## ABSTRACT

IMTIAZ, NASIF. Toward Secure Use of Open Source Dependencies. (Under the direction of Laurie Williams).

Modern software extensively uses open source packages as upstream *dependencies*. While using the open source may be free, the downstream client projects must ensure their dependencies are secure. The goal of this dissertation is to aid software engineers in securely using open source dependencies.

Managing open source security can be broadly characterized into two fronts: (a) *reactive*: when a vulnerability is discovered in a dependency, the clients should react to any potential threat; and (b) *proactive*: before pulling in new dependency code, the clients should make an informed decision about the security of the code. We work on both reactive and proactive open source security in this dissertation.

On the *reactive* front, we performed a comparative study of nine existing software composition analysis (SCA) tools that notify a client of dependency vulnerabilities. We find that the tools vary in their vulnerability reporting. The count of reported vulnerable dependencies ranges from 17 to 332 for Maven and 32 to 239 for npm projects across the studied tools. Similarly, the count of unique known vulnerabilities reported by the tools ranges from 36 to 313 for Maven and 45 to 234 for npm projects. Our manual analysis of the tools' results suggests that the accuracy of the vulnerability database is a key differentiator for SCA tools.

Next, we empirically investigated 4,812 security releases from packages across seven ecosystems. Specifically, we studied (1) the time lag between fix and release; (2) how security fixes are documented in the release notes; (3) code change characteristics (size and semantic versioning) of the release; and (4) the time lag between the release and an advisory publication. We find a time lag between security fixes within open source packages and corresponding advisory publications, resulting in delayed notifications from SCA tools. The notification delay may occur even though we find the packages to typically document the security fixes in their release notes (61.5% of the time). Based on our findings, we recommend open source packages follow a standardized practice in announcing security fixes that can help automate the notification process to client projects.

On the *proactive* front, we work on building trust in dependency updates by identifying the authors and reviewers behind the changes within these updates. We implemented Depdive, an update audit tool for packages in Crates.io, npm, PyPI, and RubyGems registries.

Depdive first (i) identifies the files and code changes that cannot be traced back to the package's source repository, i.e., phantom artifacts, and then (ii) measures what portion of changes in the update has passed through a code review process, i.e., code review coverage. We empirically evaluated Depdive over the most downloaded packages from the four registries. We find that phantom artifacts are not uncommon in the updates (20.1% of the analyzed updates had at least one phantom file). Further, we find only 11.0% of the updates to be fully code-reviewed, showing that even the most used packages introduce non-reviewed code in the software supply chain.

Finally, we studied if a social network-based centrality rating for the authors and reviewers of package code can help client project developers review upstream changes. We built a social network of 6,949 developers across the collaboration activity from 1,644 Rust packages. We then computed a rating for each developer based on five centrality measures extracted from the network. Further, we surveyed Rust developers to evaluate if code coming from a developer with a higher centrality rating is likely to be accepted with less scrutiny by the client project developers and, therefore, is perceived to be more trusted. Our results show that 97.7% of the developers from the studied packages are interconnected via collaboration, with each developer separated from another via only four other developers in the network. Our survey responses ($N = 206$) show that the respondents are more likely to not differentiate between developers in deciding how to review upstream changes (60.2% of the time). However, when they do differentiate, statistical analysis showed a significant correlation between a developer's centrality rating and the level of scrutiny their code might face from the client projects, as indicated by the respondents. Overall, our findings indicate that social network-based centrality rating can be used to estimate the trustworthiness of a developer at a package ecosystem level and, therefore, may help client project developers decide what level of review new upstream changes require.

# Toward Secure Use of Open Source Dependencies

by
Nasif Imtiaz

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2023

APPROVED BY:

## BIOGRAPHY

Nasif Imtiaz was born in Khulna, Bangladesh. He grew up and spent all his life in Dhaka, Bangladesh, before moving to the United States. He attended SOS Hermann Gmeiner College in Mirpur, Dhaka. After that, he completed his undergraduate studies at Bangladesh University of Engineering & Technology. Finally, he moved to North Carolina, U.S. To pursue graduate studies, where he received M.S. and Ph.D. degrees from North Carolina State University. Apart from work and studies, he enjoys being a lifelong learner, exploring new cultures, and picking up new hobbies. He strives to be a better human being and practice kindness and empathy towards others.

## ACKNOWLEDGEMENTS

I would like to thank my Ph.D. advisor, Dr. Laurie Williams, and my committee members, Dr. William Enck, Dr. Brad Reaves, and Dr. Alexandros Kapravelos, for their guidance on my dissertation. I would also like to thank the members of RealSearch lab, Akond Rahman, Rezvan Mahdavi-Hezaveh, Sarah Elder, Rayhanur Rahman, Nusrat Zahan, and Setu Kumar Basak, for being great lab mates during this long journey.

I thank my father, S.M. Moniruzzaman, my mother, Nadira Akhter, my sister, Ishrat Oishi, and my brother-in-law, Fahim Rashid, for raising me and supporting me all throughout my life. I owe all of my accomplishments, big and small, to them.

I want to thank my wife, Preya Shabrina, who is the love of my life. I thank her for being with me during my worst times, always believing in me, and making my life beautiful with her gracious presence.

I would also like to remember all my friends from Bangladesh and the new ones I made here in the U.S. I would also like to thank my colleagues at Microsoft and Meta during my internships there. Finally, I want to thank everyone who has been a part of my life up to this point, especially the ones who supported me during this long Ph.D. journey.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

CHAPTER

# 1 INTRODUCTION

Modern software uses third-party open source packages as upstream *dependencies*. A 2021 report found that 98% of the audited codebases contained open source packages, with an average of 528 packages per codebase [1]. Today, open source packages, such as the ones distributed through npm [2], PyPI [3], and Crates.io [4], are considered essential parts of the software supply chain (Mcbride 2021; Kaczorowski 2020).

While the use of open source may be free, downstream client projects must ensure their dependencies are secure. In recent times, software applications have been attacked through their dependencies. These attacks happened in various ways, including through the exploitation of known vulnerabilities (Fruhlinger 2020; Berger 2021) and malicious updates (Flowers 2021). The security risk of known vulnerabilities in dependencies and malicious supply chain attacks has been studied in the literature (Zimmermann et al. 2019; Ohm et al. 2020; Liu et al. 2022). The goal of this dissertation is to aid software engineers in securely using open source dependencies.

[1] `https://www.cybersecasia.net/sponsored/2021-open-source-and-risk-analysis-report`

[2] `https://www.npmjs.com/`

[3] `https://pypi.org/`

[4] `https://crates.io/`

Managing open source security can be broadly characterized into two fronts: (a) *reactive*: when a vulnerability is discovered in a dependency, the clients should react to any potential threat; and (b) *proactive*: before pulling in new dependency code, the clients should make an informed decision about the security of the code. In this dissertation, we work on both reactive and proactive open source security. We investigate engineering best practices, such as documentation and code review, and introduce novel code audit tools and social network analysis techniques, to help client projects address known vulnerabilities in their dependencies and review the code changes in each new update.

***Thesis statement:*** *Well-documented security releases from upstream packages improve the vulnerability notification process to downstream projects, promoting timely remedial action. Furthermore, automated auditing, involving code review coverage and social network analysis, helps developers prioritize security review efforts for new upstream changes and maintain a secure software supply chain.*

We demonstrate our thesis statement through four original studies, each of which investigates a series of research questions:

1. A comparative study of software composition analysis (SCA) tools. The study demonstrates how much and why the current SCA tools vary in vulnerability reporting. We find that the tools' strengths depend on the accuracy, completeness, and up-to-dateness of the vulnerability databases they maintain. This work has been published at the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (Imtiaz et al. 2021).

2. An empirical study of security releases from open source packages. The study highlights notification delay for dependency vulnerabilities from SCA tools and demonstrates what best practices, if applied by the packages, will help the tools maintain an accurate vulnerability database and reduce notification delays. This work has been published at IEEE Transactions on Software Engineering (Imtiaz et al. 2022).

3. An empirical study on code review coverage among popular packages. The findings quantify the practice of code review among the most used open source packages from Crates.io, npm, PyPI, and RubyGems registries. This work is under review at the time of writing.

4. A study on how social network-based centrality ratings can be used to estimate the trustworthiness of the authors and reviewers of most used Rust packages in terms of

how carefully the client project developers would review upstream changes coming from them. This work is under review at the time of writing.

We make the following contributions from this dissertation:

- A quantitative comparison of the vulnerability reports from nine SCA tools on a real-world case study; a manual analysis of the differences among the tools' reports; and a characterization of metrics provided by the tools for risk assessment of the reported vulnerabilities. (Chapter 3)

- A quantitative analysis of the time lag between fix and release; how security fixes are documented in the release notes; code change characteristics (size and semantic versioning) of the release; and the time lag between the release and an advisory publication for security releases across seven open source package ecosystems. (Chapter 4)

- A categorization of the security fix and the breaking change-related information presented in the release notes of open source packages. (Chapter 4)

- An audit tool, Depdive, that measures phantom artifacts and code review coverage in updates of packages from Crates.io, npm, PyPI, and RubyGems registries. (Chapter 5)

- An empirical study on the phantom artifacts and code review coverage in the updates of the most downloaded packages from Crates.io, npm, PyPI, and RubyGems registries. (Chapter 5)

- An empirical analysis of the developer community structure of the Rust ecosystem (Chapter 6).

- A social network-based rating of Rust developers derived from an aggregation of five centrality measures; and a survey-based evaluation of our rating approach in terms of a correlation between a developer's centrality rating and the level of review their code may face from the client project developers (Chapter 6).

- Artifacts: Code and data from each study are also publicly available.

The rest of this dissertation is structured as follows: Chapter 2 explains the background concepts and discusses related work; Chapter 3 presents a comparative study of SCA tools; Chapter 4 presents an empirical study of security releases from open source packages; Chapter 5 presents an empirical study of code review coverage among popular packages; Chapter 6 presents a study on social network-centrality ratings for Rust developers; and Chapter 7 concludes this dissertation.

# CHAPTER 2

# BACKGROUND & RELATED WORK

In this chapter, we explain key concepts relevant to our work and review the existing related literature.

## 2.1 Key concepts

In this section, we briefly explain the key concepts related to our work.

### 2.1.1 Dependency & client

Dependency refers to when a piece of software depends on another one. For example, a software application may be built using many third-party software packages, either open-source or from proprietary vendors. In this work, we only refer to open source packages by the term dependency. The project that depends on a package is referred to as the client of the package.

Note that, clients may specify only a certain range of versions of a package to be used as a dependency, e.g., a project can specify version X.Y.Z of package A as its dependency. In

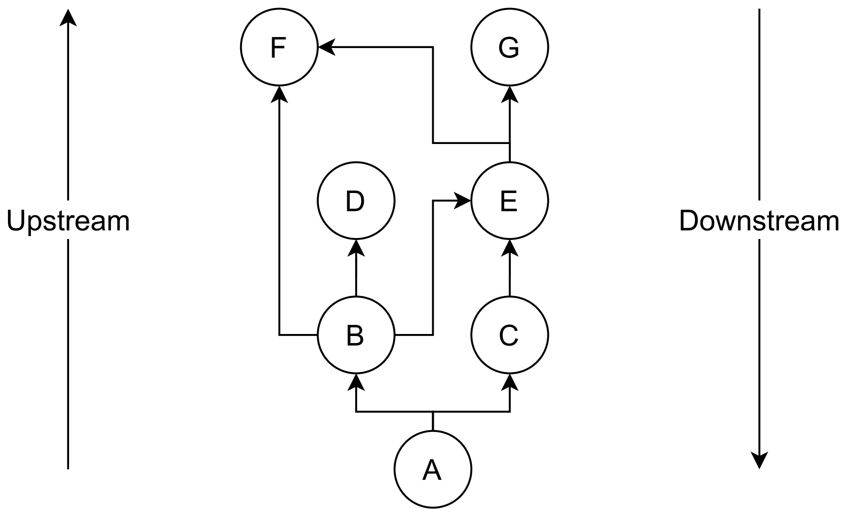


Figure 2.1: A sample package dependency network. The arrow sign indicates the dependency direction. A →B means A depends on B.

Chapter 3, we refer to dependency as an exact version of a package, as will be explained in Section 3.2.

Dependencies can be characterized in two ways based on how they are introduced:

#### 2.1.1.1 Direct dependency

When a client directly uses some functionalities of a package, the package is a direct dependency.

#### 2.1.1.2 Transitive dependency

A direct dependency of a client may bring its own dependencies with it, which become the transitive dependencies of the root client.

Through direct and transitive dependencies, a client can depend on many packages. Figure 2.1 shows a sample dependency network, where client A depends on package B, C, D, E, F, and G, through a complex dependency network. Here, package B and C are direct dependencies of A, while the rest of the packages are transitive dependencies.

### 2.1.2 Upstream & downstream

Besides dependency and client, the terms upstream and downstream can also be used to explain a dependency network. Downstream refers to client projects, whereas upstream refers to the dependencies of a project. Figure 2.1 visually explains the concept of upstream and downstream.

### 2.1.3 Package ecosystem

Programming languages, systems, or frameworks may have their own package ecosystems. For example, Python projects use packages from the PyPI ecosystem, whereas JavaScript projects use packages from the npm ecosystem. Package manager tools are commonly used to distribute packages from a particular ecosystem, e.g., Cargo for Rust, and npm for JavaScript. Packages can be stored and distributed from a centralized repository by the package managers, e.g., npm, PyPI, Crates.io, and RubyGems registry. We refer to these centralized package repositories simply as package registries. On the other hand, package manager tools can also simply fetch a package from its source repository and deliver it to the client without necessarily having a centralized repository, e.g., Composer (PHP) and Go package managers.

### 2.1.4 Semantic versioning (SemVer)

SemVer provides a formatting guideline for version numbering to indicate what type of changes have gone into the new version (Preston-Werner 2021). The format consists of three component numbers, X.Y.Z. All versions below 1.0.0 ($X < 1$) are considered ***unstable releases***, where developers are not expected to maintain backward compatibility for the public APIs. Starting from 1.0.0, a change in X indicates a ***major release*** – that is – a backward-incompatible release. A change in Y indicates a ***minor release*** where new functionalities have been added in a backward-compatible manner. Finally, any change in Z indicates a ***patch release*** – that is – a release containing only backward compatible bug fixes. Further, the version can be suffixed with qualifier strings such as ’pre’, ’beta’, and ’rc’ to indicate a ***pre-release***.

### 2.1.5 Breaking changes

Backward incompatible changes in a package update that may trigger rework for the clients are referred to as breaking changes (Bi et al. 2020; Preston-Werner 2021).

### 2.1.6 Release note

A formal document distributed with each version release of a package that explains the notable changes in the new version (Bi et al. 2020; Lacan 2021).

### 2.1.7 Vulnerability

NIST (of Standards and  NIST) defines vulnerability as “weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.” If a vulnerability gets exploited by a threat source, the potential for loss or damage is referred to as the ***risk*** of the vulnerability.

### 2.1.8 Known vulnerabilities (CVEs & Non-CVEs)

There exist public and proprietary databases that keep track of known vulnerabilities in certain releases of software applications. These vulnerability databases may have different systems to reference unique vulnerabilities. The most popular vulnerability reference-tracking method is the Common Vulnerabilities and Exposure (***CVE***) system [1], developed by MITRE [2]. CVEs are assigned by different parties across the world who belong to CVE numbering authorities (CNA) [3]. The National Vulnerability Database (NVD) [4], a publicly accessible repository, keeps track of all the CVEs. Some vulnerability databases, such as Snyk [5], may keep track of known vulnerabilities that do not necessarily have a CVE identifier. In our work, we refer to such vulnerabilities as ***Non-CVEs***.

### 2.1.9 Security advisory

Documentation of a vulnerability is typically referred to as security advisory (Maddox 2010). The advisory contains information about the vulnerability such as the vulnerability type,

[1] `https://cve.mitre.org/cve/`
[2] `https://cve.mitre.org/`
[3] `https://www.cve.org/ProgramOrganization/CNAs`
[4] `https://nvd.nist.gov/vuln`
[5] `https://snyk.io/vuln`

severity, way of remediation, mitigation, etc. In this work, we refer to a security advisory simply as an advisory.

### 2.1.10 Advisory database

Vulnerability databases are often referred to as ***advisory databases***. Each vulnerability in the database, along with the associated data, is then referred to as an advisory (Maddox 2010).

### 2.1.11 CWE

The Common Weakness Enumeration (CWE) is a category system for hardware and software weaknesses and vulnerabilities [6]. Typically, each vulnerability is associated with one or more CWE type(s).

