dependency T1 to application A is annotated with dark-red colour.

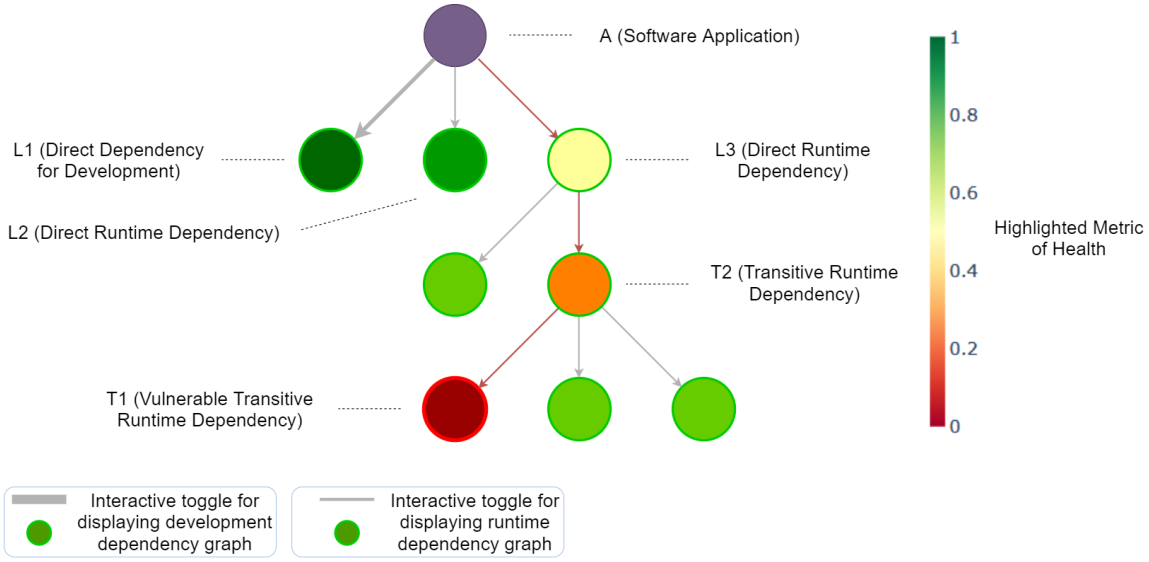


Figure 3.3: A dependency graph visualization example of the REM based on the original graph from Fig. 1.4

3.3 Graph Filtering Algorithms

It is not uncommon for large applications to have dependency graphs with dozens (if not hundreds) of vertices. Not only are the graphs large in terms of size, but also

in terms of complexity. This section describes two filtering algorithms used on the REM that attempt to address this issue. After applying the two filtering algorithm, the filtered version of the REM becomes a variant of the REM in which dependencies are *collapsed* (i.e. removed from the visualization) and *grayed-out* (i.e. had filling-colour removed from the visualization). Throughout this section, an example REM of a software application (Figure.3.4, Figure.3.5, Figure.3.6) are used to demonstrate how the two algorithms work to highlight the risky dependencies, which are transitive dependency B and direct dependency A. This example is the REM of a real world application that is small enough to clarify the filtering process as easy to understand as possible.

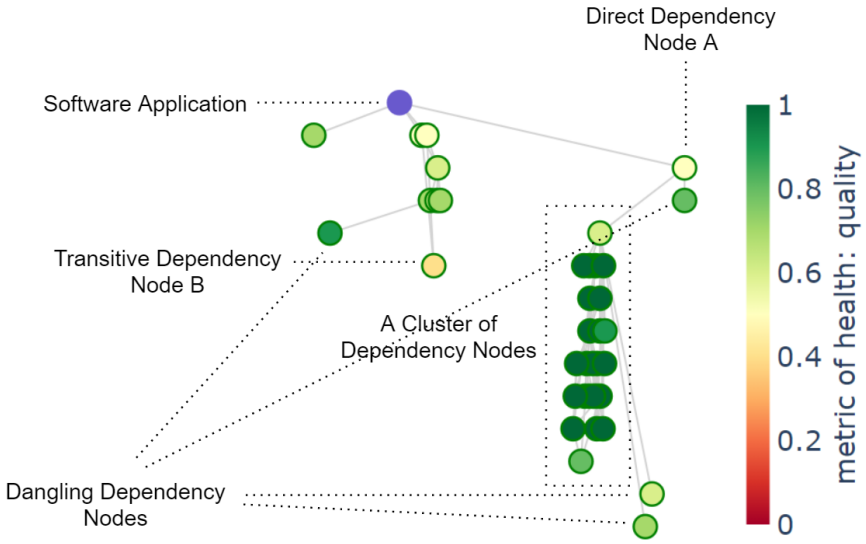


Figure 3.4: The REM graph with NPM **quality** score as metric of health and **deprecation status** as vulnerability metric. Important vertices and area are marked manually for explanation purpose

3.3.1 Collapsing Algorithm

The collapsing algorithm focuses on folding vertices that are considered healthier than their parent vertices (according to their metric of health). And after the healthy vertices are collapsed, those dependencies that are riskier than others become obvious to users in a such reduced graph. Fig. 3.5 is the filtered version of the REM (Fig. 3.4) where some of the vertices are being collapsed. The collapsing algorithm makes few important changes to the original graph. First, it removes five transitive dependency

from the left child vertex of the root except for the only leaf (vertex B), resulting in the highlight on the less-healthier vertex B with light-orange filling-colour. Then the algorithm helps to reduce the cluster's density of transitive dependency on the right-hand side of the graph marked as a box (dotted area). Note that the cluster of green vertices did not get collapsed; this is because of the very last leaf vertex with lighter-green colour (as being less healthier than all of its parent vertices). Last changes made by the algorithm are the removals of the dangling dependency vertices marked in Fig. 3.5. In the collapsed REM, shape of the vertices with at least one child collapsed will be replaced from the ◯ symbol to the ⨁ symbol.

Algorithm. 1 describes the algorithm for the collapsing the graph as described above. A vertex N that is a direct dependency of the other vertex M is collapsed if the following conditions are satisfied:

1. vertex N must be a transitive dependency. That is, vertex N is neither the software application nor the direct dependencies of the software application,

2. there is no ripple effect between vertex N and vertex M. That is, when traversing from vertex M to N, there should be no edges that are in the ripple effect.

3. and the metric of health of vertex M is lower than the metric of health of all the descendent dependency vertices that are reachable from vertex N, including vertex N itself.

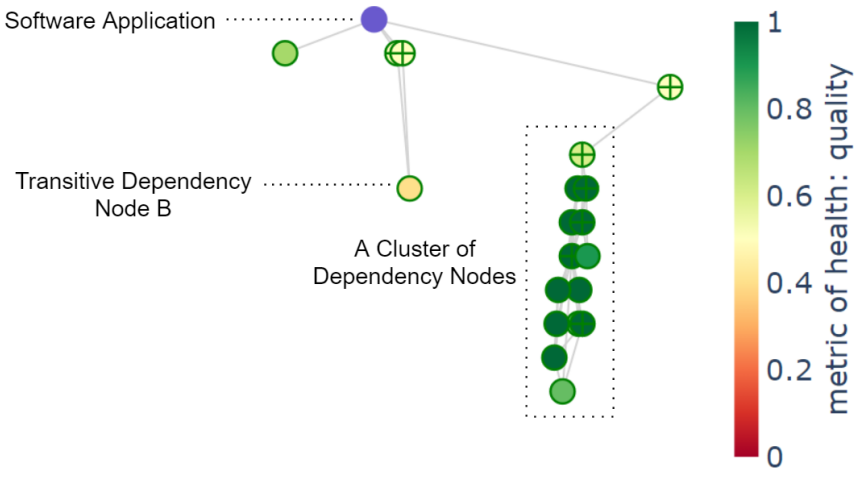


Figure 3.5: The REM graph after applying *Collapsing* algorithm on Fig. 3.4

Algorithm 1: Collapsing Algorithm

Input: G: DiGraph, *ripples*: List of Edges

Output: DiGraph

Result: a reduced Directed-Graph

```
visited ← {ni: False for every vertex ni ∈ G};
# a FIFO queue for post-order tree traversal
# push in every vertex with out degree equals to 0
queue ← [ni for every leaf vertex ni ∈ G];
while len(queue) > 0 do
    vertex ← queue.pop();
    if visited[vertex] == True OR vertex == G.root then
        continue;
    end
    visited[vertex] ← True;
    # loop through every predecessor vertex of vertex
    foreach predecessor_vertex ∈ G.predecessors(vertex) do
        if Edge(predecessor_vertex, vertex) ∉ ripples AND
         vertex.metric_of_health ≥ predecessor_vertex.metric_of_health then
            # collapse the edge if predecessor_vertex and vertex are not in
             ripple effect and vertex is healthier than predecessor_vertex
            G.collapse_edge(predecessor_vertex, vertex);
        end
        queue.push(predecessor_vertex);
    end
end
return G;
```

3.3.2 Grey-out Algorithm

As shown in Fig. 3.5, the size of the graph is reduced with dangling vertices removed and cluster density reduced by applying the collapsing algorithm described above when compare to the original REM in Fig. 3.4. It is now obvious that Vertex B is risky. But vertex A, however, is still hiding inside of the graph. This is because that the cluster of vertices contains many seemly healthy dependencies (those with green filling-colour), and becomes noise to the identification process of the possible risky vertices (vertex A, in this case). To further bring user's attention to focus on the direct dependency A in a cluster of vertices, the graph shown in Fig. 3.5 must be further reduced using the Grey-out algorithm. Fig. 3.6 is the filtered version of the graph in Fig. 3.5 that had the grey-out algorithm applied. From the greyed-out graph in Fig. 3.6, when the colours of the vertices in the cluster are dimmed (i.e. with filling-colour removed and became the outline-colour), the direct dependency A that connects the cluster is more obvious than before with a light-yellow filling-colour. In greyed-out REM, the colour of the dimmed vertices will be removed and the the outer ring colours will be changed by the metric of health.

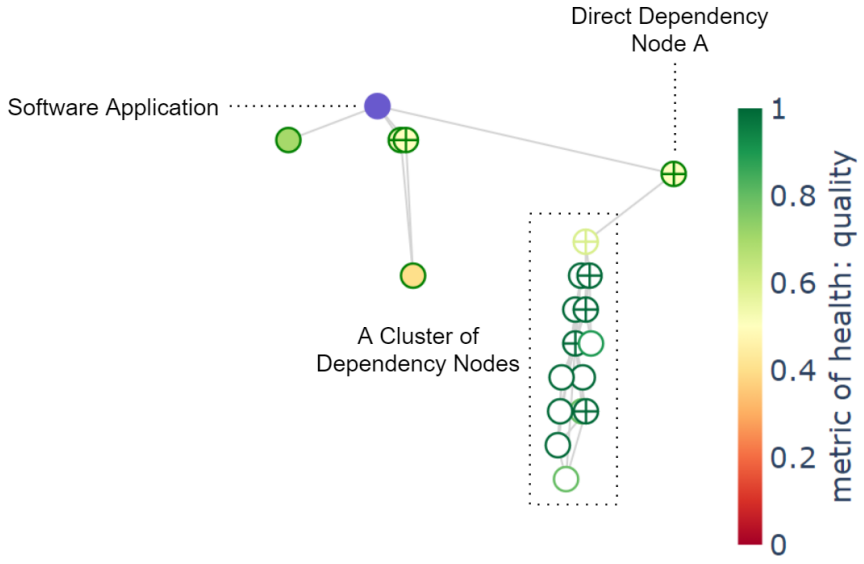


Figure 3.6: REM graph after applying *Grey-out* on Fig. 3.5

In general, the grey-out algorithm helps in highlighting the risky dependency vertices at a higher level of the hierarchical graph by dimming the colour of the child vertices healthier than their parent vertices. In other words, vertex N will be dimmed

(grayed-out) if its metric of health is better than the metric of health of all of its ancestor vertices that have an explicit connection with the application (i.e. the direct dependencies of the root of the dependency graph). An algorithm is provided in Algorithm. 2 with more details.