### 2.1.12 Vulnerability severity

Severity is a prioritization metric that tries to estimate how quickly the affected party should address a vulnerability. The most common system for severity is the Common Vulnerability Scoring System (***CVSS***). While CVSS provides a score between 0 and 10, the scores can also be interpreted into four discrete levels: *Critical*, *High*, *Medium*, and *Low*.

### 2.1.13 Security fix

We refer to a security fix as the code change that fixes a vulnerability. In the case of open source packages, once the fix is committed to the repository, the commit message and code change become public as per the development model of the open source. Security fixes are also often referred to as ***security patches***.

### 2.1.14 Security release

We refer to a security release as the release of a version (an update) that includes one or more security fixes. Clients can then upgrade to the new release to adopt the corresponding security fixes.

[6] `https://cwe.mitre.org/`

### 2.1.15 Software composition analysis (SCA)

SCA is a part of application analysis that deals with managing open source use. SCA tools typically generate an inventory of all the open source components in a software product and analyze the license compliance and the presence of any known vulnerabilities in them. In our work, we are primarily concerned with vulnerability reporting by the SCA tools, that is, reporting known vulnerabilities in the dependencies of a software project.

### 2.1.16 Malicious code

Malicious code refers to content that is *intended* to cause undesired effects to a software system, such as a security breach. Malicious code can also include backdoors and time bombs. While vulnerabilities are supposed to be mistakes, malicious code is intentional.

### 2.1.17 Software supply chain

Software supply chain refers to anything that impacts the production, evaluation, and distribution of a software product (Mcbride 2021). The chain includes anything from third-party packages, code hosting platforms, and build and distribution infrastructure, to all the people and vendors involved in creating a software product. Vulnerability to any part of the chain can affect the end product, developers, and customers of the software.

### 2.1.18 Open source security: reactive & proactive

In modern software development, the use of open source packages is extremely common (Synopsys 2021). Therefore, these packages are an essential link in the supply chain, and, consequently, can be an attack vector. While supply chain security is a broad area, we only focus on the secure use of open source packages in this dissertation. The security of the open source packages is often referred to as open source security [7].

Open source security can be broadly characterized in two ways:

#### 2.1.18.1 Reactive open source security

Vulnerabilities can get discovered and be publicly known in open source packages. If a client is using a version of a package that is affected by such a known vulnerability, the

[7] `https://www.microfocus.com/en-us/what-is/open-source-security`

client software itself can be impacted by the vulnerability (Plate et al. 2015). If unpatched, such vulnerabilities in dependency packages pose a great security risk, as the knowledge of the vulnerability is public. Further, even closed-source software products may be legally required to disclose the open source packages they use based on the licensing policy of individual packages. Two recent major security incidents, Equifax data breach (Fruhlinger 2020), and Log4Shell attacks (Berger 2021), happened through known vulnerabilities in open source dependency packages.

To address any potential threat from known vulnerabilities in dependencies, the clients need to be informed of these vulnerabilities, assess the risk, and mitigate/remediate the vulnerability if necessary. We call this approach *reactive*, as the clients have to react when a new vulnerability gets publicly known in any of the dependencies they are using.

#### 2.1.18.2 Proactive open source security

The open source packages in the supply chain can be intentionally attacked as well. Anyone can upload a package to public registries, such as npm, and therefore, there may exist malicious packages in these registries. The most common way to trick clients into downloading a malicious package is *typosquatting*, that is, the malicious package having a name similar to a popular package (Ohm et al. 2020). For example, *crossenv*[8] was a malicious package trying to impersonate a popular package, *cross-env*, in the npm registry. In this regard, the clients need to be careful when including a dependency in their project that they are not mistakenly pulling in a malicious package in their software.

Another major way to attack through open source packages is to infect an existing package that is already used by many clients around the world. A package can be infected in many ways, including through hijacking a maintainer's account, taking over the package through social engineering, a maintainer going rogue, or subverting the build and publishing infrastructure to inject malicious code into the package before they are downloaded by the end clients (Ohm et al. 2020). This way, even an existing reliable package can push a malicious update. Arguably, a malicious update from a popular package is better situated to carry out successful attacks as many clients are already using the package, and they may pull in the new update without a security review, often automatically (Mirhosseini and Parnin 2017).

Further, a new update can introduce new vulnerabilities as well, either mistakenly or intentionally. The clients may proactively audit the code changes in every new dependency

[8] `https://www.npmjs.com/package/crossenv`

update to ensure they meet a certain security standard. Overall, the clients need to proactively audit and ensure that the dependency code is secure before merging them into their codebase, both during the dependency selection and during every subsequent update. We call this approach *proactive*

### 2.1.19 Git

Git is a distributed version control system that keeps track of code changes and aid in coordination among programmers during software development [9]. In a git repository, the most granular unit to track a revision of the source code is called a commit which is identified through a unique commit hash.

### 2.1.20 GitHub

GitHub is a code hosting platform that supports git-based software development [10]. GitHub is the largest code host and arguably the most popular platform among open source projects (Gousios et al. 2014). GitHub offers many features to support different phases of software development, including code review, continuous integration, advisory publication, dependency updates, and issue tracking.

### 2.1.21 Code review

Code review is a manual review process of code changes by any developer(s) other than the author. While code review can be done in any phase of software development, the term typically refers to the peer review process of code changes before they can get merged into the main codebase. While the history of code review is not tracked by git, reviews are generally performed using a tool, e.g., Gerrit [11]. GitHub offers a pull-based development model that integrates native code review tooling. A developer can open a *pull request (PR)* on GitHub to submit code changes and ask for reviews from other developers.

Source code can be reviewed by an external party(s) after the release of a software application. We refer to such review as ***code vetting***.

[9] `https://git-scm.com/`
[10] `https://github.com/`
[11] `https://www.gerritcodereview.com/`

### 2.1.22 Supply chain Levels for Software Artifacts (SLSA)

SLSA (salsa) [12] is a security framework that enlists a "check-list of standards and controls to prevent tampering, improve integrity, and secure packages and infrastructure in your projects, businesses or enterprises." SLSA provides lists of requirements in four levels of assurance. If a client wants to ensure that its supply chain meets a certain level of SLSA assurance, all parts of the chain, including the open source dependencies need to meet all the requirements that SLSA lists at that level.

### 2.1.23 Social network analysis

Social network analysis is the process of analyzing social structures through the use of network and graph theory. The network conceptualizes individuals as nodes and their relations as edges (Scott 2012). Below, we explain common network analysis methods and centrality measures.

#### 2.1.23.1 Network structures

Social network structures are typically explored through measuring the number of nodes, edges, communities, cliques, and the average shortest distance between nodes (Freeman 2004). Real-world social networks, such as virtual friendship and protein interaction networks, exhibit a small-world phenomenon. The phenomenon indicates that a network graph is sparse, and each node can reach another via a small number of intermediary nodes (six on average) (Sherchan et al. 2013). The phenomenon ties networks from different domains under a common abstraction and makes common social network analysis approaches like centrality measures applicable across those domains. (Sherchan et al. 2013).

#### 2.1.23.2 Centrality measures

Centrality measures estimate the influence of a node within a network (Das et al. 2018). Centrality measures are global measures that provide a rating for each node from the point of view of the whole network. Centrality measures have been used in various research to estimate the central figures in the network, including, but not limited to, a network of researchers, criminals, and students (Das et al. 2018). Researchers have proposed various centrality measures that may be applicable in different contexts (Das et al. 2018).

[12] `https://slsa.dev/`

#### 2.1.23.3 Trust measures

Trust has been studied extensively in the social network analysis literature (Sherchan et al. 2013). Trust in a social network is defined in the literature as “a measure of confidence that an entity will behave in an expected manner, despite the lack of ability to monitor or control the environment in which it operates” (Sherchan et al. 2013; Singh and Bawa 2007). Trust can be measured as *direct trust*—how much an individual trusts another in the network, or as *global trust*— how much all the users in the network trust an individual.

The trust of an actor in a network can be calculated from his/her social capital, which in turn can be calculated from objective metrics such as the number of interactions of the actor with others in the network (Sherchan et al. 2013). When the actors and their interactions are modeled as a graph, Buskens (Buskens 1998) found that trust can be estimated from the (i) out-degree of a node, (ii) out-degree of the neighbor nodes, and (iii) individual centralization. Many complex trust models have been proposed that are custom to the specific context. For example, Levien et al. (Levien 2009) proposed attack-resistant trust metrics in a distributed peer-to-peer network, where the trust measure depends on initial trusted seed machines in the network. However, all the trust models consider centrality measures (e.g., PageRank) of a node when estimating global trust/reputation for an actor by constructing a graph that incorporates the volume and frequency of interactions between the actors (Bosu et al. 2014; Asim et al. 2019; Ceolin and Potenza 2017; Meo et al. 2017; Zahi and Hasson 2020; Şimşek and Meyerhenke 2020).

## 2.2 Literature Review

In this section, we discuss related work to this dissertation.

### 2.2.1 Open source dependency network

There is a rich body of literature empirically surveying the dependency networks of various package ecosystems. Decan et al. (Decan et al. 2017) have studied the evolution and characteristics of the package dependency networks in seven open source ecosystems. They observed that the dependency networks tend to grow over time, both in size and in the number of total package updates. They also find that the majority of the packages have some dependencies, but only a small fraction accounts for most of the upstream dependencies. Zimmermann et al. (Zimmermann et al. 2019) have shown how a few packages and maintainers work as single points of failure for many JavaScript projects. Researchers have also studied the growth of open source (Kikas et al. 2017), the phenomenon of outdated dependencies (Zerouali et al. 2018; Lauinger et al. 2018; Cox et al. 2015), dependency resolution mechanisms (Lam et al. 2020; Decan and Mens 2019), the phenomenon of breaking changes in the dependencies (Mujahid et al. 2020; Raemaekers et al. 2014), and dependency management by the client developers (Pashchenko et al. 2020b; Kula et al. 2018).

### 2.2.2 Known vulnerabilities in dependencies

Researchers have extensively studied the presence and risk of security vulnerabilities in open source dependencies. Decan et al. (Decan et al. 2018b) studied the impact of security vulnerabilities in npm dependency network and found that the number of packages with a known vulnerability is growing over time. They also found that half of the dependent packages do not get fixed even when a security release is available. Similarly, Lauinger et al. (Lauinger et al. 2018) found around one-third of the packages in the npm network to have at least one vulnerable dependency. Hejderup et al.(Hejderup 2015) found that context use of the dependency and breaking changes are potential reasons for not resolving these vulnerabilities.

The potential impact of the vulnerabilities in dependency has also been studied. Zapata et al. (Zapata et al. 2018) studied the impact of a vulnerability in *ws* package on its clients. The study finds that 73.3% of the clients did not actually use the vulnerable code. The study also finds that the clients that do not use the vulnerable code take longer to remediate. To detect dependencies where the vulnerable code is actually used by the client, Ponta et al. (Plate et al. 2015; Ponta et al. 2018, 2020) proposed a code-centric and usage-based approach based on which the tool Steady is developed. Further, Koshibayev et al. (Koishybayev and Kapravelos 2020) developed *Mininode* that can filter out the unused dependencies from the attack surface. Hejderup et al. (Hejderup et al. 2018) have proposed a call-based dependency network that has a higher level of granularity than the default package-based network to better identify which dependency vulnerabilities may actually impact the client projects.

### 2.2.3 SCA tools

A few researchers in academia have studied SCA tools (Foo et al. 2019). Paschenko et al. (Pashchenko et al. 2020b) found that developers think SCA tools generate many irrelevant and low-priority alerts, and may even rely on social channels than SCA tools for vulnerability notification. Developers recommended SCA tools to "report only relevant alerts, work offline, and be easily integrated into company workflow". In another work, Paschenko et al. (Pashchenko et al. 2020a) discuss the over-inflation problem when reporting dependencies with unexploitable vulnerabilities by the SCA tools. Regarding comparing existing SCA tools, Ponta et al. (Ponta et al. 2020) compared their tool Steady with OWASP Dependency-Check. The study compares the two tools over a sample of alerts generated on Java applications. The comparison was performed from the perspective of the reachability of a vulnerability in the dependency. The study finds both tools to have their unique findings. The study also finds Steady to have no false positives but a few false negatives, while OWASP Dependency-Check has non-negligible false positives.

### 2.2.4 Vulnerability life cycle

The life cycle of a vulnerability typically refers to the time of i) introduction, ii) discovery (may never be known/ documented), iii) disclosure (vulnerability being disclosed to the corresponding package maintainers), iv) advisory publication, v) fix availability, and vi) exploit availability (Frei et al. 2006; Shahzad et al. 2012). Prior work (Frei et al. 2006; Shahzad et al. 2012; Nakajima et al. 2019; Shahzad et al. 2019; Ruohonen et al. 2020; Decan et al. 2018b; Alfadel et al. 2021a; Zerouali et al. 2021) has looked at the time duration between vulnerability disclosure, fix availability, and exploit availability. Li and Paxson (Li and Paxson 2017) have looked at the time duration between vulnerability introduction, discovery, and commit date of the code changes that fix the vulnerability.

However, the above works, while studying fix availability, did not distinguish between the code changes that fix the vulnerability, a workaround, or the release of a new version of the affected product. In the case of open source packages, client projects typically adopt the security fix by upgrading to the corresponding security release (Alfadel et al. 2021b; Chinthanet et al. 2021). But there can be a time lag between a vulnerability fix and the subsequent release of a new version. Li and Paxson (Li and Paxson 2017), while studying the commit dates of security fixes, note how the release of the fix may be delayed due to a project's release cycle, expanding the vulnerability's window of exposure. Even when the security aspect of the fixes is not publicly discussed (e.g., in the bug tracker or commit messages), prior work has identified such security fixes through data mining approaches (Ramsauer et al. 2020; Zhou et al. 2021b,a; Xu et al. 2017). Therefore, a security fix that has been committed to a public repository can be discoverable by observant attackers. But if the fix is not released, the clients would not have a simple way to adopt the fix. Similarly, advisory publication delay in the context of open source packages which could cause SCA tools to send delayed notifications to the clients, has not been studied yet, to the best of our knowledge.

### 2.2.5 Security fixes & releases

Prior work has investigated code changes that fix a security vulnerability (Li and Paxson 2017; Imtiaz et al. 2019). Li and Paxson (Li and Paxson 2017) found security fixes are small, seven lines of code (LOC) change at the median, and tend to be localized in a single file. Regarding releasing the fix, Chinthanet et al. (Chinthanet et al. 2021) looked at 231 security releases across 172 npm packages and found that a) 64.5% of the security releases are patch releases (according to SemVer format (Preston-Werner 2021)), and b) 91.8% of the releases have other code changes unrelated to the security fix.

### 2.2.6 Release notes

Prior work has investigated the documentation that comes with software releases. Bi et al. (Bi et al. 2020) have studied the release notes of 100 GitHub projects to understand what information they contain. The authors also conducted interviews and a survey to understand software practitioners' information needs from release notes. Their results indicate that practitioners feel the information in the release notes is scattered and vague. However, they do not look at the mention of security-related information in the release notes. Other prior works have looked at the automatic generation of release notes (Moreno et al. 2016), classifying issues mentioned in the release notes (Abebe et al. 2016), and how system administrators consume the release notes (Martius and Tiefenau 2020).

### 2.2.7 Fix propagation

Once packages release a security fix, clients will have to update to the fixed version. Kula et al. (Kula et al. 2018) have studied why developers did not update vulnerable dependencies and found that 69% of the developers were unaware of the security issue. Bodin et al. have studied how quickly security fixes propagate in the npm ecosystem (Chinthanet et al. 2021), while Paschenko et al. (Pashchenko et al. 2020c) have studied how security plays a role in dependency management. These studies (Chinthanet et al. 2021; Pashchenko et al. 2020c; Kula et al. 2018; Martius and Tiefenau 2020; Li et al. 2019) point out that security releases should: i) come out as soon as the fixes have been applied; ii) be free of other functional changes; iii) highlight the security fix to help quick adoption of the new release by the client projects.

### 2.2.8 Open source supply chain attack

Ladisa et al. (Ladisa et al. 2022) have developed a taxonomy of attacks on open source supply chains. Their taxonomy comprises 107 attack vectors and 33 safeguards. Other researchers have also studied past supply chain attacks and have pointed out open source dependencies as potential attack vectors (Ohm et al. 2020; Zahan et al. 2021; Zimmermann et al. 2019; Kaplan and Qian 2021).