Algorithm 2: Grey-out Algorithm

Input: G: DiGraph

Result: a modified Directed-Graph with healthy vertices marked greyed-out

```
foreach vertex ∈ G do
    # check if vertex is root of the graph G or a direct dependency of the root
    if vertex == G.root() OR G.exists(Edge(G.root, vertex)) == True then
        continue;
    end
    # check if vertex is problematic (determined by the vulnerability metric)
    if G.vertices(vertex).re_metric == True then
        continue;
    end
    # a metric of health list of the vertex's direct dependency predecessors
    direct_dependency_predecessors_metrics ← [ni.metric_of_health for every ni
     ∈ G that has a path to vertex AND G.exists(Edge(G.root, ni)) ==
     True];
    if len(direct_dependency_predecessors_metrics) ≠ 0 AND
     vertex.metric_of_health ≥ MAX(direct_dependency_predecessors_metrics)
     then
        # mark healthy vertex in G as greyed_out
        vertex.greyed_out ← True;
    end
end
```

Chapter 4

Implementation Details

This chapter discussed the NPM-REM, an implementation of the REM that focuses on NPM software applications. The implementation includes the general tools used, descriptions of what and how data are collected and used, and the rendering process of the dependency graph of the REM.

The NPM-REM uses a wide range of tools. First, the NPM-REM is written in Python (version 3.7) scripts. In addition, Python libraries and tools such as Networkx[1], Plotly[2], and GraphViz[3] are used to help build and render the visualization of the REM. The NPM-REM can also be containerized and ran on Ubuntu using Docker to allow redistribution on any environment regardless of the operating system. The implementation will be discussed in details later in this chapter.

4.1 Building the Dependency Graph of the REM

The REM is a dependency graph, thus in the implementation of the NPM-REM, it is built using the dependency information of the NPM software application being developed (the root of the graph) and NPM packages data, so that its dependencies are vertices and the dependency relationships are links (edges) connecting the software application and vertices. Figure. 4.1 is the overview of the rendering process of the REM which describes that a REM is rendered from the data prepared and extracted from different sources. Specifically, there are two steps to build the REM:

1. **Data in advance**. The data include NPM packages data that store dependency

[1]https://networkx.github.io/
[2]https://plotly.com/python/
[3]https://www.graphviz.org/

information for every NPM package, and health data that store NPM score metrics and CVE security data. These data are used to build the dependency graph and to help in rendering the REM, and are prepared before the rendering process of the REM.

2. **Render the REM**. Once the data are ready, the data will be used in the rendering process of the dependency graph in the REM. The REM renderer will first create the dependency graph from the dependencies information for the dependency file extracted from the software application. And then, the REM is rendered using the health data and saved as a HTML file.

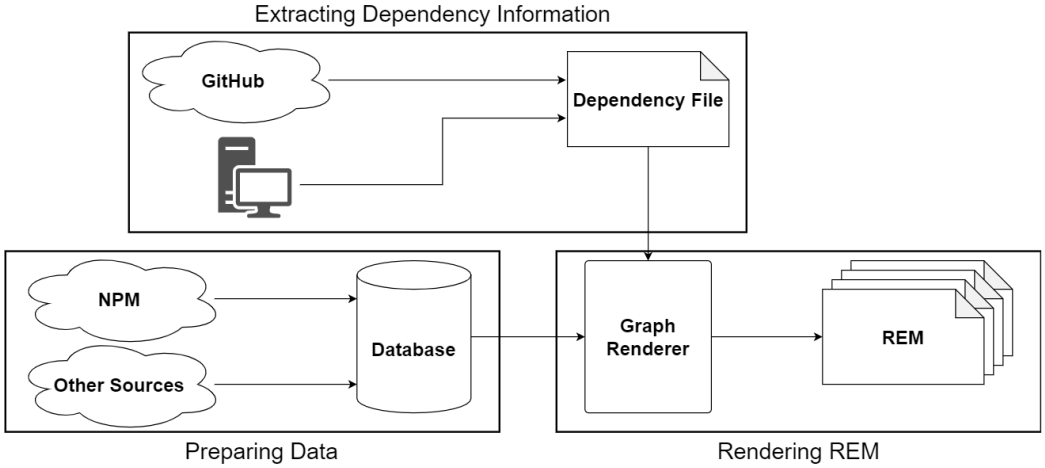


Figure 4.1: REM graph rendering process overview, including preparation of the data, extraction of the dependency file, and the rendering

The dependencies data are required to build the dependency graph, and two methods of extracting dependencies data are discussed in this thesis.

First one is to build a dependency graph from a list of dependencies that the application directly used, and for each dependency package, extract dependency information from NPM, and repeat until all dependencies have no further dependencies. This requires NPM packages data which can be further processed and stored in a a database with the dependency information. Researchers [21, 31] have been obtaining the NPM packages data via its public API endpoint[4]. However, NPM announced in 2016 that the API registry endpoint that queries all NPM metadata was no longer supported[5]. In this thesis, the NPM packages data are retrieved from an alternative

[4]registry.npmjs.org/-/all

[5]https://blog.npmjs.org/post/157615772423/deprecating-the-all-registry-endpoint

domain, namely the replicate registry[6], which is similar to the public NPM registry and is hosted on CouchDB[7] by the NPM official team. API *GET /db/_all_docs view* is executed with *include_docs=true* (a built-in CouchDB feature) enabled and successfully extracted a recent list of NPM packages' *package.json*-like metadata. As shown in Fig. 4.2, the NPM metadata for every NPM package are stored in a *Packages* table in the database, and these NPM packages data collected will then be processed to extract only the dependency information and store in a *Depend* table. Dependency information of the software application being developed can be retrieved from GitHub or directly from a local directory. For NPM applications, *package.json* dependency file of the target application contains a list of direct dependencies in the *dependencies* and *devDependencies* fields, which represent *runtime* and *development* dependencies, respectively. Once a list of direct dependencies is obtained, transitive dependencies and dependency relationships (links connecting dependencies) can be extracted from the database.

Second method to create the dependency graph is to retrieve the dependency information directly from the lockfile, an auto-generated dependency file that contains a full dependency graph, each dependency with a specified version. More details on the second method will be discussed in the next chapter.

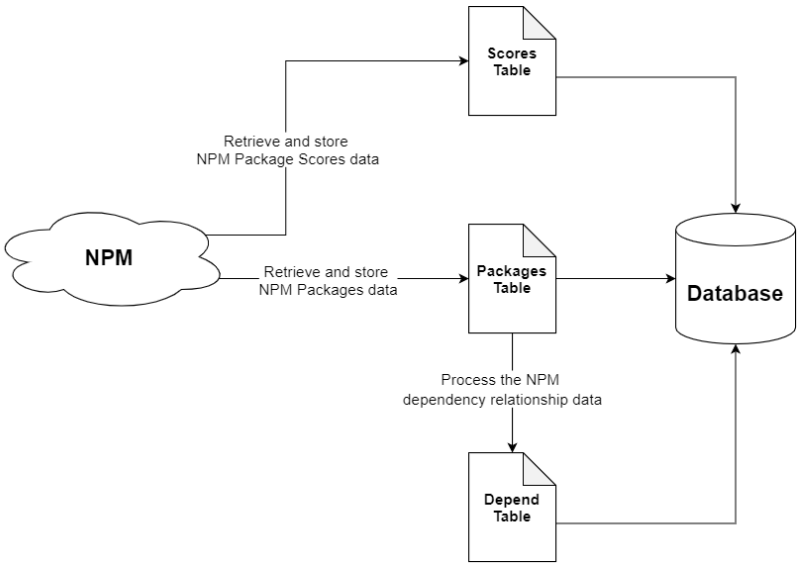


Figure 4.2: REM graph data preparation process overview, data collected from NPM contain packages data and scores (health metrics)

[6]http://replicate.npmjs.com/
[7]https://couchdb.apache.org/

The dependency information from both the software application's *package.json* and dependency data (such as data in *Depend* table or from the lockfile) are then used to render the dependency graph of the REM with additional packages information, shown as the Graph Rendering section in Fig. 4.1. The layout of the graph is a hierarchical layout from Graphviz dot program[8] that is directly accessible in the Networkx. The REM graph is rendered by the Plotly graphing library to create HTML files that can be opened by any modern web browser. The user interface that provides navigation, inspection and filters for the graph was also included in the generated HTML file by Plotly.

4.2 Metrics of Health


```
{
    "package": {
        "name": "lodash",
        ...
    },
    "score": {
        "final": 0.6331612105444097,
        "detail": {
            "quality": 0.6064945303873757,
            "popularity": 0.980478613943922,
            "maintenance": 0.3087009615652125
        }
    },
    "searchScore": 100000.69
},
```


Figure 4.3: Package scores data retrieved for *lodash* using NPM registry API, as of February 2021

Metrics of health are used to annotate the vertices in the dependency graph of the REM. Such a metric can be any numerical value (in the visualization, the range of a metric is normalized to fit between a range of colours). This thesis utilized four package *score metrics* from NPM to represent the metrics of health of NPM packages (as described in Section. 3.1.1 Chapter. 3, these were final, popularity, quality, and maintenance). These metrics are computed by NPM and are provided to the users as

[8]https://www.graphviz.org/pdf/dotguide.pdf

a method to rank and evaluate each software package. The score metrics are collected using the NPM public registry API[9] (see Figure. 4.3 as an example of the retrieved data), and these score metrics are stored in a different table (the *Scores* table in Fig. 4.2) in the SQLite database.

Another metric used in this thesis is the vulnerability metric. This is a metric that determines whether a dependency is vulnerable with existing problems. In this thesis, CVE security advisory is used as the vulnerability metric, since dependencies with known CVEs are exposed to the security risk.

[9]https://github.com/npm/registry/blob/master/docs/REGISTRY-API.md

Chapter 5

Dependabot with REM

This chapter describes the modification details on how to adapt REM graphs in the open-source Dependabot. The modification includes changes on the Dependabot core components, scripts that runs Dependabot, and architectural changes made to NPM-REM so that it can hosts publicly accessible web server. In addition, an example of the REM-Dependabot's creations is provided to should how it works on individual dependency-update pull request and on the application's dependency graph.

Chapter. 1 Section. 1.2 states the fundamental challenges with using dependencies in a large dependency graph are to identify what and how a dependency is used, and identify the propagation of unhealthy dependency. In this thesis, a modified version of the open-source Dependabot, namely REM-Dependabot, is introduced to provide developers an overview of the vulnerable dependencies and their ripple effect onto the software application.

Dependabot is one of the popular dependency management tools open-source development teams use in their software applications' development process to keep the dependencies staying healthy. Dependabot is built into GitHub and with its basic features freely accessible to public GitHub software repository. The main features of Dependabot is introduced in Chapter. 2 that it detects and notifies to users with the available update regarding any obsolete dependency.

The REM-Dependabot takes the advantage from open-source Dependabot for its existing functions on fetching and updating dependencies while enhancing the dependency management experience with the REM visualizations. By showing a REM graph in the dependency-update pull request that the Dependabot creates, the vulnerable dependency and its ripple effect onto the application will be highlighted and brought up to the user's attention. The REM graph helps developers identify and

understand the propagation of the vulnerable dependency. Furthermore, the REM-Dependabot also provides a set of REM graphs for developers to understand and assess the health of dependency graph from different aspects of the development.

This chapter contains the details regarding the REM integration and its implementation. The implementation includes the modification on the open-source Dependabot core components[1], the script[2] to run the Dependabot, and architectural and visual design changes made to the NPM-REM.

5.1 Integrating REM into Dependabot

REM-Dependabot enhances the open-source Dependabot by providing the following two main features:

1. REM-Dependabot adapts the open-source Dependabot to include a customized REM dependency graph to visualize the ripple effect of the vulnerable transitive dependencies in the dependency-security-update pull requests that it creates.

2. When a vulnerable dependency found in the dependency graph of the application being developed, REM-Dependabot creates an "issue" in GitHub that include a set of REM dependency graphs visualizing the metrics of health of project's dependency graph using four NPM's search scores(final, popularity, quality, and maintenance) and ripple effects of CVE security vulnerability.

5.1.1 Integration

Figure. 5.1 is a high-level flow to show the steps that REM-Dependabot follows to create solutions for dependency management with the REM graph. The features of the REM-Dependabot are added upon on the original high-level flow graph from the open-source Dependabot[5]. REM-Dependabot adapts functions of the original open-source Dependabot to include REM graph. REM-Dependabot can be fit in a continuous integration (CI) workflow for automatic monitoring and resolving potential dependency vulnerabilities. The REM-Dependabot runs in a regular basis (such as daily or weekly) to scan the Common Vulnerabilities Exposure (CVE) from GitHub database for each of the dependencies. The dependency files are directly fetched

[1]https://github.com/dependabot/dependabot-core
[2]https://github.com/dependabot/dependabot-script

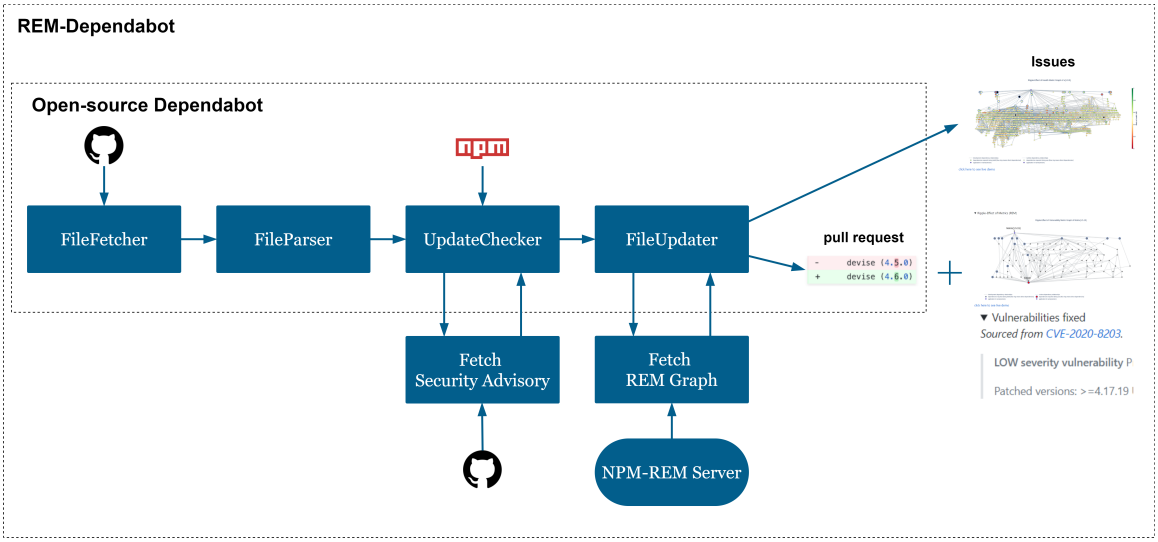


Figure 5.1: High-level flow of the REM-Dependabot built upon the original flow for open-source Dependabot

from GitHub repository. Once a CVE found on a dependency, REM-Dependabot will check with NPM for possible safer version that has CVE fixed. REM-Dependabot will then create a pull request with the dependency files updated accordingly. The pull request regarding the vulnerable dependency contains a REM graph fetched from the NPM-REM server to show the ripple effect of the vulnerability. In the CI workflow, REM Dependabot will also monitor the changes made to the dependency files, and will create a report in the GitHub's issues section showing REM graph with NPM scores as metric of health for the latest commit.

REM-Dependabot focuses on fixing security vulnerabilities, therefore, a section is added to the content of the pull request to contain the detailed information about the vulnerability exposures that the update pull request aims to resolve. Figure. 5.2 shows different parts of the vulnerabilities section. This section includes a list of known CVE with information such as the severity level, vulnerability description, affected and patched versions so that it is clear to understand how the current version of the dependency is being affected and which version or a range of versions patches the CVE.

To show the ripple effect that the vulnerable dependency has on the software application being developed, a REM section is added to the pull request content. The REM section contains a preview image of a REM graph customized in a way to only show the ripple effect of the updating dependency. In addition, an URL link to

```
- Vulnerabilities fixed
Sourced from [CVE source link]

    [vulnerability severity level] severity vulnerability [vulnerability description]
    Patched versions: [newer versions that have vulnerability fixed]
    Unaffected versions: [older versions that is unaffected by the vulnerability]
    Affected versions: [versions that is affected by the vulnerability]
```

Figure 5.2: A sample template of Vulnerabilities fixed Section in the pull request

the live view web page is also provided along with the image. Figure. 5.3 shows an example of the REM section. The REM graph shown in the REM section of the pull request is a modified version of the REM where the vulnerable updating dependency is annotated in red, application is in purple and every other dependency is in grey colour. The color of the edges is changed to blue for runtime dependency relationships and light-grey for development ones and they won't be changed to dark-red due to the fact that the whole graph is the ripple effect. To distinguish the direct dependencies from the graph, direct dependency nodes are annotated with a bigger size circle with the ring color changed to blue (example shown in Figure. 5.3).

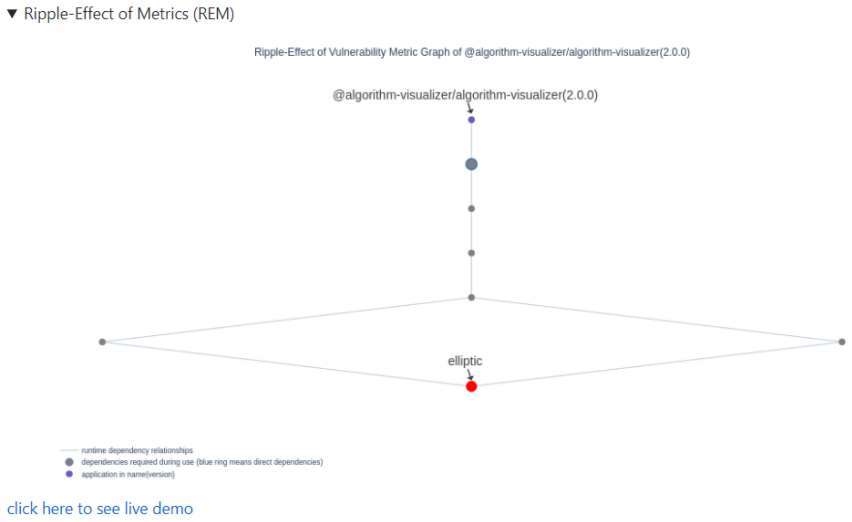


Figure 5.3: Example of a REM Section in the pull request created by REM-Dependabot showing the ripple effect that a vulnerable dependency, `elliptic`, has on the application, `algorithm-visualizer`

5.1.2 Implementation

Figure. 5.4 shows the protocols and tools used for communication among GitHub, NPM, REM-Dependabot and within the NPM-REM web server. REM-Dependabot uses same HTTP to communicate to NPM for checking on the available dependency versions and Octokit that communicates to GitHub to retrieve dependency files and manipulate the pull request as the original Dependabot. Furthermore, REM-Dependabot uses GraphQL to query the CVE data, and uses HTTP to send and receive data for the REM graph.

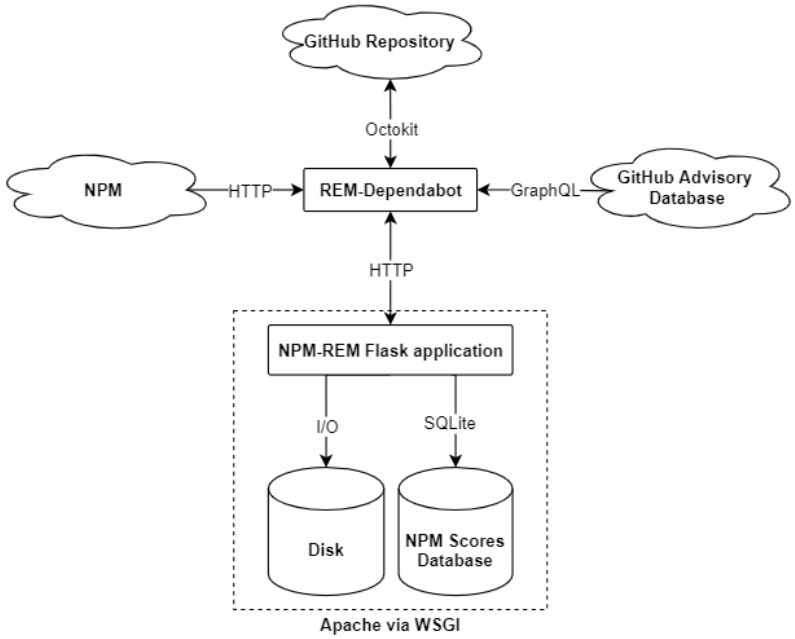


Figure 5.4: Communication between REM-Dependabot and NPM-REM web server

The process that turns the open-source Dependabot into the REM-Dependabot includes two parts, 1) customization on the source code of the Dependabot to have a new section for REM graph, 2) and a server that hosts NPM-REM for generating links to the image and live view web page for the pull request. The open-source Dependabot is modified in a way to send configured HTTP request with packed data to the REM server (more details will be discussed on the server end later) and use the returning REM graph URL links to create a section that shows a REM graph. The REM-Dependabot, similar to open-source Dependabot, requires a a series of operations to run, whether it is on a script or on by other CI tools. The set of operations can be found on an original script in a separate GitHub repository created by Dependabot

team and is adjusted in this prototype to contain a series of operations described in Figure. 5.1 that runs REM-Dependabot to check and update dependencies through the creation of the pull request and also to create the vulnerability report that containing the REM. The script is designed to run through every dependency on the dependencies list retrieved from the dependency files located at a designated directory of the target GitHub repository. The communication between Dependabot and GitHub repository is via the Octokit client. Instead of checking the latest compatible version for every dependency, the script is adapted to check the CVE advisory from the GitHub Security Advisory database using GraphQL API provided by GitHub. An example query is shown in Fig. 5.5 and one of its results is shown in Fig. 5.6 to illustrate the security data REM-Dependabot uses. REM-Dependabot then seeks available updates against the CVE security advisory and generates the pull request with the update according to `vulnerable_version_range` shown in Fig. 5.5. To add extra sections to the pull request content, the original open-source Dependabot is modified to accept the option for having REM graph. The code that makes HTTP request are added to the Dependabot so that it can send the request and receive the URLs as response for preview and live view of the REM graph when the REM option is enabled.

```
{
  example: securityVulnerabilities(
  ecosystem: NPM, package: "lodash", first: 100) {
    vulnerabilities: nodes {
      vulnerable_version_range: vulnerableVersionRange
      advisory {
        identifiers {
          type: type
          id: value
        }
        summary
        references {
          url
        }
      }
    }
  }
}
```

Figure 5.5: Example of a GraphQL query that retrieves first 100 vulnerabilities for `lodash` in NPM

REM-Dependabot fetches graphs of the REM from NPM-REM web server when detects a CVE for a dependency. In the NPM-REM server end, the Apache is chosen

```
{
    "vulnerable_version_range": "< 4.17.19",
    "advisory": {
        "identifiers": [
            {
              "type": "GHSA",
              "id": "GHSA-p6mc-m468-83gw"
            },
            {
              "type": "CVE",
              "id": "CVE-2020-8203"
            }
        ],
        "summary": "Prototype Pollution in lodash",
        "references": [
            {
              "url": "https://nvd.nist.gov/vuln/detail/CVE-2020-8203"
            },
            {
              "url": "https://github.com/advisories/GHSA-p6mc-m468-83gw"
            }
        ]
    }
}
```


Figure 5.6: First data from GraphQL result of the query in Fig. 5.5

due to the speed, reliability and the security that it provides. The procedures of the communication and operations in the NPM-REM web server can be described as:

1. Apache (the server) receives HTTP requests from REM-Dependabot for generating the REM used for pull requests or report depending on the requesting API and parameters.