Researchers have investigated how to combat malicious updates of open source dependencies. Duan et al. have proposed static and dynamic analysis approaches to detect malicious packages for the interpreted languages (Duan et al. 2020). Sejfia et al. have proposed machine learning models to detect malicious npm packages (Sejfia and Schäfer 2022). Further, Ferreira et al. have proposed a permission-based protection mechanism to contain malicious npm updates (Ferreira et al. 2021). Researchers have also studied development patterns that can explain or predict malicious contributions in open source. Giovanini et al. (Giovanini et al. 2021) have proposed predicting *team susceptibility to social engineering-based supply chain attacks* using developer interaction data. Gonzalez et al. (Gonzalez et al. 2021) and Goyal et al. (Goyal et al. 2018) have proposed anomaly detection techniques to predict malicious or unusual commits in the source code repository.

### 2.2.9 Emerging industry standards for dependency packages

While they provide the benefit of code reuse, open source packages also come with security risks. The risk of known and unknown vulnerabilities in the dependencies has been studied in the literature (Decan et al. 2018b; Lauinger et al. 2018; Imtiaz et al. 2022, 2021). Another risk of using open source packages is the possibility of supply chain attacks (Ohm et al. 2020). Attackers have compromised popular packages and used the package to conduct malicious attacks (Ohm et al. 2020). Therefore, practitioners now recommend reviewing the package code before adding or updating a dependency (Imtiaz et al. 2022; Yang et al. 2021). Consequently, industry standards have emerged that provide a checklist of controls for the safe use of open source packages. A Notable example is Supply Chain Levels for Software Artifacts (SLSA) [13].

[13] `https://slsa.dev/`

Prior research has shown that developers may decide on accepting new package code based on how trustworthy the maintainers appear to be (Zahan et al. 2022; Wermke et al. 2022). Similarly, SLSA has put emphasis on human evaluation of code by requiring open source code to be two-person reviewed [14]. Following SLSA requirements, Imtiaz and Williams (Imtiaz and Williams 2022) have proposed measuring code review coverage (*CRC*) when accepting dependency updates. However, they note that without ensuring the trustworthiness of the authors and reviewers, the *CRC* metric by itself may not be adequate. Existing work thus shows the need for an identity model for open source developers that may reflect the developer's reputation and past activity history within a particular ecosystem. We address this need in this paper through an approach of a social network-based developer ranking system.

### 2.2.10 Code review

There is a rich body of literature establishing the benefits of code review in software development (Sadowski et al. 2018; McIntosh et al. 2014; Kononenko et al. 2015). McIntosh et al. (McIntosh et al. 2016) found that "code review coverage, participation, and expertise share a significant link with software quality". Bosu et al.(Bosu et al. 2014) found that code review can detect (and prevent) common types of vulnerabilities. Bacchelli et al. (Bacchelli and Bird 2013) have noted that code review helps disseminate the knowledge of the codebase.

Regarding security implications of code review, Brewer et al. (Brewer et al. 2021) in a security blog from Google have noted how code review can limit what an adversary can do on its own. Bosu et al. (Bosu et al. 2014) studied why security defects go undetected (and detected) during the code review process. They find multiple factors, including the CWE type and number of prior reviews by the reviewer, to be positively correlated with a vulnerability being detected during code review. Meneely et al. (Meneely et al. 2014) have found that the reviewer's security experience may indicate if code review would be successful in detecting vulnerabilities.

### 2.2.11 Developer social network (DSN)

Social network analysis is the process of analyzing social structures through the use of network and graph theory. The network conceptualizes individuals as nodes and their relations

[14]`https://slsa.dev/`

as edges (Scott 2012). In a developer social network (DSN), the nodes are developers, and the edges represent their communication or collaboration relations (Herbold et al. 2021). A rich body of literature exists that studies DSN (Herbold et al. 2021; Schreiber and Zylka 2020; Bosu and Carver 2014; Meneely and Williams 2011).

Herbold et al. (Herbold et al. 2021) have conducted a systematic mapping study of DSN research. They find nearly half of the research investigates the structure of the developer community. Further, they find that most DSN studies work with a small sample of projects. Meneely et al. (Meneely and Williams 2011) have investigated if the metrics obtained from sociotechnical developer networks are reliable through a developer survey. They found statistical evidence for DSN metrics to represent how developers collaborate in practice. Further, Nia et al. (Nia et al. 2010) found the network measures to be robust even with incomplete data or missing links between some developers.

Prior research on DSN has primarily focused on single large-scale projects (Herbold et al. 2021). However, open source packages are typically split based on specific functionalities, and the developers may collaborate on many related packages. Research has shown the complex dependency network between packages within an ecosystem (Kikas et al. 2017; Schueller and Wachs 2022; Schueller et al. 2022). Therefore, we conjecture that developers in the Rust supply chain may be interconnected as well. Consequently, we may leverage the developer network at a package ecosystem level and measure social metrics like trust and reputation, as was done in prior DSN work focused on single large-scale projects (Meneely and Williams 2011; Bosu and Carver 2014).

# CHAPTER 3 COMPARATIVE STUDY OF SCA TOOLS

## 3.1 Motivation & Research Questions

Software composition analysis (SCA) tools are used to report known vulnerabilities in open source dependencies. However, these tools may differ in how they detect the dependencies and the vulnerability database they maintain. Furthermore, not all alerts generated by SCA tools are relevant or high priority to the developers (Pashchenko et al. 2020b). If and how existing SCA tools aid developers in assessing the risk of the vulnerabilities from the context of the client application needs to be studied to help future research.

The goal of this study is to aid security practitioners and researchers in understanding vulnerability reporting by software composition analysis tools through a comparative study of these tools over a real-world case study. Our research questions are:

**RQ1:** What are the differences between vulnerability reports produced by different SCA tools?

**RQ2:** What metrics are presented by SCA tools to aid in the risk assessment of dependency vulnerabilities?

To answer, we present an in-depth case study by running nine SCA tools on a large web application, OpenMRS, that utilizes two popular package ecosystems. The application comprises 43 Maven (Java) and 5 npm (JavaScript) projects. The studied SCA tools vary in their scanning technique and vulnerability database, and represent the state-of-the-art. The contributions of this study include an evaluation of SCA tools through (a) a quantitative comparison of their vulnerability reports on a real-world case study; (b) a manual analysis of the differences among the tools' reports; and (c) a characterization of metrics provided by the tools for assessment of the dependency vulnerabilities.

## 3.2 Background

In this section, we explain the key concepts related to our work.

### 3.2.1 Terminology

Typically, a client declares a specific or a range of valid versions of a package as its dependency in a manifest file that we refer to as ***dependency file***. In the remainder of this chapter, we refer to ***dependency*** as a specific version of a package. For example, version 1.0.0 and version 2.0.0 of the same package *A* will be considered distinct dependencies. However, they will be considered as the same ***package***. When a dependency is subject to publicly known vulnerabilities, it is referred to as a ***vulnerable dependency***.

The dependencies declared through dependency files are resolved through some package manager. *pom.xml* and *package.json* are dependency files for Maven and npm package manager, respectively. For most package managers, including Maven and npm, the whole dependency structure is hierarchical and forms a tree format, as shown in Figure 2.1. The ***depth*** of a dependency refers to its level in the dependency tree, with direct dependencies having a depth of one.

### 3.2.2 Maven

Maven is a package manager for Java projects.

**Dependency Scopes** Maven dependencies can have six different scopes: compile, provided, runtime, test, system, and import. The scopes determine the phase when a dependency will be used and if the dependencies can propagate transitively.

**Dependency Mediation** When there are multiple versions of a package in the dependency tree, Maven picks one with the nearest definition. Therefore, usually, a single project has a single version of a package as a dependency that is read from a local repository. In the dependency file (*pom.xml*), developers generally specify a single version for its dependencies. Version numbers can have up to five parts indicating major, minor, or incremental changes.

### 3.2.3 Node package manager (npm)

npm is a package manager for JavaScript projects.

**Dependency Scopes:** npm has two primary dependency scopes: *Prod* (production) and *dev* (development) to indicate the phase where a dependency is required.

**Dependency Mediation** npm copies all the dependencies in a project subdirectory called 'node_modules', with a similar structure to the dependency tree. If two dependencies *A* and *B* both depend on the same package *C*, two different copies of package *C* will be copied inside package *A* and *B*. Therefore, the same dependency can have multiple paths to be introduced to the root application. Also, the same package can have multiple versions as dependencies. Therefore, npm has a concept called ***dependency path***, which is not present in Maven. Each unique path a dependency is introduced to the root application is referred to as the *dependency path*. In npm, developers can list a range of versions for a package that is valid as a dependency. npm also has the concept of ***lock*** files – a snapshot of the entire dependency tree and their resolved version at a given time; and can be used to instruct npm to install the specified versions in the lock file. npm packages follow the SemVer format (Preston-Werner 2021) for version numbering.

## 3.3 Evaluation Case Study: OpenMRS

OpenMRS is a web application for electronic medical record platform [1]. A particular configuration of OpenMRS that can be installed and upgraded as a unit is referred to as a *distribution*. The general-purpose distribution of OpenMRS is the "Reference Application Distribution". We choose Version 2.10.0 of this distribution, released on April 6, 2020 (the latest release at the time of this study), as our evaluation subject. In the remainder of the chapter, we refer to the whole distribution simply as "OpenMRS".

[1]`https://openmrs.org/`

OpenMRS comprises 44 projects that are hosted in their own separate repositories on GitHub. Out of the 44 projects, 39 are Maven projects and 1 is an npm project. The other 4 projects are composed of a Maven and an npm project each. Based on the OpenMRS structure, we scope our study to Maven and npm dependencies. We use OpenMRS SDK to automate the build, test, and run of the individual projects and assemble the full application.

### 3.3.1 Why OpenMRS?

Choosing test cases to evaluate software security tools can be a complex task. For comparison of security tools, Delaitre et al. (Delaitre et al. 2018) note that the test case should have a *sufficient, and diverse number* of security weaknesses. OpenMRS depends on many third-party dependencies, as will be seen in Section 3.3.2, and being a web application, is composed of several heterogeneous components, such as a database, content generation engines, client-side code, etc., therefore increasing the probability of having a large, diverse set of vulnerable dependencies.

Another approach to comparison instead of a single case study can be running the tools on a group of diverse projects. However, three of the selected tools in this study (Steady, Commercial A, and B) are (a) resource and time-consuming to set up and run; (b) involve certain requirements, e.g., acceptance tests for interactive binary instrumentation, unit tests for executability tracing; and (c) involve permission issues in the case of the commercial tools. On the contrary, focusing on a single case study enables us to manually investigate the differences in the tools' results.

OpenMRS has also been used in security research in the past (Crain 2017; Tøndel et al. 2019; de Abajo and Ballestero 2012; Lamp et al. 2018; Rizvi et al. 2015; Amir-Mohammadian et al. 2016). (Lamp et al. 2018) evaluated OpenMRS for medical system security requirements; (Rizvi et al. 2015) evaluated OpenMRS for access control checking; while (Amir-Mohammadian et al. 2016) studied OpenMRS for correct audit logging.

### 3.3.2 OpenMRS: Dependency overview

In this section, we provide an overview of Maven and npm dependencies of OpenMRS. We parse the dependency tree of each project through native `mvn dependency:tree` and `npm list` command. We also parse each dependency's scope and depth in the dependency tree.

Table 3.1 provides a dependency overview of OpenMRS. Note that, for Maven projects,

Table 3.1: OpenMRS dependency overview

| | Maven | npm |
|---|---|---|
| No. of projects | 43 | 5 |
| Total unique dependencies (package and version) | 547 | 2,213 |
| Total unique packages | 311 | 1,498 |
| Median dependency per project | 127.0 | 840.5 |
| Median dependency path per project | NA | 1,675.0 |
| Median depth of dependencies | 2 | 4 |
| Max. depth of dependencies | 7 | 12 |
| Median Provided dependencies | 99.0 | NA |
| Median Compile dependencies | 3.0 | NA |
| Median Runtime dependencies | 5.0 | NA |
| Median Test dependencies | 24.5 | NA |
| Median Production dependencies | NA | 202.5 |
| Median Production dependency path | NA | 366.0 |
| Median Developer dependencies | NA | 807.5 |
| Median Developer dependency path | NA | 1,613.5 |

there can be *internal* dependencies – that is – a project within the OpenMRS distribution can be listed as a dependency for another project. We do not count the internal dependencies in Table 3.1. Also, npm projects can contain *lock* files such as *shrinkwrap.json* and *package-lock.json* which are not considered.

## 3.4 SCA Tools

In this section, we explain the criteria we use to select the SCA tools; the description of the tools; how we performed the scan on OpenMRS; and how we analyzed the reports produced by the tools.

### 3.4.1 Selection criteria

To identify the existing SCA tools from both industrial offerings and the latest research, we performed an academic literature search and a web search through the following keywords: (*vulnerable* OR *open source* OR *software*) AND (*dependency* OR *package* OR *library* OR *component* OR *composition*) AND (*detection* OR *scan* OR *tool* OR *analysis*). From the relevant search results, we filtered the tools with the following *inclusion criteria*: a) scans either Maven or npm projects; b) we have access to an executable tool; and c) offers unique features when compared with already selected tools. From our selection process, we selected nine tools. Three of the tools are not freely available, and the license agreements prevent us from providing names. We refer to them as Commercial A, B, and C. Out of the selected 9 tools, 4 tools can scan both Maven and npm projects, 1 tool scans only npm projects, while 4 tools scan only Maven projects.

We observed that SCA tools primarily differ in three dimensions:

1. **Vulnerability database:** To report the list of known vulnerabilities, the tools need a database. Tools can pull vulnerability data from *third-party* source(s) such as NVD CVEs [2]. Additionally, SCA tools can maintain their own vulnerability database, where they collect and verify vulnerability data through different techniques (Catabi-Kalman 2020; Zhou and Sharma 2017).

2. **Dependency scanning source:** SCA tools can detect open source dependencies from dependency manifest files, source code, and binaries. Typically, dependency files are the common source to resolve dependencies of a project, as is done by the package managers as well.

3. **Additional analysis to infer dependency use:** Tools can perform additional static and/or dynamic analysis to infer how the dependencies are being used by an application.

### 3.4.2 Tool description

For the selected tools, we describe (a) if they scan Maven or npm dependencies, their (b) data source, (c) scanning technique, and (d) how we performed the scan for this study.

1. **OWASP Dependency-Check (DC):** This tool scans both Maven and npm projects and works by scanning the dependency files, JARs, and JavaScript files [3]. It pulls vulnerability data from multiple third-party sources, including NVD, OSS Index, and npm advisories. We used the Maven plugin to scan Maven projects and the command line tool (Version 5.3.2) to scan npm projects. We had the *experimental analyzer* option enabled to perform JavaScript scanning.

2. **Snyk:** Snyk also scans both Maven and npm projects. The tool works by scanning de-

[2] `https://nvd.nist.gov/vuln`
[3] `https://jeremylong.github.io/DependencyCheck/general/internals.html`

pendency files [4] and maintains its own vulnerability database [5]. We ran the command line tool (Version 1.382.0) that is freely available through the command `snyk test -all-projects -d`

3. **GitHub Dependabot:** Dependabot scans both Maven and npm projects hosted on GitHub. GitHub maintains its own vulnerability database[6], where it pulls data from NVD, and npm advisories. Additionally, maintainers on GitHub can publish vulnerabilities in their projects as well. We hosted the 44 studied projects on the first author's GitHub account and retrieved the Dependabot alerts through GitHub API.

4. **Maven Security Versions (MSV):** This tool only scans Maven projects [7] through dependency files. We ran this tool through its Maven plugin.

5. **npm audit:** This is a native tool of npm package manager for scanning npm projects. The tool works by scanning dependency files and maintaining its own vulnerability database. We used the `npm audit -json` command.

6. **Eclipse Steady:** This tool only scans Java (Maven) projects. The tool performs additional analysis to assess the execution of vulnerable code in the dependencies of an application [8]. The approach implemented is described in (Ponta et al. 2018) and (Plate et al. 2015). The tool requires a manual setup, along with the vulnerability database provided by the tool. We used Version 3.1.10 of this tool. We set up Steady in a virtual machine, allocating 16 GB RAM, and 4 processor cores. Steady hosts their vulnerability data set on GitHub. The data set contains patch commit information for each vulnerability. We imported the data source updated on Jan 24, 2020. We then performed the patch analysis feature provided by the tool to identify the involved code constructs for each vulnerability. For reachability analysis of the identified vulnerabilities, Steady performs three analyses: 1) static call graph construction; 2) executing JUnit tests for analyzing executability traces; and 3) JVM instrumentation through integration testing. We were unable to complete the third analysis as the tool presumably ran out of memory after running for ten days.