2. The request is then sent to the NPM-REM Flask application through WSGI.

3. When REM web server receives the request, it will use the data that passed along with the request to generate REM graphs and store them to disk locations with public access for both preview image and live view HTML.

4. Once the response is generated, it is sent back to REM-Dependabot in JSON format that includes the public URLs data for easy process.

One major change made to the NPM-REM web server is that it creates the dependency graph according to the lockfile data (the second method) instead of using the dependency data collected to SQLite database (the first method) as the two different methods of creating the dependency graph described in Chapter. 4. The

lockfile method ensures the accuracy of the dependency graph to match the exact one as when the project gets built (for example, running command `npm build`) because NPM builds the project application base on the dependencies defined in lockfile prior to looking at package.json file by default. The NPM score metrics, however, are collected from the NPM registry to a SQLite database.

5.2 Examples

Similar to GitHub's version of Dependabot, the REM-Dependabot is designed to run automatically with configurable settings available in the GitHub. Before setting up the REM-Dependabot, a web server needs to be set up properly to host NPM-REM. The server is designed to receive the HTTP requests and generate REM graphs according to the request and store them in the directory that can be publicly accessed from GitHub. The REM-Dependabot is built into a Docker container to provide accessible environment for CI tools to use. Once there is a possible fix for a dependency's CVE advisory, the REM-Dependabot will create an individual branch with the dependency files modified and create a new pull request suggesting to merge this branch. Figure. 5.7 is an example of the pull requests created by the REM-Dependabot that suggests maintainers to update `lodash` dependency from version 4.17.11 to the safe version 4.17.20. The pull request contains the vulnerabilities that come from the older versions with severity level and CVE description. As shown in the figure, a REM section containing a customized REM graph (along with the link to browse the interactive view) is included to show how the `lodash` with version 4.17.11 propagates the vulnerability up to the application. The pull request usually contains other helpful information such as a history of commits to show the changes made to the updating dependency version, the release note regarding the updating version, and maintainer changes if there is any.

When detecting a vulnerability or any changes in a dependency file (via commit), REM-Dependabot will create a report in the GitHub issues section to provide the outlined vulnerabilities and a current dependency health view. Figure. 5.8 is an example of such report created by the REM-Dependabot. The report consists of few elements, first one is to list all the dependencies (whether direct or transitive) that have vulnerability based on CVE in GHSA. Then, there will be a message identifies which commit the REM graph is built upon. Once it has all the information above, a preview image of the REM graph is displayed to show a general overview of the

dependency graph. Link to the REMs are included where the user can interact with the graph. Figure. 5.9 is an example of such interactive REM graph by from the REM in the Figure. 5.8 by clicking the link to the live view. These interactive graphs of the REM are created by the NPM-REM and stored in a publicly accessible space in the same Apache server that hosts NPM-REM. The link redirects the user to a HTML with interactions and choices such as NPM metrics, filtering, and showing development or runtime dependencies in a dependency graph. The user can choose from four collected NPM score metrics described previously as the metric of health. For each NPM score, the user can also choose between two versions of graph, one of which is a full size dependency graph and another is a filtered version with reduced graph. And the user can see only development dependencies or runtime dependencies by toggling the legends at the bottom left corner of the web page.

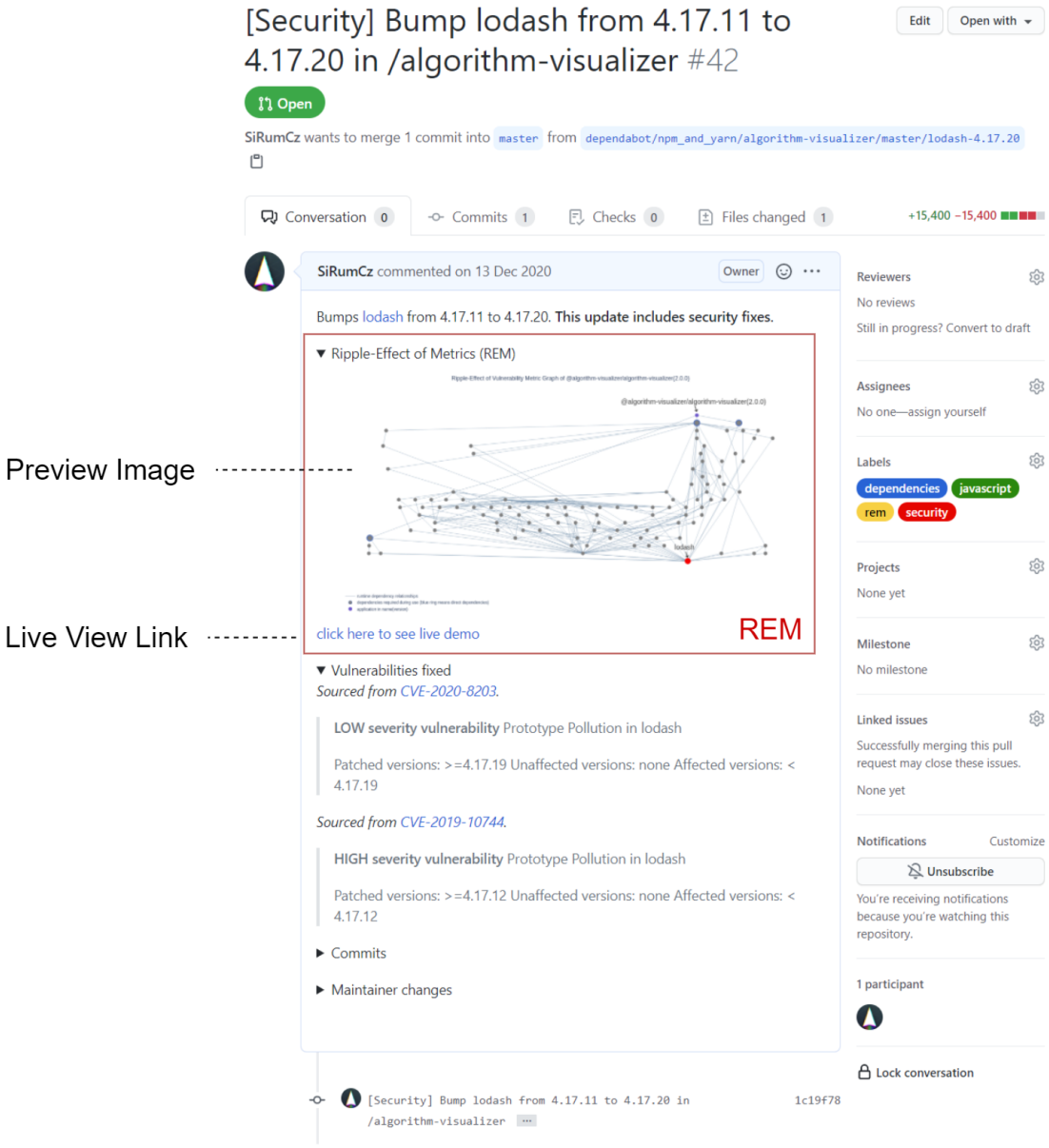


Figure 5.7: Example of a pull request created by REM-Dependabot showing a security update on dependency `lodash` in an application, a customized REM is included in the first section with a preview image and a link to an interactive web page

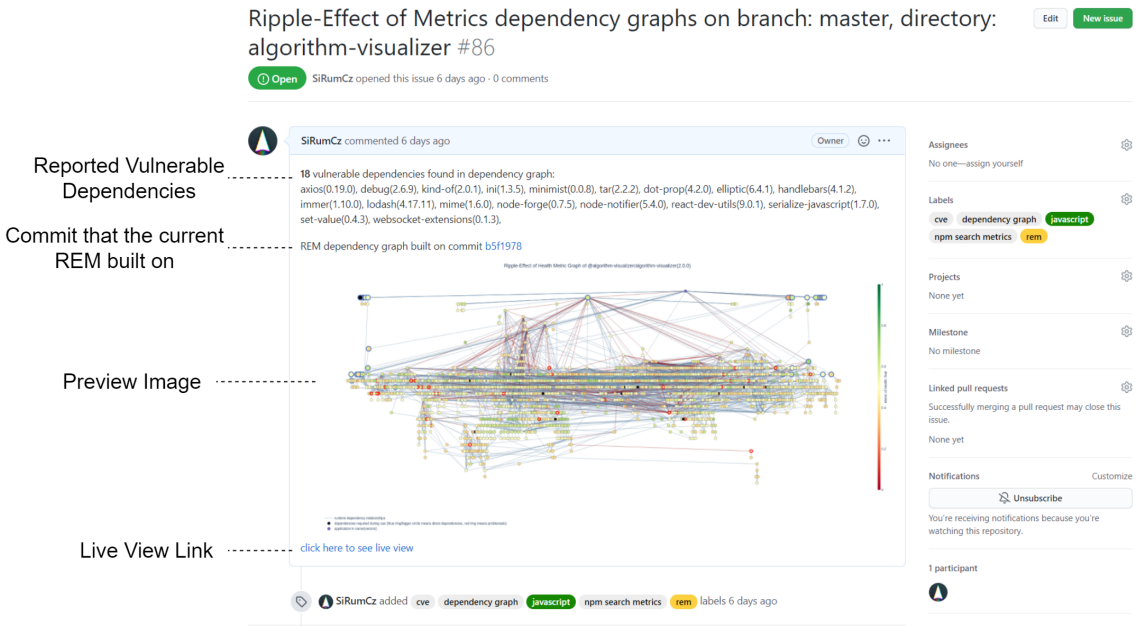


Figure 5.8: Example of a report created by REM-Dependabot under the GitHub's issues section showing a dependency graph of an application with a list of vulnerable dependencies

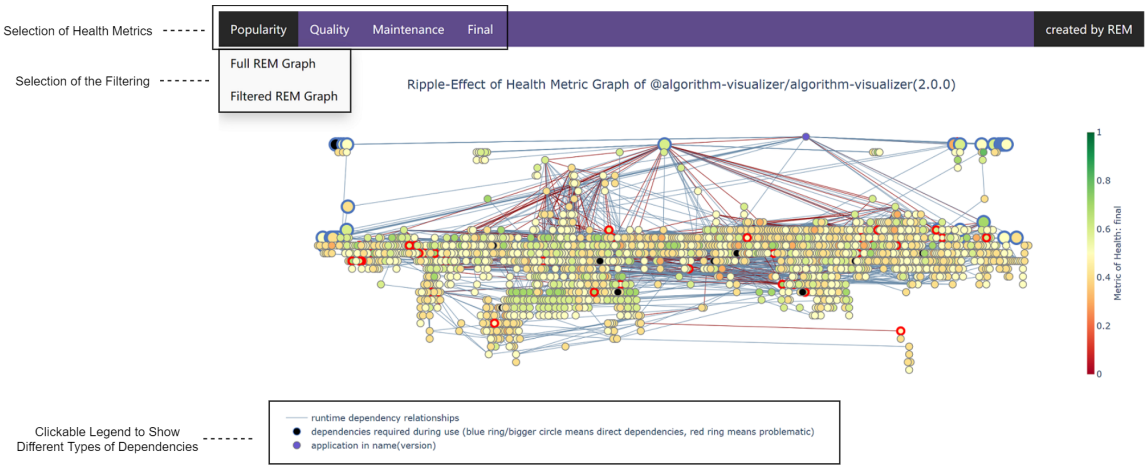


Figure 5.9: Interactive view of the example in Figure. 5.8, it can be opened by clicking the "click here to see live view" link shown in examples of pull request and issues

Chapter 6

Case Study

This chapter contains a case study using a chosen NPM software application from a process of selection which will be discussed later. This case study argues that the REM can effectively help developers maintain their dependencies by showing two use cases and comparing the scenarios with and without the presence of the REM (the NPM-REM and REM-Dependabot as described previously).

Vulnerable dependencies can be hidden inside of dependency chain and cause health problem to the software application by silently exposing the application to security risks. The goal of the REM is to help maintainers (developers who focus on the maintenance of the application) understand and manage dependencies more effectively by identifying vulnerable dependencies (both direct and transitive) and the ripple effect of those vulnerable dependencies have on the software application. This chapter presents 3 use cases; each shows its use in different situations to achieve a certain goal in dependency management. Together, they demonstrate the use of the REM (the NPM-REM and the REM-Dependabot) in managing the dependencies of an NPM application that is hosted in GitHub. Each use case starts with a description of a goal and how commonly available tools are used to complete the goal, then a description of how the REMs can be used instead followed by a comparison between the two methods.

6.1 Selection Process of the NPM Application

The REM-Dependabot was run on a selected application from the most popular GitHub JavaScript software applications that use NPM as their dependency package

manager. GitHub data was used in the selection process of the application is from GHTorrent[1], which is collected until the end of June 2020. 107,467 GitHub JavaScript repositories with star received from at least 1 user on GitHub were collected from the GitHub data. To make sure that each of these 107,467 repositories is indeed a software application and is using NPM dependencies, those that have no dependency (either don't use NPM or have 0 dependency) and are not software applications (10 out of 50 top starred JavaScript projects on GitHub are documentations or tutorials) are then removed from the list. From the top 15 projects, each project repository was manually inspected and *Motrix*[2] is selected to be used as the NPM application in the case study of the REM. *Motrix* is a full-featured download manager that supports downloading contents from many different types of source. The reason this specific application is chosen is because it is a popular open-source JavaScript software application hosted on GitHub with more than 25,000 stars (around 15 JavaScript repositories have received more 20,000 stars by June 2020, according to the collected GitHub data from GHTorrent) and more than 3,000 forks. *Motrix* uses NPM dependency management files that include the lockfile(i.e. `package-lock.json`), which is essential for the NPM-REM server to use to build the dependency graph accurately. And *Motrix* also has one of the most complicated dependency graphs of the popular NPM software applications. Therefore, it is included in this thesis as one of the representations of real-world NPM applications and an ideal application example in our case study.

6.2 Use Cases

This section describes three use cases in details how REM graphs can help in managing dependencies. Specifically, three use cases are:

1. **Report on vulnerabilities**. Before updating any dependency in an application, it is important to have a way to be aware of the dependency update opportunity. REM-Dependabot maintains a report on the dependency graph of NPM software applications regarding CVE vulnerabilities. With this report, maintainers of software applications can quickly see the full dependency graph of their applications and gain the awareness on dependency vulnerabilities in it.

2. **Browsing the REM data to identify vulnerabilities**. To have a closer

[1]https://ghtorrent.org/
[2]https://github.com/agalwood/Motrix

look of a vulnerable dependency, the REM-Dependabot includes the REMs in a vulnerability report for maintainers to browse. The REMs utilizes health metrics from NPM and the filtering techniques described in previous chapters. By browsing the REMs, maintainers can quickly navigate and identify vulnerable dependencies.

3. **Updating a vulnerable dependency**. When a vulnerable dependency can be updated, Dependabot creates a pull request that helps maintainers to update that dependency to a safe version. Inside the pull request, REM-Dependabot includes a REM that shows the ripple effect of that vulnerable dependency to help maintainers understand the impact of that dependency on their applications, and thus, make a better decision on whether they should update that vulnerable dependency.

6.2.1 Use Case 1 - Report on vulnerabilities

Since modern software are being developed and updated rapidly, it is a challenging task for maintainers to keep up with the latest version of the software dependencies that their software applications are using. Maintainers of the software applications would keep their dependencies up to date so that the applications are less prone to vulnerabilities. However, updating dependencies can sometimes be time consuming for maintainers if they manually check on their dependencies. When maintainers look for vulnerable dependencies to update (especially when they are transitive dependencies), a general report on the status of the dependencies can be useful for them to understand their dependencies and make a better decisions such as whether a dependency needs to be updated.

When developing NPM software applications, developers often use commands from NPM for different tasks, such as build and install a project, and run the application in different modes. In NPM, the `npm audit`[3] command helps maintainers assess dependencies' security vulnerability and provides suggested solutions to the vulnerabilities. The most common scenario is when maintainers build a NPM project. When `npm install` is run, a security audit is also ran automatically and it will produce a report containing list of vulnerability information on every detected vulnerable dependency, each with its name, vulnerability severity, description, and path. The report generated by `npm audit` is shown directly in the terminal (it also can provide

[3]https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities

a JSON file). Screenshot on the left side of the Figure. 6.1 is a fraction of the report taken from a complete report generated by *npm audit* on the example application *Motrix*, and a text block extracted from the report on the right of the figure shows the vulnerability details. The report is detailed with information on dependencies that are vulnerable and NPM commands that can be used to update them.

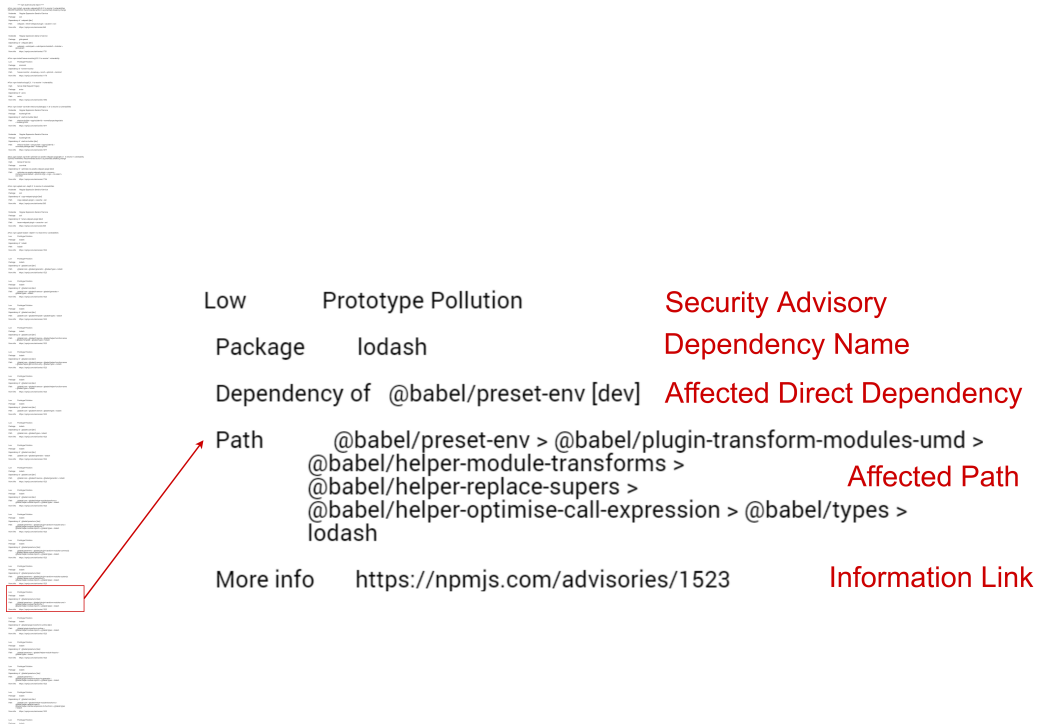


Figure 6.1: A fraction (600 lines out of 11,000 lines) of the report generated by NPM audit is present due to the size of a complete report, one vulnerability in dependency *lodash* is highlighted with annotated explanation

REM-Dependabot creates and maintains a vulnerability report of the dependency graph for software application. As described in the previous chapter, this report consists of two parts: a list of vulnerable dependencies found in dependency graph, and the REMs showing the ripple effects of those vulnerable dependencies including an entry to the REMs with interactive features. Figure. 6.2 is a report generated by REM-Dependabot for our example NPM application *Motrix*. In this report, dependencies with CVE security risks are listed along with their versions so that they can be referred back to database to see the actual CVE information such as description details and severity. Followed by the list is the commit id (an unique metadata used by Git to keep track of a version of the application built on a specific timestamp) that

indicates which version of the dependency files that the dependency graph is based on. Then, an image of the REM showing the dependency graph with dependencies coloured according to their metric of health (in this case, the metric of health is set to NPM *final* metric), and the ripple effects of the vulnerable dependencies. Since it is a preview image that only gives a general idea of what the REM of the example application looks. For example, a dependency (shown in Figure. 6.2) immediately stands out from this dependency graph for creating a huge ripple effect in the application's dependency graph (surrounded by many dark-red edges). To be able to look closer at this dependency or to browse the REMs, an entry to the interactive REMs is provided at the end of the report (which will be discussed in the next use case).

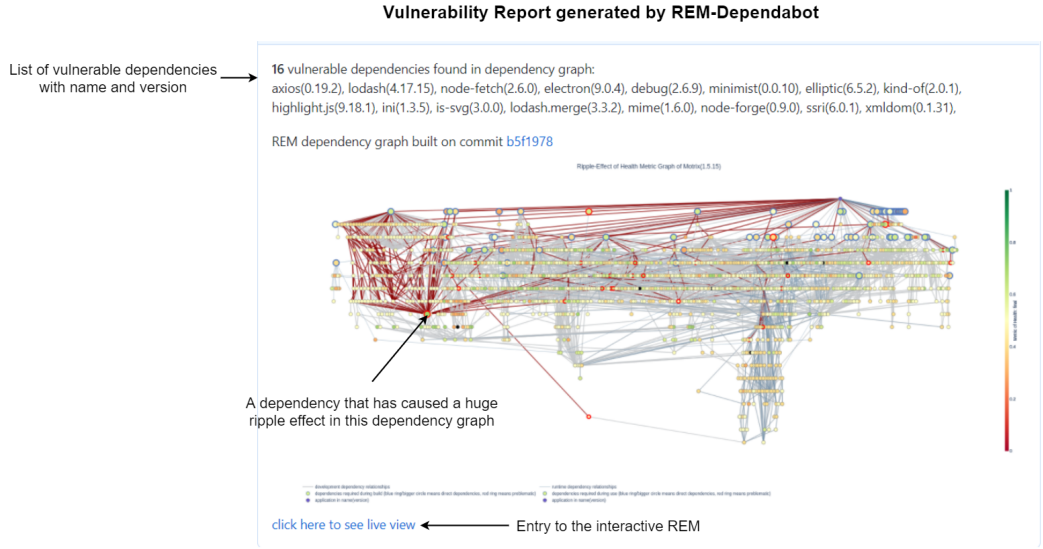


Figure 6.2: Vulnerability report created by REM-Dependabot that contains a list of vulnerable dependencies and the REM including a preview image and entry to the interactive REMs, a dependency is highlighted for causing a huge ripple effect

The dependency vulnerability report from REM-Dependabot is created and maintained in the Issues section in GitHub, a section where developers communicate regarding the current project (such as requesting new features or to raise questions on problems they have encountered). GitHub Issues is also a common place where active users will raise questions on dependencies, such as reporting dependency failures, or posting vulnerability concerns. Having the dependency vulnerability report in this place can bring the dependency problem to the attention of developers and ensures maintainers to always access the same version of the vulnerability list and the REM graphs.