7. **Commercial A:** This tool has scientific papers discussing their approach (not citing to maintain blindness). We contacted their research team and provided them with the

[4]`https://support.snyk.io/hc/en-us/articles/360000925438-What-does-Snyk-access-and-store-when-scanning-a-project-`

[5]`https://snyk.io/vuln`

[6]`https://github.com/advisories`

[7]`https://blog.victi.ms/`

[8]`https://eclipse.github.io/steady/about/`

repository links for the studied projects. They returned to us with scan reports only for Maven dependencies for 37 projects and reported that they failed to complete the automated scans for the rest of the projects, which may have required manual intervention. This tool offers static analysis by default and dynamic analysis as an option to identify vulnerable call chains. We received results only with static analysis performed on the code. The tool maintains its own vulnerability database.

8. **Commercial B:** We used the free cloud edition of the tool, which only scanned the Java dependencies (the customer support informed us that the tool does not scan front-end libraries). The tool checks for the reachability of vulnerabilities in dependency through interactive application security testing – that is – monitoring dependencies in use when an application is run and interacted with either through automated testing or human testers. The tool uses third-party vulnerability databases, including NVD, which they curate themselves to enhance accuracy. To run this tool on OpenMRS, we make use of 123 test cases provided by OpenMRS for integration testing that interact with the application through a Selenium web driver. We connected OpenMRS to this tool and used the integration test suite to interact with the application.

9. **Commercial C:** This tool has a free/limited offering as a third-party application (bot) on GitHub, which scans both Maven and npm projects. Com. C also maintains its own vulnerability database. We connected the GitHub bot with our hosted repositories and retrieved the issues created by the bot through GitHub API.

   We collected the scan reports separately for 44 projects for 8 tools. For Commercial B, which analyzes the application during runtime, we get a single report for the whole OpenMRS distribution. As vulnerability data gets updated over time, we ran all the tools during September 2020 to ensure a fair comparison, except for Steady, whose vulnerability data is from January 2020.

### 3.4.3 Analyzing tool results

Below, we discuss the metrics and information that we processed from the tool reports to answer our research questions.

**Quantity of Alerts:** When a project is scanned by a tool, the tool reports a raw count of alerts identified on the project. However, the alerts do not represent either unique dependencies or unique vulnerabilities. We observed that the same alerts can be repeated

Table 3.2: Vulnerable Dependencies for Maven (Java) projects

| Tool | Alert | Unique Dependency | Unique Package | Unique Vulnerability | CVE | Non-CVE | Scan Time (Minutes) |
|---|---|---|---|---|---|---|---|
| | Total (Median per project) | | | | | | |
| OWASP DC | 12,466 (254.0) | 332 (38.0) | 149 (36.0) | 313 (117.0) | 289 | 24 | 14.4 |
| Snyk | 4,902 (66.0) | 96 (6.0) | 46 (6.0) | 189 (23.0) | 178 | 11 | 15.1 |
| Dependabot | 136 (0.0) | 20 (0.0) | 11 (0.0) | 61 (0.0) | 61 | 0 | NA |
| MSV | 3,197 (58.0) | 36 (12.0) | 14 (12.0) | 36 (22.0) | 36 | 0 | 3.4 |
| Steady | 2,489 (51.0) | 91 (20.0) | 39 (19.0) | 97 (41.0) | 89 | 8 | 385.0 |
| Commercial A | 2,998 (70.0) | 107 (24.0) | 53 (24.0) | 208 (70.0) | 187 | 21 | NA |
| Commercial B | 205 | 35 | 35 | 127 | 127 | 0 | NA |
| Commercial C | 434 (0.0) | 76 (0.0) | 44 (0.0) | 146 (0.0) | 127 | 19 | NA |

Table 3.3: Vulnerable Dependencies for npm (JavaScript) projects

| Tool | Alert | Unique Dependency Path | Unique Dependency | Unique Package | Unique Vulnerability | CVE | Non-CVE | Scan Time (Minutes) |
|---|---|---|---|---|---|---|---|---|
| | Total (Median per project) | | | | | | | |
| OWASP DC | 1,379 (208.0) | 498 (72.0) | 239 (71.0) | 160 (57.0) | 234 (71.0) | 78 | 156 | 4.4 |
| Snyk | 2,210 (135.0) | 1,004 (44.0) | 90 (20.0) | 54 (17.0) | 121 (26.0) | 79 | 42 | 1.0 |
| Dependabot | 97 (8.0) | NA | 32 (1.0) | 30 (1.0) | 45 (4.0) | 29 | 16 | NA |
| npm audit | 1,266 (37.0) | 852 (28.0) | 58 (12.0) | 45 (12.0) | 62 (16.0) | 31 | 31 | 0.1 |
| Commercial C | 205 (32.0) | 205 (32.0) | 89 (14.0) | 55 (9.0) | 96 (18.0) | 58 | 38 | NA |

in tools' reports for various reasons. The alert count, however, may indicate the amount of audit effort required from the developers.

**Tracking unique dependency, dependency path, package, and vulnerability:** The definitions of these four metrics, as used in this study, are provided in Section 3.2. When processing the analysis reports from all the tools, we store the data in a relational database schema. In the schema, we keep an identifier for each unique package, dependency (package:version), dependency path, and CVE identifier. For the non-CVEs, all tools except OWASP DC and Commercial A provide a tool-specific identifier. While OWASP DC and Commercial A provide no reliable identifier to track unique non-CVEs, upon manual inspection, we noticed that the vulnerability description along with the affected package(s) are a reliable way to track non-CVEs. However, we have no reliable way to map non-CVEs across different tool reports.

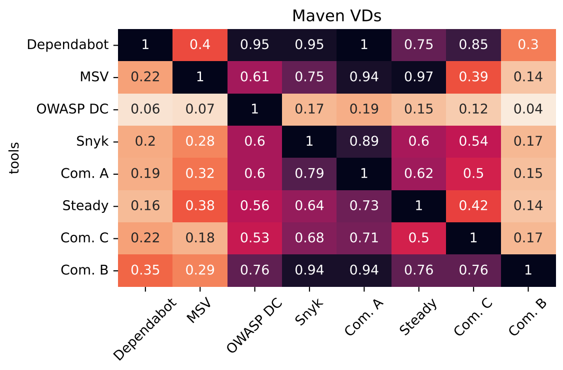
**Scan time** indicates the total number of minutes a tool took to scan all the projects. We have no scan time for GitHub and Com. A, as they are GitHub cloud services. We collected the issues and alerts from GitHub at the end of September, at least two weeks after hosting the repositories. Commercial B monitors dependency during runtime through interaction, therefore, also does not have a definite scan time.

**Other information:** Tools had additional information in their reports, generally to aid developers in assessing the risk of the alerts and to help in fixing them. We also collected these additional data, which will be explained in Section 3.5 when discussing the findings.

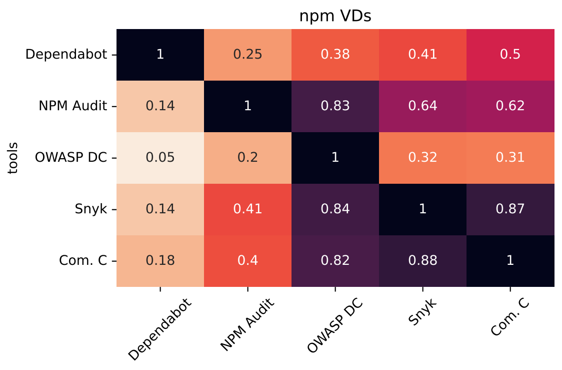
**Manual analysis of the tools' report:** To understand why there are differences in the tools' results, we manually inspected the tools' results. We specifically focused on the project *coreapps* as this is the project with the largest dependency count and includes both Maven and npm dependencies. The first author went through results from all the tools for *coreapps*, and categorized the differences. The second author then independently went through the results from the studied tools and verified the categorization done by the first author.

## 3.5 Findings

In this section, we present descriptive statistics on how SCA tools differed on vulnerability detection, a manual analysis on why the tools differed (RQ1); and a characterization of the metrics provided by the studied tools for aiding in risk assessment of vulnerability in dependencies, (RQ2).



(a) Overlap ratios for Maven vulnerable dependencies



(b) Overlap ratios for npm vulnerable dependencies

Figure 3.1: Overlap analysis of unique vulnerable dependencies for each tool pair: Cell(i, j) indicates the percentage of the i'th tool's reported vulnerable dependencies that are also reported by the j'th tool.

### 3.5.1 RQ1: What are the differences between vulnerability reports produced by different SCA tools?

Table 3.2 and 3.3 show the tools' result summary for Maven and npm dependencies, respectively. The table provides the total count for alerts and unique dependencies, dependency paths (for JavaScript), packages, and vulnerabilities for OpenMRS as a single application. The tables also report the total count of CVEs, non-CVEs, and scan time. For the eight tools that scanned projects individually (besides Commercial B), the tables provide in parentheses the median count per project for alerts and for unique dependencies, dependency paths, packages, and vulnerabilities.

The alert counts are higher than the count of unique vulnerabilities or dependency paths, as discussed in Section 3.4.3. While the total alert count repeats the same vulnerabilities found across projects, some tools repeat the same alert within a project as well due to the modular project structure. We also see the unique dependency count is higher than the unique package count. Different versions of the same package may be declared as a dependency in different projects, while npm can have multiple versions of the same package as dependencies even within a single project. We now discuss how the SCA tools have differed in their reporting:

**The tools differed both in identifying unique vulnerable dependencies and the unique vulnerabilities:** OWASP DC detects the highest number of unique dependencies and unique

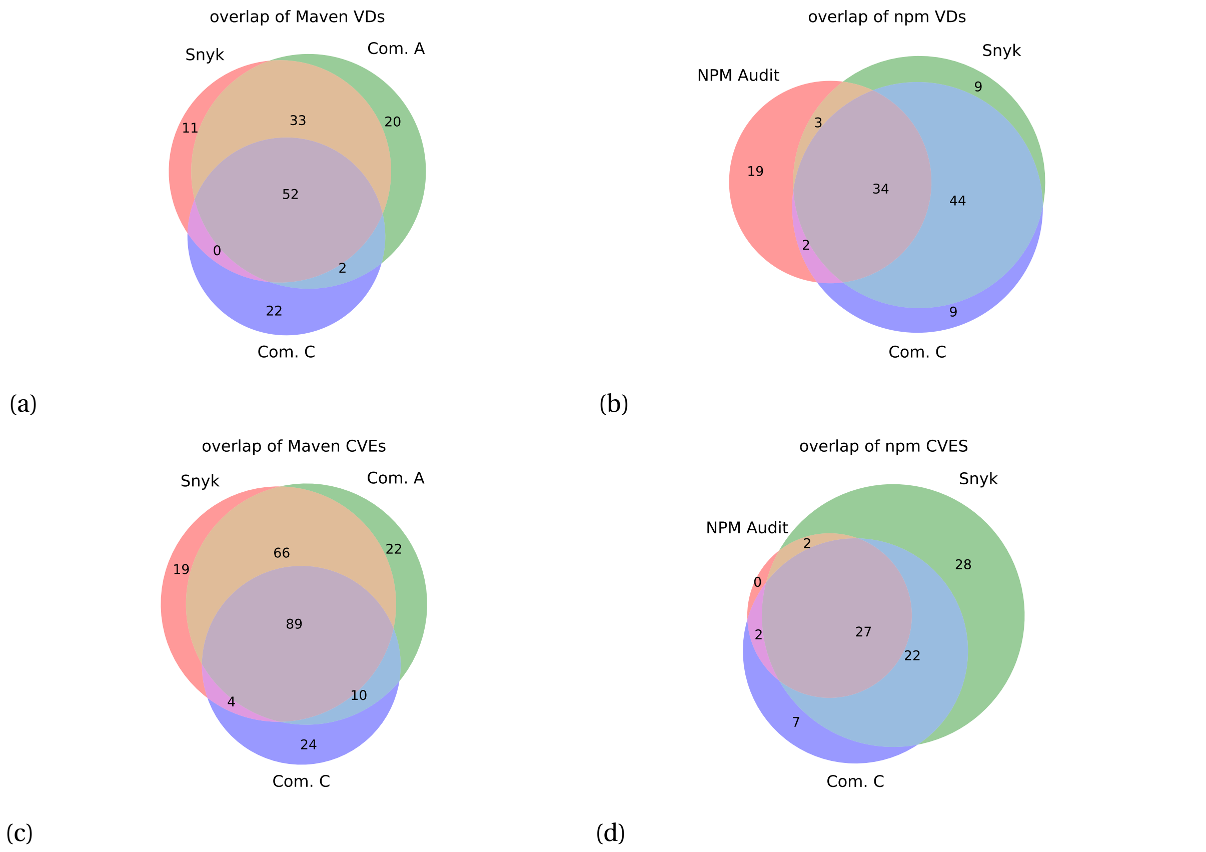


Figure 3.2: Venn Diagram for overlap of vulnerable dependencies and CVEs among three representative tools: OWASP DC, Snyk, and Com. C (Maven)/NPM Audit (npm). The sub-figures represent an overlap of (a) Maven vulnerable dependencies, (b) npm vulnerable dependencies, (c) maven CVEs, and (d) npm CVEs.

vulnerabilities for both Maven and npm projects. However, our analysis in Section 3.5.2 indicates more may not necessarily be better. Conversely, Commercial B, which monitors the dependency under use during runtime, detected the lowest number of vulnerable dependencies for Maven projects. MSV and Dependabot detected the lowest number of unique vulnerabilities, respectively, for Maven and npm projects.

**5 out of the 8 tools for Maven reported non-CVEs while all 5 tools for npm reported non-CVEs.** We observe that npm packages have a higher proportion of non-CVEs to CVEs than Maven packages. We find OWASP DC to report higher non-CVEs than any of the other 4 tools for npm projects. However, as OWASP DC does not provide an identifier for non-CVEs, we tracked unique non-CVEs through vulnerability descriptions and affected packages, which may have resulted in the duplication of the same vulnerabilities.

Table 3.4: Scope breakdown, rate of direct dependencies among reported vulnerable dependencies (VDs), and max. depth for the reported transitive VDs for Maven projects

| Tool | Scope breakdown | | | | Direct VDs (across all projects) | Max. Depth of VDs |
|---|---|---|---|---|---|---|
| | Compile | Provided | Runtime | Test | | |
| OWASP DC | 58 | 66 | 4 | 54 | 5.8% | 6 |
| Snyk | 56 | 62 | 2 | 25 | 14.8% | 7 |
| Dependabot | 15 | 5 | 1 | 2 | 97.0% | 2 |
| MSV | 19 | 30 | 1 | 3 | 1.7% | 5 |
| Steady | 60 | 60 | 4 | 11 | 5.3% | 5 |
| Commercial A | 72 | 79 | 1 | 0 | 8.7% | 5 |
| Commercial C | 54 | 0 | 2 | 0 | 60.0% | 5 |

Table 3.5: Scope breakdown, rate of direct dependencies among reported vulnerable dependencies (VDs), and max. depth for the reported transitive VDs for npm projects

| Tool | Scope breakdown | | Direct VDs (across all projects) | Max. Depth of VDs |
|---|---|---|---|---|
| | Prod | Dev | | |
| OWASP DC | 65 | 207 | 4.4% | 10 |
| Snyk | 13 | 83 | 3.5% | 10 |
| Dependabot | 6 | 8 | 65.9% | 6 |
| npm audit | 15 | 51 | 0.9% | 10 |
| Commercial C | 12 | 76 | 8.8% | 10 |

**Only 2 out of the 5 tools that scanned npm projects report vulnerable dependency paths:** In npm, the same package *A* can be introduced transitively through multiple direct dependencies and, therefore, can lie in multiple dependency paths. The developers may need to fix each path separately if there is a vulnerability in package *A*. We find that only npm audit and Snyk report all possible dependency paths to each unique vulnerability.

**Tools have non-overlap in reported vulnerabilities and dependencies:** We measured how much of the unique vulnerable dependencies reported by the tools overlap with each other. The heat maps in Figure 3.1 show the overlap ratio across tool pairs for both Maven and npm projects. For a tool pair (*A*,*B*), the heat map shows how many dependencies reported by *A* were also reported by *B* and vice versa. For example, for maven projects, 54% of Snyk's reported vulnerable dependencies were also reported by Com. C. Conversely, 68% of Com. C's reported maven dependencies were also reported by Snyk. Figure 3.2a, 3.2b demonstrates non-overlap in dependencies through a Venn diagram for three representative tools. We can not show such a heat map for all unique vulnerabilities, as we were unable to cross-reference non-CVEs across tools. However, we also found non-overlap over-reported CVEs as well across tools. Figure 3.2c, 3.2d demonstrates non-overlap in

CVEs for OWASP DC, Snyk, and Com. C.