Both vulnerability reports are generated at the same commit of the application *Motrix*. The report from NPM can become very large and overwhelm maintainers; sometimes the report can exceed the maximum size that a terminal can display (for example, Windows command prompt can only show up to 9,999 lines). However, with the report from REM-Dependabot, the dependencies that have vulnerability problems are exposed directly in the list and the general idea of how the dependency graph is affected is shown in an image of the REM on the application.

6.2.2 Use Case 2 - Browsing the REM to explore the impact of vulnerabilities in dependencies

In Use Case 1, a dependency was highlighted for having caused a huge ripple effect in the dependency graph of the *Motrix* shown in Figure.6.2. Therefore, Use Case 2's task for maintainers is to take a closer look at the dependency graph and investigate a vulnerable dependency and explore the impact of this dependency on the dependency graph of the *Motrix* .

From the report created by *npm audit* in Figure. 6.1, the report is filled with text blocks (shown in the left side of the figure), each text block contains one vulnerable dependency and a chain of other dependencies that it affects (shown in the right side of the figure), which is similar to the ripple effect in the REM. The report created by *npm audit* does not visually highlight the impact of any dependencies, providing no ranks of dependencies in terms of vulnerability severity. Therefore, to know which vulnerable dependencies are having more negative impacts on the application than others, maintainers need to manually scan a whole text report to explore the dependencies and their impacts. Also, to investigate a vulnerable dependency, dependency *lodash* for example, maintainers will go over the long list of report, find the information (as stored in text blocks) about *lodash*, and extract each information and combine them to understand the impact of the vulnerability.

In the report created by REM-Dependabot, the dependency graph of the REM is included and created based on the dependency graph of the application *Motrix* using NPM *final* as the metric of health (the filling-colour of dependencies shows their health) and CVE security advisory as the vulnerability metric (the outline-colour of dependencies shows the whether a dependency is vulnerable and the colour of edges represents the ripple effect of those vulnerable dependencies). To browse this REM and other REMs that have been created with different health metrics and with

the filtering, users need to go to a web page included in the report created by the REM-Dependabot.

In the report created by REM-Dependabot, by clicking the link in the report in Figure. 6.2, maintainers can browse and navigate in a REM with many interactive features shown in Figure. 6.3. From the web page developers can look at that highlighted vulnerable dependency described in Figure.6.2 using the interactive features, such as zoom, selection of different types of dependencies, and tooltip. Figure. 6.4 is a part of the zoomed REM in Figure. 6.3 that shows the parts of the dependency graph that is affected (through its ripple effect) by that vulnerable dependency, and from the tooltip which contains every information on a dependency, the indentity of the highlighted dependency described in Figure. 6.2 is revealed to be the dependency *lodash*. By zooming in the dependency graph in Figure. 6.3, maintainers can look close to the highlighted vulnerable dependency. Since the REM allows users to switch to different types of dependencies for different purposes. By looking at the dependency file(*package.json*), *lodash* is one of the direct dependencies that are required during runtime. However, in the REM graph of *Motrix*, most of the edges in the ripple effect are connecting development dependencies (as they are thicker lines), meaning the vulnerable dependency *lodash* is also used during the development stage and hugely affecting the application with its CVE vulnerabilities. In Figure. 6.4, maintainers can only see the dependency graph for the development of *Motrix* as shown.

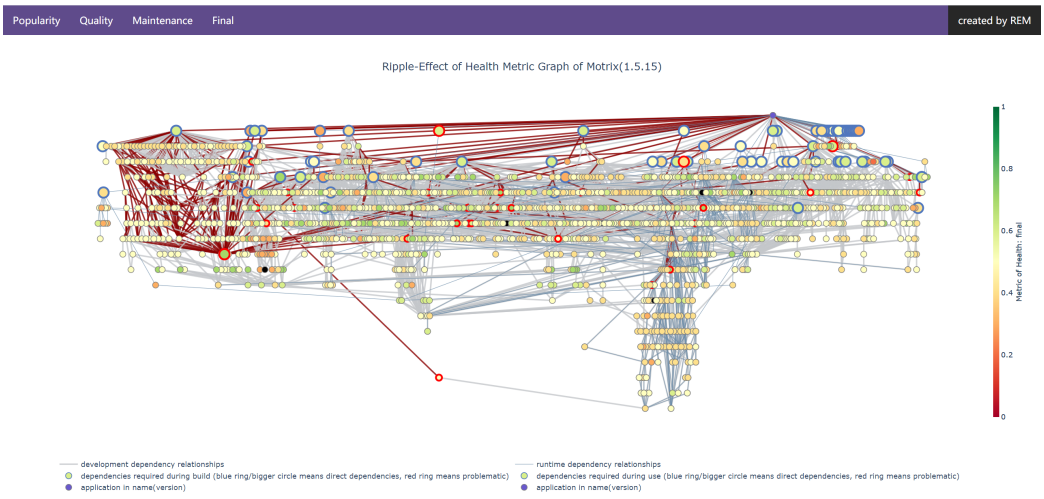


Figure 6.3: The REM web page from the url provided in the report for *Motrix*

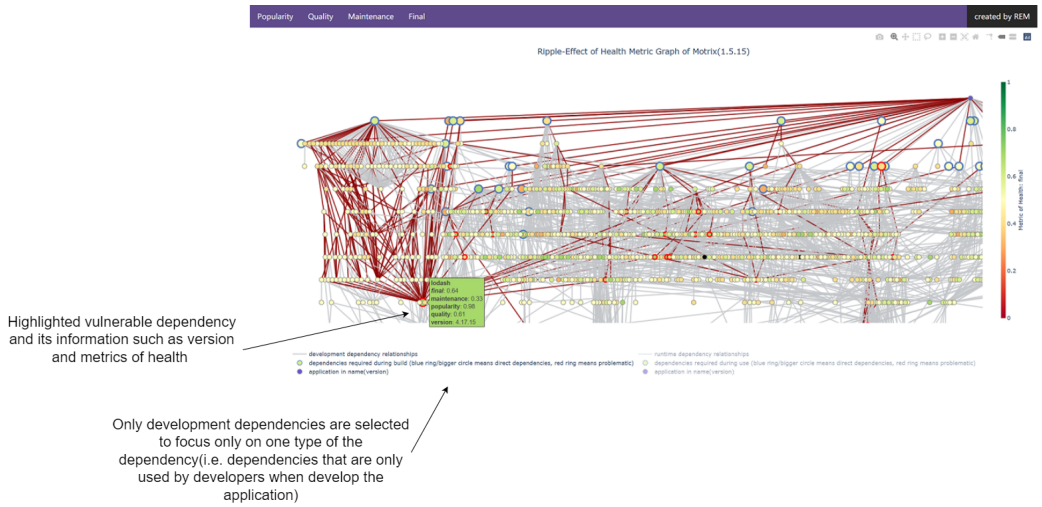


Figure 6.4: A zoomed REM of the *Motrix* with only development dependencies selected and vulnerable dependency *lodash* highlighted

Both reports created by NPM audit and REM-Dependabot reveal a list of vulnerable dependencies and their impact on the dependency graph of *Motrix*. However, from NPM audit report, since information on vulnerable dependencies and their ripple effects on the application is across the whole report, maintainers of *Motrix* can be overwhelmed by the long size of the audit report and fail to understand the dependency graph of the application *Motrix* and the ripple effects of those vulnerable dependencies. Therefore, maintainers will find it hard to know how a vulnerable dependency (*lodash* for example) is negatively affecting application *Motrix*. With the visual benefits brought by the REM, maintainers are able to see that *lodash* is having a strong impact in the dependency graph of *Motrix* and identify the direct dependencies that is affected by the ripple effect of *lodash* with less effort.

6.2.3 Use Case 3 - Updating a vulnerable dependency

In the third Use Case, after maintainers of *Motrix* have investigated the dependency graph of *Motrix* and spotted a vulnerable dependency *lodash*, the next goal is to update this dependency in order to fix the vulnerability issues in the dependency graph of their software application.

In the report created by *npm audit*, when a vulnerable dependency is detected, an NPM command that updates that vulnerable dependency will also be included in the report if possible. Therefore, when looking for dependency *lodash*, an NPM command is provided: `# Run npm update lodash --depth 9 to resolve 612 vulnerabilities`.

By running this command, maintainers perform an update for *lodash* that is hidden as deep as 9th level in the dependency graph of *Motrix*. However, with this method, the dependency will be updated immediately and maintainers will not know any information regarding the updated version of the vulnerable dependency such as changelog, compatibility, and maintainers changes.

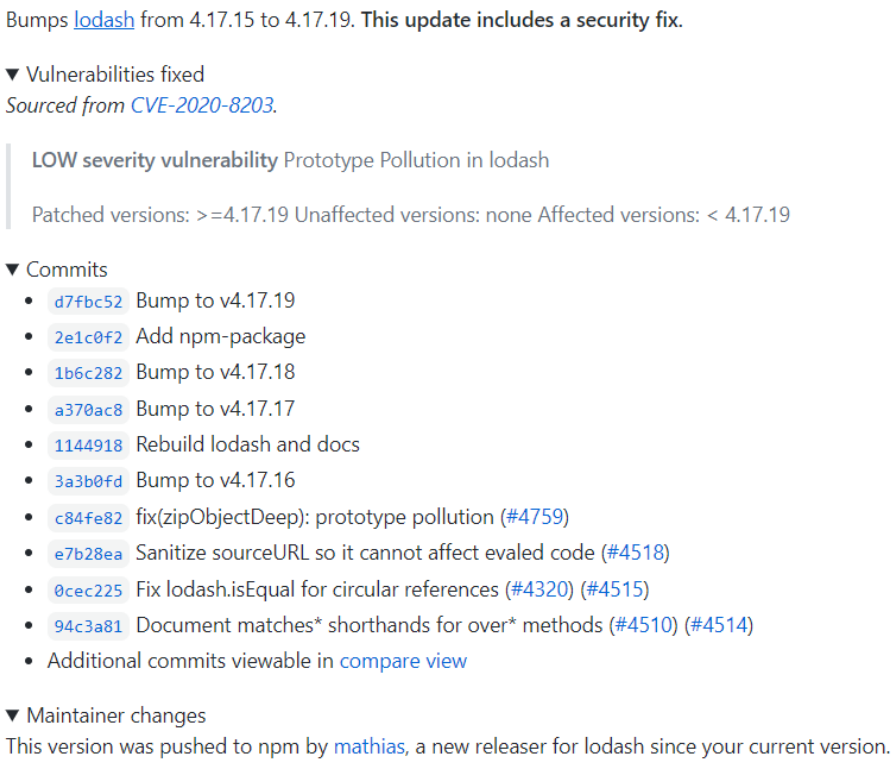


Figure 6.5: The pull request that updates *lodash* in application *Motrix* created by open-source Dependabot, without the REM

Updating dependencies using the NPM command from *npm audit* report does not provide maintainers the opportunity to review, and updating dependency without reviewing is risky as there are chances that it can break the application with other problems such as incompatibility problem. As a plugin to GitHub, Dependabot offers a feature to securely update vulnerable dependencies via GitHub pull requests. The pull request enables the development team to review the dependency update before the changes being made to their source code. This type of the pull request also includes information regarding the updated version of the vulnerable dependency, such as the vulnerability information, commit histories of the new version, changlogs,

and maintainers information.

The REM-Dependabot enhances the pull request created by the original Dependabot by adding a REM showing ripple effects of the updating vulnerable dependency on the application being developed. Figure. 6.5 and Figure. 6.6 show the pull request that the open-source Dependabot and REM-Dependabot created respectively to update vulnerable version of `lodash` in the application *Motrix*.

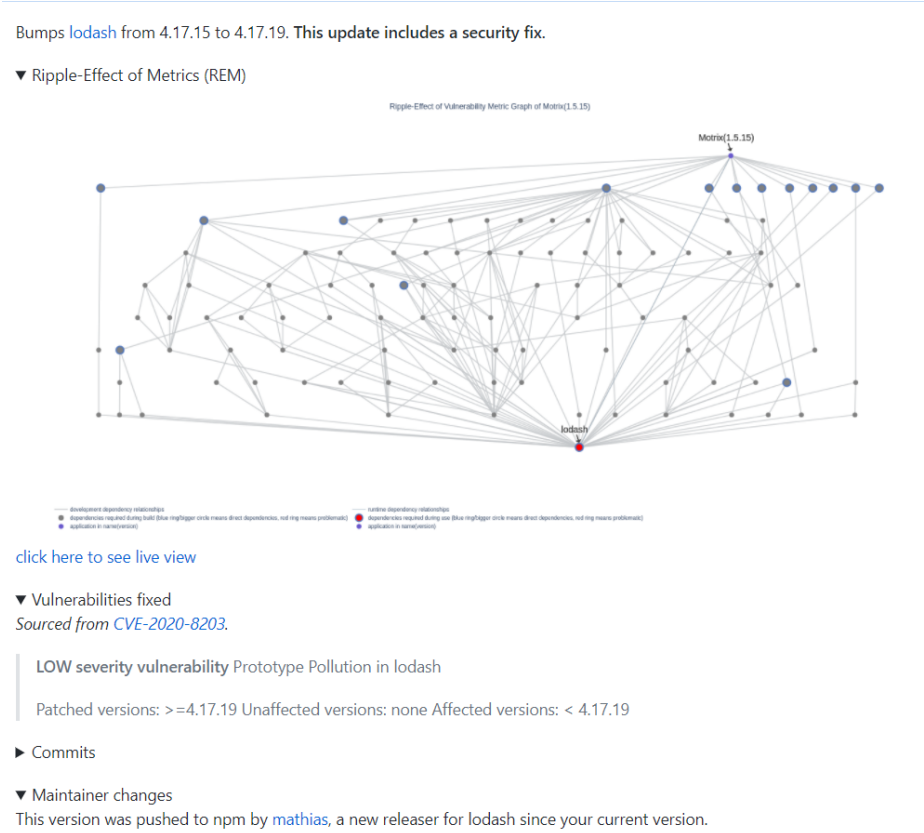


Figure 6.6: The pull request that updates *lodash* in application *Motrix* created by REM-Dependabot, with the REM

In a typical GitHub pull request, a section named *Files changed* is included for the user to browse any changes that the author wants to make to a repository. Therefore, if maintainers want to know where modified dependencies are located, they can go through the changes in files, as shown in Figure. 6.7, that is included in both pull requests created by the open-source Dependabot and REM-Dependabot. However, with the file changes, maintainers can find it hard to understand where that vulnerability is actually located and how it can affect their software application. This is

because file changes are only partially shown (in Figure. 6.7, only lines that changes occurred are shown as red and green blocks, other lines are folded automatically). In the Figure. 6.6, with the REM added to the pull request by the REM-Dependabot, the REM shows an overview of the ripple effects of the vulnerable dependency *lodash* have on the application *Motrix* on both direct and transitive dependencies. By removing every other dependencies not in the ripple effect of *lodash*, maintainers can focus on reviewing the vulnerable dependency and its ripple effect, and therefore, make a better decision to whether or not perform the update in the pull request. And by looking at the REM in the pull request in Figure. 6.6, maintainers can see that vulnerable dependency *lodash* is having negative impact on application in both development and runtime dependencies, which means, it can affect both development team (during the development stage of the application) and clients that use their application (when running the application). In addition, from the REM, there are 16 (including *lodash* itself) direct dependencies (dependency vertices with bigger size of circle) that are affected by the ripple effect of the vulnerable dependency *lodash*, suggesting maintainers should update this dependency as soon as possible as large number of direct dependencies are vulnerable as well. Then, by merging this pull request, maintainers has successfully performed an update to a vulnerable dependency.

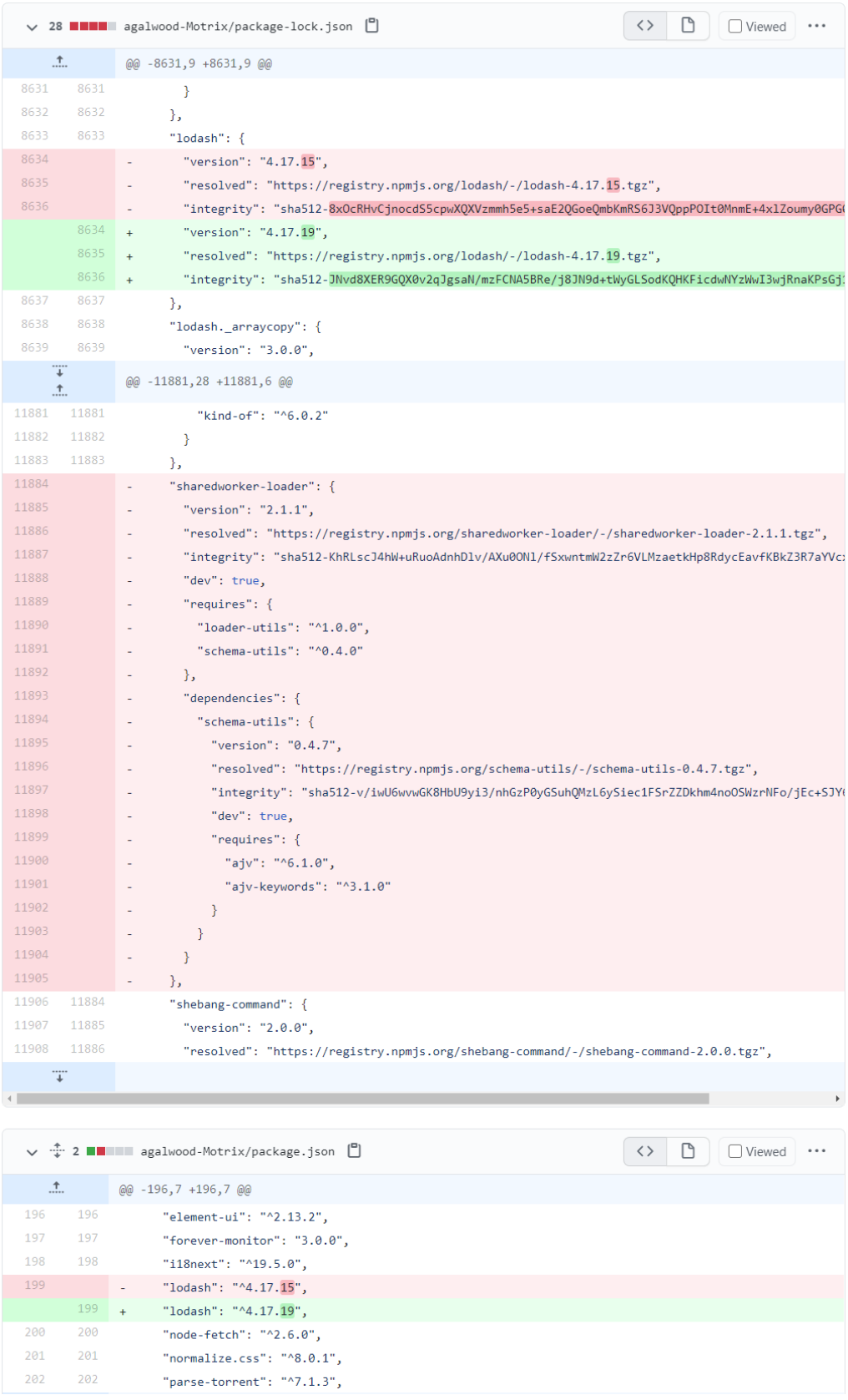


Figure 6.7: File changes shown in both pull requests in Figure. 6.5 and Figure. 6.6 for updating `lodash` from version 4.17.15 to 4.17.19 in *Motrix*

6.3 Summary

In this chapter, 3 use cases were used to demonstrate how the REMs can be useful to maintainers by reporting vulnerabilities in dependency graph, providing an opportunity to explore the impact of vulnerabilities, and providing an visual aid to highlight the impact of the updating dependency on the application.

With traditional tools, the vulnerability text report can get excessively long and overwhelm users, making it hard for them to comprehend the vulnerabilities and explore their impacts. REM-Dependabot creates the REMs with an overview of a dependency graph health in a dependency vulnerability report, and creates dependency update pull requests containing the highlight of the impact that each vulnerable dependency has on the application. By having a dependency graph with extra information on dependencies in the REM, maintainers can benefit from the visual overviews and make a better decision when performing updates on unhealthy dependencies.

In conclusion, through out the 3 use cases, having REM in each use case makes it easier for maintainers to identify vulnerable dependencies, explore their negative impacts on the dependency graph, and review the update on vulnerable dependency.

Chapter 7

Discussion and Conclusion

This chapter contains a further discussion on the different aspects of the REM, including the data that are used to build NPM-REM, the effectiveness of the REM, challenges with using the REM, future paths that can improve the research prototype of the NPM-REM and the REM-Dependabot, and a conclusion that concludes the thesis.

7.1 Threats to Validity

The main goal in this thesis is to design a visualization that helps reveal the vulnerabilities in the dependency graph of a software application. This section discusses the threats to the validity of the effectiveness of the REM.

The NPM-REM builds the dependency graph based on the dependency relationship specified in the runtime and development dependency fields that are either defined in a `package.json` file or a lockfile (for example, `package-lock.json`) of a software application. However, the current implementation of the REM can be biased to not accurately describe a dependency graph for a software application for several reasons:

1. *External dependencies can be hiding in source codes.* IN NPM, dependencies can be imported and used directly from source files. For example, developers can use an NPM dependency `find-me` stored in a local path in the project by simply adding `require('./lib/find-me')` in a file,

2. *Dependencies from private domains do not have public information and health*

metrics. Dependencies from non-NPM or private domains might have inaccurate or have no metrics of health,

3. *No further tracking on the dependencies defined in dependency files*. The REM identifies dependencies are being used in the application if they are defined in the dependency file. The REM does not track the dependency usage in the application, meaning it is uncertain in the REM whether an application is actually using the dependencies it includes.