**Tools detected vulnerable dependencies across all scopes and depths:** Table 3.4 and 3.5 show a breakdown of scan results per dependency scope and what portion of the reported vulnerable dependencies are introduced directly by OpenMRS. We find that reported vulnerabilities are mostly introduced through transitive dependencies, except for Dependabot and Com. C. The latter two tools assist GitHub projects in automatically fixing the vulnerable dependencies (by upgrading to a safer version), which may explain the high rate of direct dependencies in their reporting.

We find SCA tools to vary widely in the reporting of known vulnerabilities, for both Maven and npm dependencies.

### 3.5.2 Why do the tools differ in vulnerability reporting?

We list the reasons we identified (with no particular order) through manual analysis behind the differences in the tools' results:

1. **OWASP DC and Com. C detected JavaScript dependencies in Maven projects:** The Maven projects in OpenMRS can also contain front-end JavaScript files. OWASP DC was able to identify dependencies from JavaScript files such as *jquery*, *handlebars*, etc. These dependencies are not resolved by Maven itself or declared in any dependency manifest file. Besides OWASP DC, only Com. C detected JavaScript dependencies in Maven projects. In total, 42 JavaScript dependencies were found by OWASP DC, while Com. C found 20.

2. **Only OWASP DC reported vulnerabilities in internal dependencies:** As mentioned in Section 3.3.2, OpenMRS can have internal dependencies which were reported only by OWASP DC [9]. OWASP DC reported 200 dependencies which are OpenMRS projects. However, these 200 dependencies contain only 14 CVEs and 6 non-CVEs. OpenMRS projects are divided into many submodules and OWASP DC reports the same vulnerability separately for each submodule, which results in an inflation of reported dependencies.

[9] Note that these OpenMRS modules are published as Maven packages, and therefore, a distinction between internal and external dependency in these cases is not mandatory. However, besides OWASP DC, no other tools reported vulnerabilities in OpenMRS modules.

3. **Same vulnerabilities can be repeated over multiple packages:** We observe tools may report the same vulnerability across many related packages, such as dependent packages of a vulnerable package. For example, CVE-2014-3625 was only reported for *spring-webmvc* by MSV, Snyk, Steady, and Commercial A. However, OWASP DC reported this CVE for five separate *spring* packages as NVD simply lists the whole *spring-frameowrk* as affected by this CVE. Conversely, OWASP DC detected functions of npm packages as individual dependencies. In the *lodash* package, OWASP DC detected 31 functions, such as *lodash._baseassign* and *lodash._reevaluate*, separately beside the package itself, and repeated the same 7 vulnerabilities for each of them while other tools simply reported the *lodash* package as vulnerable.

   Prior work reported that relying on the Common Platform Enumeration (CPE) identifier that comes with CVE data may be a reason behind inaccurate vulnerability to package mapping (Kinzer 2015). For example, OWASP DC reported the same 17 CVEs for *activeio-core*, *activemq-core*, and *kahadb* as they all map to the same CPE identifier while other tools only reported *activemq-core*.

4. **Tools may have a different mapping of vulnerability to affected versions of packages:** Incorrect mapping of a vulnerability to the affected version range of a package may result in inaccurate alerts. For example, in *commons-beanutils:1.7.0*, OWASP DC, Com. C and Commercial A reported CVE-2014-0114 and CVE-2019-10086, while MSV reported only CVE-2014-0114; Dependabot reported only CVE-2019-10086, and Snyk reported no CVEs at all. To investigate this difference, we looked into Snyk's vulnerability database [10] and found that Snyk lists the affected version range as $[1.8.0, 1.9.2)$ and $[1.9.2, 1.9.4)$ respectively for the two CVEs and therefore, considers the version OpenMRS uses as free of these vulnerabilities. Similarly, Dependabot lists $[1.8.0, 1.9.2)$ version range as affected for CVE-2014-0114 but all versions below 1.9.4 as affected for CVE-2019-10086. In NVD, the affected versions for the two CVEs are listed simply as up to 1.9.1 and up to 1.9.3. Again, CVE-2014-3576 is reported by Snyk in *activemq-core:5.4.3* but not by Com. C and Commercial A. We see that the Snyk database lists $[0, ]$ version range as affected by the CVE while Commercial A lists $5.8.0 - 5.10.1$.

   Similarly, in the npm ecosystem, CVE-2018-1000620 was detected by all tools except Snyk for *cryptiles:0.2.2*. We found that Snyk's database lists $[3.1.0, 3.1.3)||[4.0.0, 4.1.2)$ range as affected by this CVE. The NVD CVE data simply lists versions up to 4.1.1 as affected by the CVE.

[10] https://snyk.io/vuln

5. **Dependabot reported transitive dependencies through *lock* files:** We notice that Dependabot typically only detects direct dependencies. In the two cases where Dependabot reported transitive dependencies were due to: a) the Maven dependency file explicitly declared the required version for the transitive dependency, and b) the *lock* file was present in the repository that declared the resolved versions of the full dependency tree.

6. **Commercial B only reported vulnerabilities in dependencies under use during runtime:** As Commercial B tracks dependencies through interaction testing as explained in Section 3.4.2, the tool only reported dependencies that were under use by OpenMRS during integration testing, which explains the low count of dependencies reported.

7. **The state of CVEs may result in differences in tools' results:** After a CVE is published, the CVE may become *reserved*, *disputed*, or *rejected* based on new information. We observed that the state of CVEs may be one possible reason behind differences in CVE reporting, as SCA tools need to be timely updated and verify the changes in CVE states. For example, CVE-2019-10768 and CVE-2020-7676 in npm projects were detected by Snyk, Dependabot, and Com. C, but not by OWASP DC and npm audit. However, the latter two tools reported one of them as non-CVEs with a more elaborate explanation. The other CVE is *awaiting reanalysis* (subject to further changes) which may be a possible reason they are not incorporated by the latter tools. Further, we found four rejected CVEs to be reported by Com. C and Snyk, which were not reported by other tools.

8. **Tools can report unique non-CVEs not reported by other tools:** A comparison between non-CVEs across different tools requires manual analysis, as there is no common identifiers. We manually looked at a random sample of dependencies that were reported to have non-CVEs by multiple tools and found that each tool reported non-CVEs that were not reported by any other tools in the study set.

   For example, we observe the following cases in *angular:1.6.1*: OWASP DC reports two *improper input validation* vulnerabilities not reported by any other tool. While Snyk, Com. C and Dependabot reported a similar *XSS* vulnerability, Snyk, and Com. C also reported unique *XSS* not reported by others. Snyk also reported a unique *denial of service* not reported by the other tools. npm audit did not report any of these non-CVEs. We noticed similar differences in non-CVEs for other packages as well, e.g.,

Table 3.6: Code analysis based prioritization metrics: Vulnerable code reachability analysis

| Steady: Static Analysis (Vulnerable code potentially executable) | | | |
|---|---|---|---|
| Total Alerts | Package not in use | Non-vulnerable code of package used | Vulnerable code of package used |
| 2,489 | 2,095 (84.2%) | 340 (13.7%) | 54 (2.1%) |
| **Steady: Dynamic Analysis (Vulnerable code actually executed)** | | | |
| Total Alerts | Package not in use | Non-vulnerable code of package used | Vulnerable code of package used |
| 2,489 | 2,437 (97.9%) | 11 (0.4%) | 41 (1.6%) |
| **Commercial A: Vulnerable call chains** | | | |
| Total Alerts | Vulnerable Method Calls | Total Vulnerable Call Chain | Median Call Chain per Method |
| 2,998 | 31 | 93 | 2.0 |

*lodash* and *ws*.

We identify 8 reasons behind differences in vulnerability reporting among the studied SCA tools, such as inconsistency in vulnerability to affected package version(s) mapping.

### 3.5.3 RQ2: What metrics are presented by SCA tools to aid in the risk assessment of dependency vulnerabilities?

When a vulnerability lies in a dependency, the risk of the vulnerability may need to be determined by how the client application uses the dependency – that is – in the context of the dependency. We have observed that the studied SCA tools reported several metrics in scan reports to aid in a such contextual assessment. We characterize these metrics into five categories:

#### 3.5.3.1 Code analysis-based metrics

Tools may analyze source code or binaries to infer dependency usage and vulnerability reachability. Three of the tools, Steady, Commercial A, and B, use code analysis-based metrics for Java language:

**Reachability Analysis:** Tool can curate their vulnerability database with details on which part of the code (e.g., method, class) is involved in a specific vulnerability. Tools then can infer if the vulnerable code is reachable from the dependent application through static and/or dynamic analysis. Steady and Commercial A provide reachability analysis for each vulnerability in dependency.

Steady constructs static call graphs of an application to infer reachability referred to as ***potentially executable*** (static analysis). Steady also looks at the executability traces through unit testing to determine if the vulnerable code is ***actually executed*** (dynamic analysis). Commercial A, similarly performs static analysis to identify vulnerable call chains – that is – the call chain from the application code that reaches the vulnerable method of the dependency.

Table 3.6 shows the reachability analysis from Steady and Commercial A. We find that for 84.2% of the alerts, Steady did not find the corresponding dependency to be used by the dependent application. Further, Steady found only 2.1% of the alerts were *potentially executable* and 1.6% of the alerts were *actually executed*. However, we found a disconnect between the findings of static and dynamic analysis. Only for 13 alerts, both static and dynamic analysis found the vulnerable code to be in use. Also, for 11 alerts where dynamic analysis found the vulnerable code to be actually executed, the static analysis did not find any part of the dependency containing the vulnerability to be in use at all. This observation may indicate limitations to reachability analysis. Similar to Steady, Commercial A also found a low number of cases where the vulnerable code of dependency can actually be reached from the application source code specifying 93 distinct call chains.

Static analysis, such as call graph construction for Java, is known to have limitations (Sui et al. 2020). The effectiveness of dynamic analysis, such as Steady's, is also dependent on having a good test suite and test coverage. We see that OpenMRS projects reach only around 20% test coverage in Steady. The limited test coverage may have affected Steady's findings.

**Dependency Usage:** The client application may only use a subset of the functionalities offered by a dependency. The code proportion of a dependency used by an application may indicate the probability of a vulnerability being reachable. Steady and Commercial B report how many classes out of the total available are used in a dependency. For example, Commercial B found 203 out of 414 classes (49%) for *spring-web* and 790 out of 4,414 (17.9%) classes for *groovy-all* to have been used by OpenMRS.

#### 3.5.3.2 Package based metrics:

The characteristics of the dependency package itself may indicate the risk associated with it.

**Package security rating:** Commercial B provides a letter grade on their assessment of the security of a package. The tool calculates the security rating of a package based on its age, the count of released versions, and the number of known vulnerabilities. Out of the 17 packages being identified as vulnerable by Commercial B, 16 have an *F* rating, while one has a *D* rating.

#### 3.5.3.3 Dependency characteristics based metrics:

The scope and depth of the dependency may indicate the risk of the vulnerability it contains in the context of the application.

**Dependency scope:** For Maven projects, only Steady reported the scope for each dependency. For npm projects, Snyk and npm audit mentions the dependency scope.

**Dependency depth:** Risk may be associated with how deep a dependency lies within the dependency tree. Only Snyk and Steady indicate if a dependency is direct or transitive for Maven projects. For npm, Only Snyk and npm audit report all possible dependency paths for each vulnerability, indicating the possible depths.

#### 3.5.3.4 Vulnerability based metric:

The characteristics of the vulnerability itself can be used in assessing risk. We found three types of information provided by the tools:

**Severity:** The industry standard for rating the severity of a vulnerability is the Common Vulnerability Scoring System [11] (CVSS), which is publicly available for CVEs. For the non-CVEs, Snyk and Commercial A also present a CVSS score. However, Dependabot and npm audit present a severity rating on a scale of their own for both CVEs and non-CVEs. For both the tools, the scale consists of four levels similar to CVSS3 levels: *low, moderate, high,* and *critical*

**Available exploits:** The availability of known exploits may contribute to assessing the risk of a vulnerability in a dependency. For each vulnerability, Snyk provides information on whether an exploit is publicly available. For the 310 Snyk vulnerabilities, Snyk reports that 218 do not have a public exploit; 10 have a functional exploit; 37 have a proof of concept exploit; 45 have unproven exploits available. Commercial A also reports on available exploits.

**Popularity:** How popular or well-known is a vulnerability may indicate the probability it may get exploited in the wild. Steady integrates Google trend analysis for each vulnerability

[11] `https://www.first.org/cvss/`

in its reports, which indicates the count of search hits within the past 30 days for the CVE identifier.

#### 3.5.3.5 Confidence in alert validity:

SCA tools may provide a confidence rating for each alert, which may aid developers in prioritizing auditing.

**Evidence count:** As OWASP DC detects dependencies from scanning multiple sources, it can provide an evidence count as proof for each dependency. The tool provides a confidence label, from *Low* to *Highest*, based on this evidence count. For all alerts except 4 Maven alerts, OWASP DC provided either a *High* or *Highest* confidence rating.

Our tool survey highlights the research need for a common risk measurement framework specifically for dependency vulnerabilities that can be adopted by the SCA tools.

## 3.6 Discussion

Below, we discuss the observations and implications of our work:

**Resolving dependencies only through dependency manifest files may not provide all the open source code in use:** We find that OWASP DC and Com. C detect JavaScript dependencies in Maven projects through source code scanning, which are not declared in the Maven dependency file. This observation sheds light on the importance of scanning source code, binaries, and deployment environments to resolve all the open source code used by an application. Further, projects can use code fragments from open source packages (Haddad 2020) that should also be identified in order to report known vulnerabilities. Future research is necessary to understand how reliably we can identify all the open source components used in a deployed application.

**Accurate mapping of vulnerability to affected versions of packages should be ensured by the tools to avoid false positives:** We showed examples in Section 3.5.2 on how inconsistency in vulnerability to package mapping can result in differences between tools' results. We also find inconsistencies in what version range is listed as affected for a specific CVE by different tools. Our findings highlight the importance of maintaining the accuracy of the tools' vulnerability databases.

**Non-CVEs should get reported to CVE database to establish their validity and cross-tool vulnerability mapping:** We find that SCA tools report known vulnerabilities in dependencies that do not have a CVE identifier. To understand why the non-CVEs might not have been incorporated into the CVE database, we look at their publication date. Of the 53 non-CVEs reported by Snyk, 41 were published before 2020; while of the 54 non-CVEs reported by Com. C, 50 were published before 2020. Therefore, developers may question why a reported vulnerability does not have a CVE identifier, as CVE validation usually takes around three months. Furthermore, we see that different tools can report the same non-CVEs. However, cross-tool referencing cannot be done automatically without a common identifier like CVE. We suggest reporting the non-CVEs to the CVE database to establish validity, make cross-tool referencing easy, and make the vulnerability widely known.

**Developers may employ more than one tool to leverage different vulnerability databases:** The reporting of unique non-CVEs by the tools highlights the existence of known vulnerabilities in the wild, not necessarily tracked by a centralized database. SCA tools may have delays in incorporating such non-CVEs if not completely missing them. Prior work has shown that unawareness of the vulnerability is a major reason developers do not update vulnerable dependencies (Kula et al. 2018). Therefore, our findings suggest developers may use multiple SCA tools to be timely informed of all the known vulnerabilities.

**Tools should suggest fix options while explaining the risk of any potential backward incompatibility:** Out of the studied tools, Snyk, Dependabot, npm audit, and Com. C offer automated fix suggestions. However, prior work has found that fear of breaking change is one of the primary reasons developers do not want to update vulnerable dependencies (0patch.com 2017). Tools may provide a more in-depth analysis of what code change is there with a certain version change in a dependency and if there are any possibilities of introducing regression bugs.

**Developers may need to evaluate the security risk of dependency vulnerabilities case-by-case basis:** Prior research has shown that not all vulnerabilities in dependency may be relevant for the client application (Pashchenko et al. 2020a,b). We find two tools to offer reachability analysis for each dependency vulnerability. However, future research is required to evaluate if developers in the real world actually use such metrics and find them useful or not. The lack of a common framework for risk assessment among SCA tools suggests that developers will have to evaluate the vulnerability alerts case-by-case basis based on their expertise on the project codebase and how the dependencies are being used by the specific project.