Another threat in the implementation of the NPM-REM is that the health metrics from NPM do not vary from different versions of the same dependency package. In the implementation, the four NPM metrics (search scores) were collected at a certain timestamp for every NPM packages, and the data are also not bound to specific versions of those packages (instead, the metric only reflects the status of the latest package at the time the data were collected). Therefore, the accuracy of the metrics of health on dependencies can be outdated and become less valid given that the versions of dependencies in dependency graphs can be created at different times so they are different than their current versions.

7.2 Challenges

One of the the challenges in this thesis is the deployment of the REM. The implementation of the NPM-REM sourced different data from NPM. To deploy the NPM-REM on a machine and successfully use it, users need to download data from NPM on their own machines to collect NPM packages and metrics. In addition, users will need to maintain these data so that the REM will always generate accurate dependency graphs. The implementation includes a script that helps download and manage NPM metrics data. However, users would need to manually download NPM packages data, and data needs to be frequently updated.

Presenting a huge graph on a web page is an other challenge in this thesis. Because vertices and edges are two dimensional elements in the REM, the visibility of the edge on huge graphs can be difficult to see as they can overlap each other. Especially when developers try to inspect the dependency relationships among certain packages, they can get confused because of the overlapping edges in the graph visualization. Similarly, because vertices that are on the same depth of the graph will be horizontally aligned, it can be confusing with great amount of vertices on the same level of the

graph. In this thesis, two filtering algorithms have been developed to mitigate this issue by filtering down the number of vertices and edges of those that are considered less important. While use cases were provided in Chapter. 6, it has not been tested and evaluated by the developers in real world. The filtering is also considered a non lossless compression of the information because many information were discarded(by collapsing) during the process due to the limitation of the graphing library used.

7.3 Limitations

An accurate dependency graph builds upon reliable data sources. We implemented the NPM-REM described in Chapter. 4 based on the data collected from a database[1] containing NPM package metadata. However, this database is behind the official NPM database[2], which has more recent and accurate package metadata than what was used in this thesis. This is due to the removal of the API endpoints for large data retrieval in NPM database. Another source that contains package data is the libraries.io[3], which had been used as the data source by few researchers in their works [13, 33]. Although the data integrity on either of the data sources were not systematically tested, the data source from the official team should be more reliable and consistent than third-party platforms.

In REM-Dependabot described in Chapter. 5, the NPM-REM server requires a lockfile to create the dependency graph. Using the lockfile, NPM-REM can produce the most accurate dependency graph as the dependency management tool sees since the NPM will also use the lockfile to build the project if the lockfile is presented.

Another data used in this thesis is the dependency health metric. Four scores from NPM were used as examples of the metric of health in the REM. These four metrics, also known as NPM search rank criteria, are what developers see when searching for packages. However, community members often question the reliability of these scores[4]. Since this thesis focuses on the design of the visualization, the metrics used were not not being closely examined.

In this thesis, the graph visualization of the REM used a Python graphing library, Plotly, for its easy access to a wide range of graphing tools that helped in drawing

[1] https://replicate.npmjs.com/
[2] http://registry.npmjs.org/
[3] https://libraries.io/
[4] https://npm.community/t/package-search-scores-are-broken/10188/4

the REM with many features such as colour-scale, zoom feature, and HTML output option. However, Plotly does not have great support on graph. For example, Plotly does not support adding arrows to the edges to make a graph directed. It is difficult to improve on the current implementation of the REM to have new features without switching to other graphing libraries.

7.4 Future Work

In this thesis, the dependency graph visualization of the REM can be the first step to a better dependency management system. There are many routes to utilize and improve the REM.

First, as mentioned in the limitation section, in the NPM-REM, dependency information is extracted from the dependency management files; the REM does not check whether all of those dependencies were actually used in the code nor check source codes for dependencies imported directly from the code. Therefore, the current implementation of REM can be improved by finding a way to check dependencies that are explicitly used in the software, including those dependencies that are defined outside of dependency files or are imported and used in other source codes.

Second, the fast pace of the software development process requires developers to assess their dependencies every short period of time due to feature requests and security patches. Therefore, we would like to see the possibility to cooperate the REM with real-time data to deal with the data used in the NPM-REM that can quickly be outdated. To achieve this, a cost-efficient way needs to be investigated to keep NPM data updated and update the REM in real-time.

An other future work for this thesis is to thoroughly evaluate the usefulness of the REM. For example, the REM-Dependabot can be set up as a GitHub plugin and offer it to a wide range of developers to use it on their NPM software projects with a set of instructions and tasks, and analyze the the feedback collected upon the approval of developers.

Finally the REM focuses on showing different metrics on dependencies so that developers can view its dependencies' health and identify risks and vulnerabilities base on those metrics shown. An other research path is to use the REM on developing meaningful metrics that can effectively evaluate and represent the health of the dependencies whether in certain aspects or in general, and the REM would be able to help in such process of developing and evaluating a new metric.

7.5 Conclusion

Dependency management is a critical task for developers in the modern software development process. The average number of transitive dependencies and hidden layers in the dependency graph of a modern software project has increased dramatically in recent years. The challenge for developers and researchers has become to find the best practice to gain awareness of the transitive dependencies, their health, and potential impact of the vulnerable dependencies on the dependency graph of the software application.

This thesis presents the REM, namely Ripple Effect of Metrics, a dependency graph visualization that exposes the vulnerabilities of transitive dependencies by using the health metrics and showing the negative impact of vulnerable dependencies. The open-source Dependabot, an open-source automated dependency tool in GitHub, was enhanced with the REM to help maintainers explore the dependency health of the software application and identify the risks and vulnerabilities. While many tools are designed to provide developers with the opportunity to update dependency with different methods, none has successfully provided enough information to help maintainers and brought the problems with transitive dependencies to the developers' attention despite being a critical link to the dependency graph of a software application. Some automated dependency tools, such as the Dependabot, can detect vulnerabilities in transitive dependencies and provide update suggestions. However, they do not have an effective method to show the impact of the vulnerability on the application in its dependency graph.

The REM graph takes advantage of a hierarchical graph layout, which effectively shows the structure of the dependency graph and results in representing accurate dependency relationships. To make the graph visualization more readable, I created two filtering algorithms that help to reduce the size of the graph in terms of the number of nodes and edges to emphasize important dependency nodes.

With the idea of the REM in mind, I developed the NPM-REM server, which takes in specific HTTP requests and responses with web links to the HTML pages containing the REM graph. I modified the open-source Dependabot in GitHub to include the REM in its functionality to communicate with the server. This modified version of Dependabot is named REM-Dependabot. It provides a modified REM graph that focuses on the ripple effects for problematic dependency and provides a dependency vulnerability report with complete REM graphs for users to explore their

dependency problem with more detailed information and interactions.

The case study is also included in this thesis with three use cases to argue that the REM makes the dependency management process easier for maintainer. The three use cases forms a dependency management scenario where maintainers of an NPM software application will be provided with a dependency vulnerability report, then they will be exploring the REM to identify a vulnerability in the dependency graph, and finally, update the dependency identified.

The REM provides a meaningful way of visualizing the dependency graph by annotating the dependency nodes with different metrics and highlighting the rippled paths to emphasize the ripple effects of vulnerabilities. Due to the lack of evaluation from the real-world developers, the REM and its concept require further investigation. However, this thesis has demonstrated and argued in the case study section that the REM can support the automated Dependabot and provide maintainers the opportunity to explore the dependency graph with extra information in some scenarios where the dependency problem has not yet been resolved by the automated tools.

Bibliography

[1] 2020 open source security and risk analysis (ossra) report. https://www.synopsys.com/software-integrity/resources/analyst-reports/2020-open-source-security-risk-analysis.html. Accessed: 2021-01-26.

[2] 6th annual report on 2020 state of the software supply chain. https://www.sonatype.com/campaign/wp-2020-state-of-the-software-supply-chain-report. Accessed: 2021-01-20.

[3] Common vulnerabilities and exposures. https://cve.mitre.org/. Accessed: 2021-01-25.

[4] Community health analytics open source software. https://chaoss.community/. Accessed: 2021-05-25.

[5] Dependabot. https://dependabot.com/. Accessed: 2020-11-15.

[6] Github advisory database. https://github.com/advisories. Accessed: 2021-01-25.

[7] libraries.io. https://libraries.io/. Accessed: 2021-01-18.

[8] A. Bergel, S. Maass, S. Ducasse, and T. Girba. A domain-specific language for visualizing software dependencies as a graph. In *2014 Second IEEE Working Conference on Software Visualization*, pages 45–49, 2014.

[9] C. Bogart, C. Kästner, and J. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 86–89, 2015.

[10] Zhe Chen and Daniel M. German. Rem: Visualizing the ripple effect on dependencies using metrics of health. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 61–71, 2020.

[11] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser. Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 109–118, 2015.

[12] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 181–191, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Softw. Engg.*, 24(1):381–416, February 2019.

[14] Martin Dias, Diego Orellana, Santiago Vidal, Leonel Merino, and Alexandre Bergel. Evaluating a visual approach for understanding javascript source code. ICPC '20, page 128–138, New York, NY, USA, 2020. Association for Computing Machinery.

[15] R. Falke, R. Klein, R. Koschke, and J. Quante. The dominance tree in visualizing software dependencies. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6, 2005.

[16] D. M. German, B. Adams, and A. E. Hassan. The evolution of the r software ecosystem. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 243–252, 2013.

[17] D. M. German, J. M. Gonzalez-Barahona, and G. Robles. A model to understand the building and running inter-dependencies of software. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 140–149, 2007.

[18] Daniel M. German. Using software distributions to understand the relationship among free and open source software projects. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, page 24, USA, 2007. IEEE Computer Society.

[19] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[20] K. E. Isaacs and T. Gamblin. Preserving command line workflow for a package management system using ascii dag visualization. *IEEE Transactions on Visualization and Computer Graphics*, 25(9):2804–2820, 2019.

[21] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, page 102–112. IEEE Press, 2017.

[22] R. G. Kula, C. De Roover, D. German, T. Ishio, and K. Inoue. Visualizing the evolution of systems and their library dependencies. In *2014 Second IEEE Working Conference on Software Visualization*, pages 127–136, 2014.

[23] R. G. Kula, C. De Roover, D. M. German, T. Ishio, and K. Inoue. A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 288–299, 2018.

[24] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, Feb 2018.

[25] Mircea Lungu, Michele Lanza, Tudor Gîrba, and Romain Robbes. The small project observatory: Visualizing software ecosystems. *Science of Computer Programming*, 75(4):264 – 275, 2010. Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).

[26] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 84–94. IEEE Press, 2017.

[27] S. Raemaekers, A. van Deursen, and J. Visser. The maven repository dataset of metrics, changes, and dependencies. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 221–224, 2013.

[28] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation? the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[29] M. Shahin, M. Ali Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[30] B. Todorov, R. G. Kula, T. Ishio, and K. Inoue. Sol mantra: Visualizing update opportunities based on library coexistence. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 129–133, 2017.

[31] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 351–361, New York, NY, USA, 2016. Association for Computing Machinery.

[32] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC '78.*, pages 60–65, 1978.

[33] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. An empirical analysis of technical lag in npm package dependencies. In Rafael Capilla, Barbara Gallina, and Carlos Cetina, editors, *New Opportunities for Software Reuse*, pages 95–110, Cham, 2018. Springer International Publishing.

[34] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Smallworld with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 995–1010, USA, 2019. USENIX Association.