## 3.7 Limitations

We evaluate nine SCA tools on one web application at a certain release point, which poses a threat to the generalizability of our findings. However, OpenMRS consists of 44 projects with 547 Maven and 2,213 npm dependencies, making our case study suitable for a comparative evaluation. Further, the single case study enables us to look in-depth with manual analysis on why the tools' results differed. However, our findings may not generalize to other package ecosystems, such as Ruby or Rust. Similarly, any other application with a difference in dependency management may have yielded different findings. For example, the application that actively maintains its dependencies updated will have a few vulnerable dependencies overall and, therefore, would not show large differences among tools' results. However, that would not invalidate the differences we observed for the studied SCA tools on OpenMRS. Another threat to external validity involves the selection of the SCA tools. We explain our decision criteria in Section 3.4.1. While we are unable to cover all existing SCA tools, we do not claim the findings we have in Section 3.5.2 and 3.5.3 to be exhaustive.

Another limitation of our study involves the absence of ground truth. In the context of SCA alerts, there can be three steps for building such a ground truth - a) determining all the open source dependencies in use; b) determining the correctness of the reported vulnerability data; c) determining exploitability of the dependency vulnerabilities. These steps are nontrivial to perform manually for a real software application and may be more suitable with a synthetic test subject (Delaitre et al. 2018). Regarding exploitability, no tools except Commercial B claim to filter out dependencies on any use criteria, while Steady and Commercial A only offer additional analysis to aid in a such contextual assessment. Therefore, exploitability would not be a fair comparison criterion for the studied SCA tools, and is out-of-scope for this study. Similarly, how developers respond to alerts from SCA tools is also not in the scope of this study. Further, for RQ1, we do not conduct any statistical test to measure the significance of the difference, as we only perform a single case study.

## 3.8 Conclusion

We evaluate nine SCA tools on a large web application composed of Maven (Java) and npm (JavaScript) projects. We find that the tools vary across a wide range in the count of reported unique vulnerabilities and the dependencies that contain these vulnerabilities. Evidence in our findings suggests that accuracy, up-to-dateness, and completeness of vulnerability database are the key strengths of an SCA tool. To this end, automation technologies for continuous monitoring of vulnerability data from package ecosystems are required. Further, SCA tools should be able to pick up open source components beyond what is declared in the dependency manifest files, if any. We find that tools can provide code-analysis-based metrics to assess the risk of dependency vulnerabilities. However, the effectiveness of such an analysis needs to be evaluated. As developers may get overwhelmed with frequent SCA alerts and patching may require extensive regression testing (Radichel 2017), research is required on what metrics and information we can provide the client developers to aid them in assessing and prioritizing the fix of dependency vulnerabilities.

CHAPTER

4

# EMPIRICAL STUDY ON SECURITY RELEASES

## 4.1 Motivation & Research Questions

Scrutiny from security researchers and bug bounty hunters has resulted in a continuous discovery of new vulnerabilities in the open source ecosystem (Coker 2020). However, detecting a vulnerability is only the start of the cure. Once a vulnerability is discovered, the ideal is that a fix will be quickly developed and released in a new version, i.e., security release, so the client projects are able to adopt the fix. Delay in the propagation of security fixes downstream creates a window of opportunity for an attacker to exploit the unpatched vulnerability within the client projects (Li and Paxson 2017; Chinthanet et al. 2021). Therefore, packages should follow good developmental practices to ensure client projects are promptly informed about a vulnerability (e.g., through release notes or publication of a security advisory) and can upgrade to the security release with minimal migration effort (Chinthanet et al. 2021; Pashchenko et al. 2020c; Kula et al. 2018; Martius and Tiefenau 2020).

In this chapter, we study how open source packages release security fixes through four research questions that are relevant to how fast client projects can adopt these fixes. We scope our study to seven popular package ecosystems, namely Composer (PHP), Go (Go), Maven (Java), npm (JavaScript), NuGet (.NET framework: C#, F#, and Visual Basic), pip (Python), and RubyGems (Ruby). Below, we state our research questions:

**RQ1: What is the time lag between a security fix and the publication of a release that includes the fix, i.e., security release?**

**Motivation:** Vulnerabilities may be disclosed either publicly or privately (Ruohonen et al. 2020). However, once applied to the codebase, the vulnerability fix is public by the nature of the open source. Even when the security aspect of a fix is not publicly discussed, research has shown that it is possible to detect security fixes through data mining approaches (Ramsauer et al. 2020; Zhou et al. 2021b,a; Xu et al. 2017). These fixes can give attackers information on how to design exploits. But until the fixes are included in a new release, the client projects may not be able to adopt the fix [1] and remain exposed to the vulnerability, creating a “window of opportunity” for the attackers. Therefore, we quantitatively analyze the fix-to-release delay in past security releases.

**RQ2: How is the security fix documented in the release notes?**

**Motivation:** Once security fixes are included in a release, security advisory databases need to be updated and client projects need to be notified. Paschenko et al. (Pashchenko et al. 2020c) find that “well-documented, well-indicated security fixes that do not require significant development effort are more likely to be adopted by developers”. Therefore, we investigate if security releases come with a release note – that is – a formal document distributed with each version update that explains the notable changes in the new version (Bi et al. 2020; Lacan 2021). We then qualitatively analyze what information regarding the corresponding security fix is documented in the release note if one exists.

**RQ3: What are the code change characteristics of security releases in terms of size and semantic versioning?**

**Motivation:** Once notified, client projects need to migrate to the fixed version of the dependency. The migration process may involve (a) reviewing new code in the update

[1]While patching without an update can be possible, in a typical workflow, developers update the dependencies through package manager tools to pull in new changes including any available security fixes.

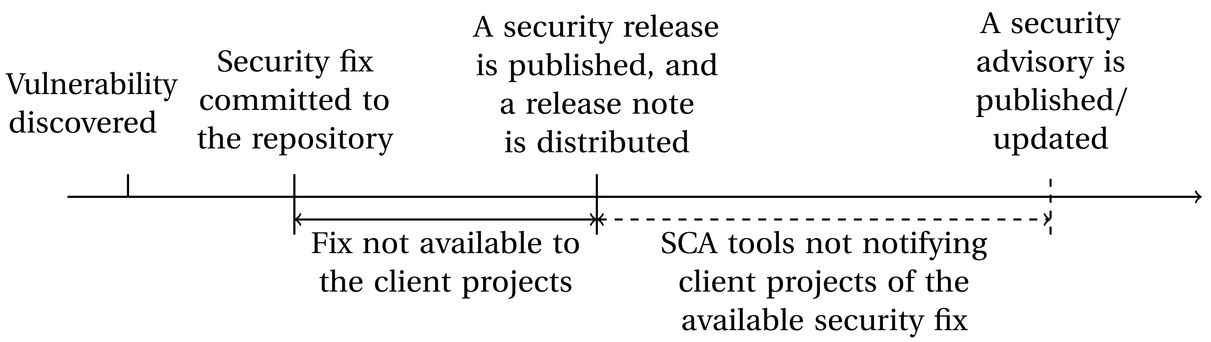

Vulnerability discovered
Security fix committed to the repository
A security release is published, and a release note is distributed
A security advisory is published/ updated
Fix not available to the client projects
SCA tools not notifying client projects of the available security fix

Figure 4.1: A timeline on the possible windows of delay in security fix propagation.

to check against supply chain attacks (e.g., injection of malicious code, backdoors) (Zimmermann et al. 2019); (b) regression testing; and (c) changes in the client code in case of any regression. Prior research finds that developers are more likely to upgrade a package when security fixes are isolated in a separate release (Pashchenko et al. 2020c; Martius and Tiefenau 2020). However, security releases of open source packages often come bundled with unrelated functional changes (Chinthanet et al. 2021). Therefore, we quantitatively measure the code changes in the security releases with an intuition that the large code change indicates functional changes unrelated to the security fix have been bundled in the release. We also look at the semantic versioning (SemVer) (Preston-Werner 2021) of these releases as SemVer format is commonly used by package maintainers to indicate the type of changes that have gone into a new release and the potential migration effort.

**RQ4: What is the time lag between a security release and the publication of an advisory on Snyk and NVD?**

**Motivation:** Client projects may use software composition analysis (SCA) tools, such as Dependabot[2], for notifications on dependency vulnerabilities (Alfadel et al. 2021a). The SCA tools leverage security advisory databases, such as NVD[3] or Snyk[4], to track vulnerability data among open source packages. Therefore, a delay in advisory publication will also cause a delay in notification to the client projects. We measure the time lag between a security release and advisory publication on two popular databases, Snyk[5] and NVD[6], to understand the notification delay from SCA tools (that rely on these two databases).

[2]https://github.com/dependabot
[3]https://nvd.nist.gov/vuln
[4]https://snyk.io/vuln
[5]https://snyk.io/vuln
[6]https://nvd.nist.gov/vuln

To help understand the motivation of our research, Figure 4.1 shows a timeline on how security fixes in the packages become available to the client projects and the possible windows of delays [7]. Note that, a client project may not upgrade to the security release even after getting notified (Chinthanet et al. 2021). However, such delays are out of scope for this study, as we only investigate the security release practices from the root (of the security fix propagation) packages. Conversely, client projects may adopt a security release even when an advisory is not published. However, an advisory publication delay may cause a project to remain unaware of the security fix and consequently, not prioritize the new update [8].

To the best of our knowledge, ours is the first study investigating RQ1, RQ2, and RQ4 for open source packages. Regarding RQ3, Chinthanet et al. (Chinthanet et al. 2021) studied SemVer versioning and lines of code (LOC) change in security releases for the npm ecosystem, and we extend such analysis over seven ecosystems. Code & data for this study is publicly available at `https://figshare.com/s/b0567722400c5b398c61`.

Our **contributions** include:

1. A quantitative analysis of (1) the time lag between fix and release; (2) how security fixes are documented in the release notes; (3) code change characteristics (size and semantic versioning) of the release; and (4) the time lag between the release and an advisory publication for security releases across seven open source package ecosystems;

2. A categorization of the security fix and the breaking change-related information presented in the release notes;

3. Publicly available code & data.

The rest of the chapter is structured as follows: Section 4.2 explains the methodology and dataset. Section 4.3.1, 4.3.2, 4.3.3, and 4.3.4 present the findings of the four research questions. In Section 4.4, we discuss our findings and avenues for future work. Finally, we discuss the limitations of our study in Section 4.5, before concluding our chapter in Section 4.6.

[7] A security advisory can be published before the security release. However, the advisory will still need to be updated with the fixed version to guide in remediation once the fix is released.

[8] Prior work has shown that developers often do not update and keep using the outdated versions of their dependencies (Kula et al. 2018). However, research has also shown that if developers knew that a certain update contained security fixes, they are more like to upgrade to that version (Pashchenko et al. 2020c; Martius and Tiefenau 2020; Kula et al. 2018).

Table 4.1: Methodology overview

| Study dataset |
|---|
| Obtain security releases from Snyk advisory database |
| Identify package repository |
| RQ1 |
| Identify fix commit |
| Identify Release date |
| Measure fix-to-release delay |
| RQ2 |
| Select a sample subset |
| Locate release note |
| Analyze the documentation of security fix |
| Identify if other unrelated changes were mentioned |
| Analyze the documentation of breaking changes |
| RQ3 |
| Identify prior release |
| Download source code |
| Measure code changes |
| Infer SemVer release type |
| RQ4 |
| Filter advisory published after security release |
| Measure advisory publication delay |

## 4.2 Methodology

Table 4.1 shows an overview of our methodology in this chapter. In the following subsections, we explain in detail:

### 4.2.1 Study dataset

Documentation of a vulnerability is typically referred to as a security advisory (Maddox 2010). In this chapter, we work with the Snyk advisory database [9]. Snyk is an SCA tool that reports vulnerabilities in open source dependencies and maintains an advisory database focused on open source packages (Catabi-Kalman 2020). While the full Snyk database is not publicly accessible, a subset of its data remains visible on its web interface. We developed a web scraper to obtain data from the Snyk database. Further, we complement Snyk data with data regarding individual packages fetched through the APIs of the ecosystem registries.

[9] `https://snyk.io/vuln`

#### 4.2.1.1 Advisory data

Metadata on a vulnerability associated with an affected package is considered an advisory on Snyk. Each advisory contains information on: (i) the affected package; (ii) affected versions; (iii) the version where the corresponding vulnerability has been fixed (referred to as security release in this study); (iv) vulnerability type; (v) description; (vi) publication date; (vii) severity level (low, medium, and high) (Snyk 2023); (viii) Common Weakness Enumeration (CWE) type(s) [10]; (ix) external reference links (including links to fix commits (ix) a CVE identifier [11], if the vulnerability has been assigned an identifier. A vulnerability without a CVE identifier is referred to as a **non-CVE** in this study, as was done in Chapter 3. Note that, CVEs are also present in the National Vulnerability Database [12] (NVD). In this chapter, we assume all the CVEs in NVD corresponding to the packages in the studied ecosystems are also present in the Snyk database (Catabi-Kalman 2020). Conversely, Non-CVEs are only present in Snyk, and not present in NVD.

We collected data on March 5, 2021 [13]. We collected 6,956 advisories across eight ecosystems: CocoaPods (Objective-C), Composer (PHP), Go (Go), Maven (Java), npm (JavaScript), NuGet (.NET framework: C#, F#, and Visual Basic), pip (Python), and RubyGems (Ruby). From this initial set, we exclude i) 589 advisories (8.5%) related to malicious packages [14]; ii) advisories from the CocoaPods ecosystem (358 advisories (5.2%), as they cover only 37 unique packages (mostly from *ImageMagick*); iii) 1,233 advisories that do not have a security release listed by Snyk in our data.

**Advisory publication date:** In this chapter, we refer to the advisory publication date as the publication date on either Snyk [15] or NVD [16]. For the CVEs, we take the earlier publish date among NVD and Snyk as the advisory publication date. For the non-CVEs, we take publish date on Snyk as the advisory publication date.

[10] `https://cwe.mitre.org/`

[11] `https://cve.mitre.org/cve/`

[12] `https://nvd.nist.gov/vuln`

[13] Advisory databases are continuously updated. Our data only reflects the state of the Snyk database on March 5, 2021.

[14] The Snyk advisory type is *malicious package*. These packages got revoked from the package registry and do not have a fixing security release, and therefore, out of the scope of this study.

[15] `https://snyk.io/vuln`

[16] `https://nvd.nist.gov/vuln`

Table 4.2: Study data set overview (Each RQs leverage a subset of this data set).

| Ecosystem | Advisories | Packages | Security Releases | # of unique CWE types | Severity* | advisory with CVEs | advisory with Non-CVEs |
|---|---|---|---|---|---|---|---|
| Composer | 855 | 228 | 976 | 68 | | 599 (70.1%) | 256 (29.9%) |
| Go | 235 | 183 | 293 | 50 | | 190 (80.9%) | 45 (19.1%) |
| Maven | 1,374 | 694 | 1,607 | 112 | | 1,132 (82.4%) | 242 (17.6%) |
| npm | 792 | 540 | 858 | 82 | | 474 (59.8%) | 318 (40.2%) |
| NuGet | 333 | 92 | 233 | 34 | | 277 (83.2%) | 56 (16.8%) |
| pip | 567 | 269 | 534 | 84 | | 404 (71.3%) | 163 (28.7%) |
| RubyGems | 221 | 121 | 311 | 47 | | 168 (76.0%) | 53 (24.0%) |
| All | 4,377 | 2,127 | 4,812 | 172 | | 3,244 (74.1%) | 1,133 (25.9%) |

Severity levels: low, medium, high

### 4.2.1.2 Identifying package repository

For the analysis of the research questions in this study, we require the source code repository of the packages. We obtain metadata for each package through APIs of each of the seven ecosystem registries and collect repository URLs if present in the metadata. We used Sonatype's API [17] for Maven packages, while the other six package registries have their own APIs. We only considered git [18] repositories in this study [19]. We could not identify the repository URL for 300 packages, which further excludes 399 advisories from the collected dataset [20]

### 4.2.1.3 Dataset summary

Table 4.2 shows the final dataset used in this study. The dataset contains 4,377 advisories covering 4,812 distinct security releases from 2,217 packages. Open source packages can maintain multiple branches at the same time (e.g., 1.x.x and 2.x.x) and may push a security fix individually for all the maintained branches. Additionally, a single release can contain a fix for multiple advisories. In our dataset, 1,231 advisories (28.1%) had more than one security release, while 738 releases (15.3%) contained fixes for more than one advisory. The table also shows the percentage of advisories with and without a CVE identifier.

[17]`https://central.sonatype.org/search/rest-api-guide/`

[18]`https://git-scm.com/`

[19]Besides git, we only encountered subversion projects (Alali et al. 2008) for Apache and OpenSymphony projects. We manually looked for their GitHub mirrors and use the GitHub repository URL if we found one.

[20]177 packages (59.0%) are from *jenkins* organization for whom Sonatype API does not list the repositories. Note that, we refrained from manually locating the repository, as there can exist multiple forks for the same project. Therefore, we follow a uniform methodology to locate repository only via package metadata obtained from the respective registries for all the packages.

Our advisory dataset contains 25.9% non-CVEs which provides an opportunity to compare the characteristics between CVEs and non-CVEs. Further, over 80% of the advisories (3,505) were disclosed on or after the year 2016, which makes our analysis representative of recent data. However, for each of the three RQs individually, we would require additional data, which would further restrict our dataset, explained in the following three subsections.

For RQ1 and RQ3, we followed a conservative automated approach in identifying fix commits for an advisory and code change in a security release. For RQ2, we performed manual analysis over a randomly sampled data subset. In the following subsections, we explain the data set for each of the three RQs. We also present a data accuracy check by manually validating a random sample of the data set for RQ1 and RQ3. Based on our manual validation, we present the accuracy rate of our dataset using Agresti and Coull's score confidence interval (Agresti and Franklin 2007) (C.I.) with a 95% confidence level.

### 4.2.2 RQ1: Fix-to-Release Delay

In RQ1, we study the time lag between the fix commit and the publication date of the subsequent release that includes the fix. We explain how we collected fix commits and release dates for each advisory.

#### 4.2.2.1 Identifying fix commit

Fix commits are the commits (code changes) that fix a vulnerability. Snyk, in its external reference URLs, contains fix commit information with associated tags like 'GitHub commit', 'Fix Commit', 'GitHub PR' pointing to the links to commit(s) and pull request(s) that have fixed the corresponding vulnerability. To identify fix commit information for an advisory from the external reference URLs, we look for: (i) *commit* sub-string in the tag or URL; and (ii) *pull* or *PR* sub-string in the tag and */pull/* sub-string in the URL.

For commits, the URLs contain the commit hash identifier. For pull requests, we only look at the GitHub links and use the GitHub API to list the commits under that pull request. We verify the validity of a fix commit if the following conditions are met:

1. The commit is present in the package repository, and the reference URL points to the package repository.

2. The commit is not present in the package repository, but the reference URL points to the package repository. Such a case arises when a commit is squashed or the corresponding branch is deleted so that the commit is no longer referenced in the

repository. However, the commit will still be viewable on GitHub [21]. We verified the validity of such commits through the GitHub API.

3. The commit is present in the package repository, but the reference URL points to a different repository. Such a case arises when the same code is shared over multiple repositories. In these cases, we considered the commit valid if the commit message in the package repository matches the one in the reference URL, therefore, ensuring they are the same commit.

If the fix commit(s) is valid, we retrieve the time of the commit. All time-related data in this study are converted to UTC timezone for comparison. Through this process, we identified fix commits for 2,497 (of 4,377) advisories.

#### 4.2.2.2 Identifying release date

As explained in Section 4.2.1.2, we use ecosystem registry-specific APIs to retrieve release dates, except for Maven and Go. For Maven, we crawled package version information from the Maven central repository [22]. For the Go language, the packages do not publish new versions on any central registry. Instead, a new version is published by pushing a tag to the repository [23], and client projects can pull Go packages directly from repository URLs. Therefore, for Go packages, we looked at the tags in the respective package repositories and retrieved the tag annotation date where the tag matches the release version.

For the advisories, we had fix commit for 75 Maven packages that cover 225 advisories are not hosted on Maven central repository and therefore, we excluded them from our analysis. From the remaining, we could not resolve the publication date for 357 security releases. While investigating, we found the primary reason is that the fixing version is not registered in the respective package registry. While it may be technically possible to pull a package version directly from the source, we have excluded such releases in this study as client projects would not be able to pull these releases through the package manager, which is the usual scenario. After this step, we had 2,068 advisories for analysis of RQ1.

[21] `https://docs.github.com/en/github/committing-changes-to-your-project/commit-exists-on-github-but-not-in-my-local-clone#the-branch-that-contained-the-commit-was-deleted`

[22] `https://repo1.maven.org/maven2/`

[23] `https://github.com/golang/go/wiki/Modules#publishing-a-release`

#### 4.2.2.3 Data accuracy check

For 67 advisories, we found that the release date is before the fix commit date. While investigating, we found that the inaccurate listing of the security release version and fix commit by Snyk is the primary reason. For the rest of the 2,001 advisories, we perform an accuracy check over a randomly sampled set of 100 advisories (5%) by checking the correctness of (i) the fix commit, (i) the first version where the fix is included, (ii) and release date of the version.

For 98 data points, we identified our collected data to be accurate. For 1, the fix was already included in an earlier release than the version listed by Snyk (NVD [24] for the corresponding CVE also has the wrong version listed). For another, the release date on Maven central repository is at a later time than the date when the corresponding version was tagged in the source repository (a possible explanation is that the package version was uploaded/updated at a later time in the Maven repository by the package maintainers). Our check provides an indication of the accuracy of our dataset (accuracy between 95.3% and 100% with 95% C.I.). We select these 2,001 advisories to answer RQ1.

#### 4.2.2.4 Data analysis methodology

We have the fix commit information and publication date of the security release for 2,001 advisories. As one advisory may have a security release in multiple release branches, we have 2,640 distinct advisory-security release pairs for the analysis of RQ1. We identified more than one commit fixing the corresponding vulnerability for 806 advisories (38.9%). In that case, we considered the commit date for the last of those commits as an indicator of the end of the fix. We then take the time gap between the fix commit date and the publication date of the security release and present it in a unit of days.

Besides fix-to-release delay, we also measure if an advisory was published on Snyk/NVD before a fix was made, or the fix was released, from the same motivation that an early circulation of the vulnerability gives attackers a window of opportunity [25]. To determine the advisories published before a security release, we take the earliest release date in cases when an advisory has multiple releases in different branches. Further, we test if the severity level provided by Snyk and an advisory having a CVE identifier impacts the fix-to-release delay using Mann-Whitney U statistical test (Mann and Whitney 1947). We perform Bonferroni

[24] `https://nvd.nist.gov/vuln`

[25] A vulnerability can be published as an advisory elsewhere earlier to Snyk and NVD, e.g. GitHub security advisory database. However, such a case does not change the findings on advisories that were published on Snyk/NVD before a fix or a release. Therefore, our findings here maybe only an under-approximation.

correction (Weisstein 2004) by setting $\alpha = .025$ (as we perform two hypothesis tests on the same data) to reject the null hypothesis that there lies no difference between the two tested populations.

### 4.2.3 RQ2: Documentation

We investigate if the security fixes are mentioned in the release notes for open source security releases and if they are mentioned, then what information about the security fix is present. We also investigate if other changes unrelated to the security fix are mentioned in the release note, including the inclusion of breaking changes (i.e. changes that may trigger rework for the client projects (Bogart et al. 2016; Preston-Werner 2021)). The latter two analyses (of unrelated changes and breaking changes) will help us understand the potential migration effort required by the security releases, and we can triangulate the findings from the release notes with the analysis for RQ3. Further, we also investigated if the vulnerability severity impacts if the fix would be documented in the release note or not.

#### 4.2.3.1 Dataset

In RQ2, we qualitatively analyze the release notes associated with the security releases over a subset of the study data set. We randomly sample 25 CVEs and 25 non-CVEs from each ecosystem, disclosed on or after 2018. NuGet and RubyGems only have 15 and 14 non-CVEs disclosed on or after 2018 in our dataset. Therefore, we took non-CVEs from 2017 to fill the gap. In total, we analyze 350 advisories covering 465 security releases. We focus on selecting recent security releases to capture the current trend. As the methodology for RQ2 involves manual analysis, we perform an accuracy check (and make corrections if needed) at the same time during the analysis.

#### 4.2.3.2 RQ2 sub-questions

To answer this research question, we manually explore the possible locations of release notes (Lacan 2021; Bi et al. 2020) and identify if a security release had an associated release note. If there is a release note, we qualitatively analyze the document through open coding (Khandkar 2009) (explained in the following subsection). We ask the following sub-questions to answer RQ2:

1. How many of the security releases contain a release note? What are the sources to find the release notes?

2. How many of the release notes mention the security fix? What information regarding the security fix is present in the release notes?
3. How many of the release notes mention changes unrelated to the security fix?
4. How many of the release notes mention breaking changes? What information regarding the breaking change is present in the release notes?

We answer these sub-questions for 350 randomly-sampled advisories as explained in Section 4.2.3.1. While these advisories have 465 distinct security release notes, one release can contain fixes to multiple advisories. We answer RQ2 by considering a security release for each advisory as individual data points, which gives us 499 distinct advisory-release pairs. In the following subsection, we explain our methodology to answer RQ2 for 499 data points [26].

#### 4.2.3.3 Data analysis methodology

For each security release and advisory, we locate if there is a release note and analyze the presence of a description related to the security fix, changes unrelated to the security fix, and any breaking changes in the release note. Our methodology consists of four steps, as explained below. The first and the second author independently performed these steps for all 499 data points and resolved disagreements through discussion to finalize the manual analysis. Below we explain each of these four steps followed by an explanation of *the open coding* technique and agreement rate measurement.

**Step I- Locating release note:** To locate if a security release contained a release note, we performed a combination of automated and manual exploratory searches, divided into four steps as explained below. While the first three steps were automated and were done for all the releases, we also performed a manual search in case we failed to locate the release note automatically.

1. We search for possible changelog documentation files within the source code repository. Following *keep changelog* (Lacan 2021) suggestions, we search for files with keywords 'change', 'history', 'news', and 'release' under documentation formats - .md, .txt, .rst, .adoc, .org, .html, .rdoc, or if the format is unspecified.

[26]While a single release can contain multiple security advisory fixes, we present our findings for RQ2 over each advisory. Therefore, we take 499 distinct advisory-release data points as the base data set for RQ2.

2. The GitHub platform offers a release note option for the hosted repositories. We identify GitHub releases through the version (and package name, if the repository contains multiple packages) string and look if the security release has a GitHub release note.

3. Packages can tag a commit as a release point. We look if the release tag has an associated tag message.

4. If we could not find a release note in the prior steps, we browse through a) external links provided in the project README file, e.g., project homepage; and/or b) package listing in the corresponding package manager site, and manually search if we can find a release note.

Package maintainers can put a release note in multiple locations, e.g, in both changelog and GitHub release notes. In such cases, we consider the note with the longest description (e.g., GitHub release notes can have an additional description on the update than what is written in the changelog). More commonly, the same note is copied in all the locations. In those cases, we give preference to the changelog, as changelog files are stored within the source repository and linked to other external locations.

Moreover, the tag message or a GitHub release note can be auto-generated, such as *version x.y.z*, without a textual description of the changes in the release. We do not consider such cases as release notes. If we can identify a valid note for a release, we perform the rest of the following steps.

**Step II- Analyzing the documentation of security fix:** Based on the vulnerability description and external reference links provided in a Snyk advisory, we classify if the release note mentions the fix for the corresponding vulnerability. To categorize what information regarding the security fix is documented, we apply open coding (Khandkar 2009). In cases where the Snyk advisory does not provide sufficient information and the release note does not explicitly state the vulnerability, we were unable to definitively decide if the security fix is mentioned. We classified these as unmentioned, with the rationale that a typical reader would not be able to understand that the release contains a security fix.

**Step III- Identifying mention of unrelated changes:** We classify if the release note mentions any other change than the fix for the corresponding vulnerability. A release can contain multiple security fixes. However, if we found any other change than the fix for the corresponding advisory, we classified them as unrelated changes with the rationale that unless explicitly stated, we cannot independently verify if the other changes are a security fix or not. However, we keep notes when a release contains multiple security fixes, and the release note only mentions the security fixes (and no security irrelevant changes).

**Step IV- Analyzing the documentation of breaking change:** We analyze if the release note mentions any breaking change that may trigger rework for the client projects, and categorize what information about the breaking change is documented through open coding.

**Open coding:** Open coding is a qualitative data analysis technique to create emergent concepts (codes) from textual data without any pre-defined concepts at the beginning of the process (Khandkar 2009; Hancock et al. 2001). In our study, the coders [27] independently *coded* the information documented in the release notes regarding security fixes and breaking changes, which upon discussion, were grouped into categories. The coders had their first discussion after an analysis of 70 npm data points. Afterward, whenever a coder identified a possible new category, the category was discussed with the other coder. If the category was accepted by both coders, the data points already analyzed were revisited to identify instances of the new category. In this way, the two coders independently followed an *iterative process* as suggested in prior work (Wicks 2017; Pashchenko et al. 2020c). Finally, we resolved the codes where we disagreed through discussion to finalize the analysis for all 499 data points.

### 4.2.4 RQ3: Code Change Characteristics

In RQ3, we investigate the code change in a security release since its prior release. In this section, we explain (i) how we determine the prior release of a security release; and (ii) how we collected the source code for each release version.

#### 4.2.4.1 Identifying prior release

Determining the prior release of a version is different from simply chronologically ordering all the released versions, as packages can maintain multiple branches at the same time. Each package ecosystem follows a formatting algorithm to determine version order. APIs for Composer, npm, NuGet, and RubyGems return the version list in an ordered format according to their respective algorithms, which we leveraged to determine the prior release. However, Maven and Pip return the list of versions sorted in alphabetic order. We used

[27] the first and second author of this study

Python library, *packaging* [28], and *mvn-compare* [29], to sort versions for pip and Maven, respectively. We found that Maven packages can contain version names not in accordance with Maven guidelines [30], e.g. *1.532.2.JENKINS-22395-diag*. We discarded 234 packages that did not follow Maven naming guidelines for RQ3 analysis, as there is no reliable automatic method to determine the prior release for their security releases [31].

Go packages follow SemVer (Preston-Werner 2021) formatting, which we leveraged to identify prior releases from the version tags available in their source repository. We excluded packages with tags that contained prefixes and/or suffix strings we were unable to parse according to the SemVer formatting. We also found and excluded packages where all metadata regarding versions prior to the security release were removed from the package registry. After this step, we could identify the prior release of 3,856 security releases over 3,520 advisories out of the 4,377 in the initial study data set.

#### 4.2.4.2 Downloading source code

For each security release and its prior release version of a package, we download the source code directly from the respective package registries, except for Go and NuGet. While Go packages are not published in any central registry, the NuGet registry contains only the compiled artifact. Therefore, we followed a heuristic-based approach to identify package-specific code from the source code repositories of the Go and NuGet packages. We explain our approach below:

We identify code for a specific version by pattern matching the version with the tag strings in the source code repositories [32]. However, one repository may contain multiple packages. Go packages are named according to their exact location within the repository (e.g. *github.com/go-gitea/gitea/modules/auth*), which we used to identify package-specific code. NuGet packages do not contain such information in their metadata. Therefore, for NuGet packages, we apply a simple heuristic of checking if any subdirectory within a repository

[28]`https://pypi.org/project/packaging/`

[29]`https://pypi.org/project/mvn-compare/`

[30]`https://maven.apache.org/ref/3.3.3/maven-artifact/apidocs/org/apache/maven/artifact/versioning/ComparableVersion.html`

[31]While the security release itself may follow Maven formatting, due to the presence of at least one version that does not follow Maven formatting, we were unable to automatically determine the prior release from the full release list.

[32]Packages typically follow standard formats of putting version string as the tag name, often prefixed by 'v' (e.g. *v1.0.13*). In cases, where a repository contains multiple packages that are released separately, the tag name typically also contains the package name. We only included a security release if we were able to determine the tag version following these two heuristics.

matches the package name. If it matches, which is often the case as per typical formatting in NuGet package repositories, we only measure the code within the subdirectory. Otherwise, we consider the full repository.

For Go and NuGet packages, we excluded security releases for which we could not identify the version tag and corresponding commit, either for the security release or its prior release versions. For the other five ecosystems, we excluded security releases if we were unable to identify a valid download URL for the source code for any of the versions. After this step, overall, we could successfully download source code for 3,445 security releases (and respective prior releases) over 3,171 advisories (out of 3,520 advisories from the prior step).

#### 4.2.4.3 Measuring code change

We measure the code change in a security release through the 'git diff' tool, the default code change measuring approach in git repositories [33]. For 64 Go security releases, we found that the release does not change any file within the affected package itself. Repositories for Go packages often contain many related packages (*modules,* as referred to in the Go ecosystem) that follow a single versioning scheme. Therefore, a vulnerability may get fixed in a Go module by changing code in a related module when the modules are used together. However, we exclude these 64 Go releases to avoid under-approximation in our analysis.

We also noticed that packages often contain non-source-code files, such as documentation, configuration, auto-generated, and resource (e.g. images) files. Changes in non-source code files often inflate the code change measurement. Therefore, we separate the source code files and measure only source code change. We used the file formats to filter out non-source-code file changes. Our dataset includes 1,270 distinct file formats. The first author manually inspected the 182 formats for which the corresponding files were changed at least 100 times (99% of the total file change count in our data set) and classified 42 formats as source-code files across the seven ecosystems [34]. Our procedure follows prior work on measuring code changes in security fixes by Li and Paxson (Li and Paxson 2017).

For 136 releases, we found that the code change does not modify any source code files. Investigating 10 randomly sampled cases, we find that - (i) in 6 cases, the package

[33]We ignore blank lines while measuring code change using the native git-diff feature. However, we do not ignore comment lines to keep the methodology consistent across the seven language ecosystems, as the native 'git diff' tool does not have the comment line filtering feature.

[34]We used the below file formats as source code files (and exclude any other file format as non-source-code files for RQ3 analysis): *js, ts, java, sh, swift, tsx, h, cc, jsx, php, vue, coffee, c, m, bat, as, py, patch, ps1, rb, cpp, hpp, pl, sql, thrift, cs, go, hx, pm, groovy, scala, asm, jsp, bats, factories, erb, phpt, s, cxx, fs, vb, sol.*

Table 4.3: Excluded security releases for RQ3 analysis

| Advisories | Security releases | Reason |
|---|---|---|
| 857 | 956 (19.7%) | Unable to identify the prior release of the security release |
| 349 | 411 (8.5%) | Unable to obtain source code of both the releases |
| 143 | 207 (4.3%) | Source code files not modified in the security release |

made changes to its dependency versioning in the configuration files (a possible case of vulnerable dependency); (ii) in 2 cases, changes were in data files (e.g., *yaml* file that stores regex patterns); and (iii) in 2 cases, the prior release already contained the fix while the security release listed by Snyk only adds a security notice in the release notes. However, we excluded configuration files as their file formats often share the same formats as auto-generated files (e.g. *json*, *xml*), and auto-generated files often contain large changes between releases. Further, we do not focus on fixes for vulnerable dependencies in this study.

Finally, we select 3,238 releases over 3,028 advisories for RQ3 analysis [35]. Table 4.3 shows excluded security releases and the reasons we were unable to analyze them in each of the three steps explained in this subsection.

#### 4.2.4.4 Data accuracy check

We randomly sampled 15 security releases from each ecosystem to verify our collected data, which is in total 105 releases ($\approx 3.2\%$). Specifically, we verify (i) the prior release; and (ii) code change measurement by manually exploring respective source code repositories. For 101 data points, we verified our collected data to be accurate. In 2 npm releases, file renaming, and auto-generated code inflated code change measurement. In 1 npm release, the package itself is small and downloads binaries (compiled from C code) during package installation. Therefore, while the code change in the package code is small, the effective code changes are presumably in the binaries, which we did not measure. In 1 Composer release, GitHub failed to compare the two releases, claiming they come from different commit histories. However, our analysis measured large changes (over 4000 files modified). Our check (accuracy between 92.5% and 99.8% with 95% C.I.) provides an indication of the reliability of RQ3 findings.

[35] We developed a python package, *version-differ*, that measures the code diff between two versions of a package as per our study methodology, available at `https://pypi.org/project/version-differ/`.

#### 4.2.4.5 Data analysis methodology

We measure the count of changed files and lines of code (LOC) in a security release and parse the release version to determine the SemVer release type. We explain the three metrics below:

1. **Files modified:** As explained in Section 4.2.4.3, we filter out the non-source-code files (e.g. configuration, documentation, resource files) and count only the source code files that have been modified in the security release.

2. **Lines of Code (LOC) change:** For the source code files that have been modified in the security release, we take the count of lines added and removed as the total LOC change.

3. **SemVer release type:** SemVer (Preston-Werner 2021) format consists of three component numbers, X.Y.Z. All versions below 1.0.0 ($X < 1$) are considered **unstable release** where developers are not expected to maintain backward compatibility for the public APIs that the package provides. Starting from 1.0.0, a change in X indicates a **major release** – that is – a backward-incompatible release. A change in Y indicates a **minor release** where new functionalities have been added in a backward-compatible manner. Finally, any change in Z indicates a **patch release** – that is – a release containing only backward compatible bug fixes. Therefore, releases with security fixes should typically come out as patch releases.

   Further, the version can be suffixed with qualifier strings like 'pre', 'beta', 'rc' to indicate a **pre-release**. While only npm strictly enforces SemVer formatting, open source packages in the studied ecosystems typically follow this versioning format. We parsed the version strings as major, minor, patch, unstable, and pre-release according to this format. If we were unable to parse a version in SemVer format, we exclude the release from SemVer measurement.

### 4.2.5 RQ4: Advisory Publication Delay

In Section 4.2.1, we explained how we determine the advisory publication date (either on Snyk or NVD, whichever is earlier). Further, in Section 4.2.2.2, we retrieved release dates for the security releases in our data set. Overall, we have a security release date and advisory publication date for 3,655 advisories which we analyze to answer RQ4. For these advisories, we analyze RQ4 in two steps: (i) distinguish the advisories that were published before and after the security release; and (ii) quantify the advisory publication delay for the advisories that were published after the security release.

### 4.2.6 Impact of vulnerability Severity

We investigate if the severity of the vulnerability impacts the package maintainers' approach to releasing a security fix (as measured through the four RQs in this study). The rationale behind this investigation is that not all vulnerabilities may pose similar security risks. Hence, package maintainers may handle critical vulnerabilities with more attention than the low severity ones.

To measure severity, we look at two metrics: (i) the severity level provided in the Snyk advisory that is based on CVSS [36] framework version 3.1 (Snyk 2023); and (ii) if the advisory is also published on NVD (as a CVE), with an assumption that the critical vulnerabilities are reported more widely (in more than one advisory database in this case).

### 4.2.7 Ecosystem-specific analysis

For each of the RQs, we present our measurement individually across each of the seven studied package ecosystems. In modern software development, a project depends on many direct and transitive packages within a particular ecosystem. For example, a Python project pulls in pip packages, whereas a Ruby project pulls in RubyGems packages. Further, through complex dependency chains, a project may transitively pull in many packages, large and small, and from a diverse group of maintainers, even when the project directly depends on only a few packages (Zimmermann et al. 2019). Therefore, researchers have studied the security hygiene and practices individually for different ecosystems (Zimmermann et al. 2019; Alfadel et al. 2021a). While our methodology in this study is language-agnostic, therefore, enabling us to study security releases across multiple ecosystems, we present our findings separately for each ecosystem to quantify how each community handles security releases.

## 4.3 Findings

In the following four subsections, we present our findings for each of our RQs:

[36] `https://www.first.org/cvss/`

Table 4.4: Analysis of time lag between a security fix and the subsequent release that includes the fix (RQ1)

| Ecosystem | Advisories | Packages | Security Releases* | # of unique CWE types | Fix-to-release delay (median days) | Occurrences of advisory published before fixed | Occurrences of advisory published before fix released |
|---|---|---|---|---|---|---|---|
| Composer | 183 | 93 | 204 (295) | 40 | 1 | 13 (7.1%) | 27 (14.8%) |
| Go | 184 | 150 | 224 (242) | 46 | 6 | 21 (11.4%) | 48 (26.1%) |
| Maven | 574 | 385 | 745 (807) | 83 | 13 | 40 (7.0%) | 153 (26.7%) |
| npm | 553 | 412 | 586 (640) | 72 | 0 | 72 (13.0%) | 113 (20.4%) |
| NuGet | 70 | 22 | 52 (75) | 10 | 3 | 8 (11.4%) | 14 (20.0%) |
| pip | 324 | 200 | 308 (442) | 68 | 6 | 39 (12.0%) | 79 (24.4%) |
| RubyGems | 113 | 80 | 121 (139) | 41 | 3 | 9 (8.0%) | 15 (13.3%) |
| Total | 2,001 | 1,342 | 2,240 (2,640) | 143 | 4 | 202 (10.1%) | 449 (22.4%) |

* A single release may contain multiple advisory fixes. The number in the parenthesis shows the count for distinct advisory and security release pairs for which we have a fix to release delay.

### 4.3.1 RQ1: Fix-To-Release Delay

We report our quantitative findings for RQ1, followed by a manual analysis to explain the large delays, and statistical tests to measure the impact of vulnerability severity.

#### 4.3.1.1 Quantitative findings

Table 4.4 shows the median fix-to-release time delay for all the studied open source packages and also individually across each ecosystem, while Figure 4.2 shows a box plot in the logarithmic scale. Across all the ecosystems, we find that the median release comes within 4 days of the corresponding security fix (0 and 20 days for the first and the third quartile, respectively). We find npm packages to have the lowest median delay (50.3% of the security releases to come under 24 hours of the fix), and Maven packages to have the highest median delay (only 14.9% of the releases coming out under 24 hours).

We find that advisories can be published on Snyk/NVD before a security fix is released in a new update, or can be published even before a fix has been made. As shown in Table 4.4, we find that 22.4% of the advisories were published on Snyk/NVD before the security fix was released, and 10.1% were published even before the fix was committed. When an advisory is published before a fix is committed, we find the security release to come within 32 days of the advisory publication at a median. When an advisory is published after a fix has been made, but before the fix is released in a new version, we find the release to come within 17 days of the advisory publication at a median. A possible explanation behind this delay is

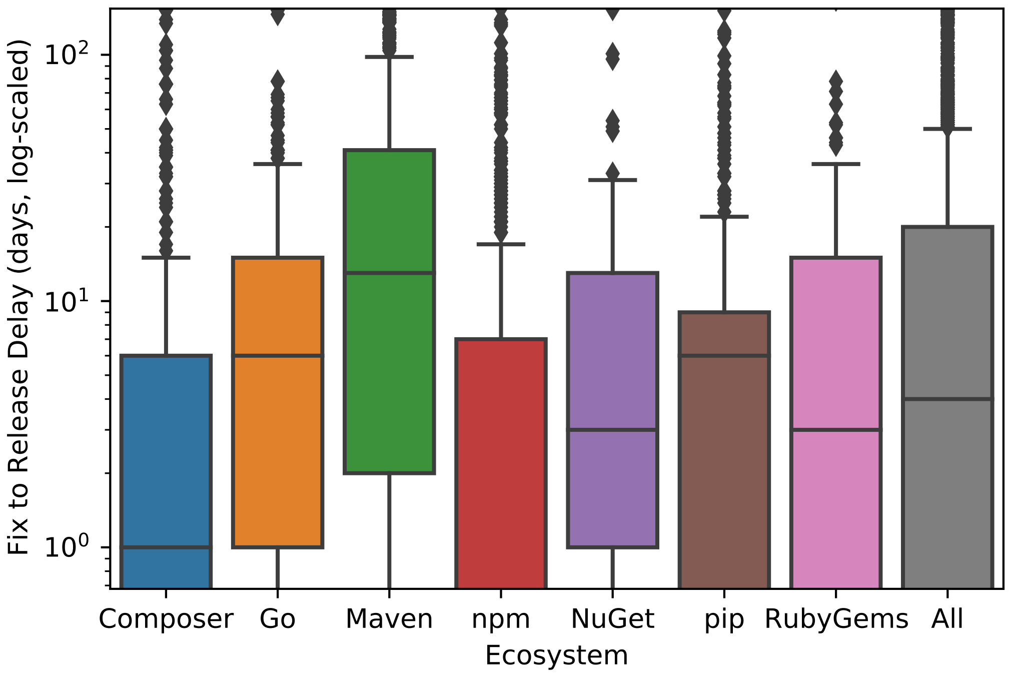


Figure 4.2: Box plot for fix-to-release delay (log-scaled) for each ecosystem. Note that, the median for npm stands at zero.

that the reporters of the vulnerabilities (e.g., bug hunters) shared the vulnerability in public forums and reported to the advisory database without providing the package maintainers sufficient time to issue a security release first, an *irresponsible* disclosure method.

The "window of opportunity" can be perceived to be riskier in the case where an advisory is published before the corresponding fix is released, as the advisory circulates vulnerability information. However, given that the development of open source is public by nature, a delay in releasing a security fix in any scenario carries a risk as attackers may become aware of the vulnerability by monitoring package development, but client projects may remain unaware and not take any mitigation approach if applicable (Hat 2015).

Further, 392 (19.6%) of the advisories in our data set had fixes released in multiple branches, while 203 of them (51.8%) of them had all the fixing releases published on the same day. For the rest, the median time delay between the earliest and the latest releases (in different branches) was 8 days. Our data suggest that security fixes are pushed at a similar time in all the maintained branches.

#### 4.3.1.2 Understanding large delays

To understand the large fix-to-release delay, we look at 50 randomly-sampled data points (from 680 data points) where the security release came at least 20 days after the corresponding fix, as such releases constitute 25% of our dataset (the fourth quartile). We find in 6 cases that our collected data was inaccurate. In 4 cases, the fix came earlier than the version Snyk listed; in 1 case, the listed fix commit was incorrect; and in 1 case, the Maven central repository listed a date for the security release later than what appeared on the repository [37].

For the remaining 44 data points, we manually explored the release date, advisory publication date, release note, and the relevant commit messages and issue-tracker discussion to understand the reasons behind the large delay. We present our observations below:

1. **Advisory published after the security release (27):** In 27 cases, the advisory was only published once the security fix was included in a new release (10 of these advisories were published at least a year after the security release). However, for 14 of these advisories, we found that the commit message or the issue tracker discussion explicitly mentioned the security aspect of the fix. While the package maintainers may not have rushed a new release as an advisory was not yet published, such public discussion on security fixes make a vulnerability easily discoverable (e.g. by potential attackers), and validates our motivation behind RQ1. Further, in 2 of these cases, we noticed a long delay (9 months and 48 days) in the approval of the pull request making the security fix [38].

2. **Advisory published after a fix has been committed, but before the release (8):** In 8 cases, the advisory was published once a fix for the corresponding vulnerability was committed in the source repository, but before the fix was released in a new version. We could not identify any reasoning behind this phenomenon, besides that the package maintainers may not have prioritized the task of releasing the security fix in a new version.

3. **Delay in backporting the security fix (5):** In five cases, we found the security fix was included quickly only in the latest release branch of the respective package (and an advisory was published only after that). However, the delay occurred in backporting

[37]While we present our accuracy check for RQ1 data in Section 4.2.2.3 (96%), our results indicate that outlier points are more likely to have data reporting inaccuracies.

[38]Pull requests are often merged by creating a merge commit which we then consider as the fix commit. So the fix-to-release delay reporting in this study may or may not include any delay in the pull request approval process.

the fix to an older release branch. In this case, client projects who are not using a version from the latest branch of a package will face a delay in receiving the security fix.

4. **Advisory published before fix (4):** In four cases, the advisory was published even before a fix was made. In two of these cases, the respective package maintainers initially suggested that it was not a vulnerability in their package, but rather a misuse of the package. In one case, the advisory was published on a security platform, presumably without giving the maintainers a time window to work on a fix. However, the maintainers responded quickly afterward. In another case, the package maintainer mentioned that the vulnerability is only relevant in the case of legacy browsers (and, therefore, not a high priority).

Overall, we find that package maintainers may delay releasing a security fix, as they may not plan on publishing an advisory until the release anyway. However, observant attackers can still discover such security fixes through commit messages and issue-tracker discussions.

#### 4.3.1.3 Impact of vulnerability severity

**Fixes for high severity advisories get released faster than medium severity ones:** We observe a statistically significant difference using the Mann-Whitney U test ($U = 780357.5, p < .005$) for fix-to-release delay between high and medium severity advisories, as fixes for high severity advisories (median 3 days) were quicker than medium severity (median 5 days). We do not observe any statistically significant difference between medium and low (median 3 days) severity advisories ($U = 98661.5, p > 0.20$). However, only 4.5% of the advisories in the RQ1 dataset were of low severities (possibly insufficient for a comparison).

**No statistical difference in fix-to-release delay between CVEs and non-CVEs:** We found no statistically significant difference in fix-to-release delay between CVEs and non-CVEs ($U = 5346.0, p > .04$). CVEs have a median fix-to-release delay of 4 days, while non-CVEs have a median delay of 3 days.

Table 4.5: Analysis on how many security releases have a release note and the release notes mention the security fix (RQ2)

| Ecosystem | Advisories[1] | Packages | Security Releases[2] | # of unique CWE types | Contains Release Note | Mentions Security Fix | Mentions Unrelated Changes | Mentions Breaking Change |
|---|---|---|---|---|---|---|---|---|
| Composer | 50 | 30 | 72 (98) | 18 | 63 (64.3%) | 58 (59.2%) | 50 (51.0%) | 2 (2.0%) |
| Go | 50 | 49 | 65 (66) | 25 | 54 (81.8%) | 43 (65.2%) | 43 (65.2%) | 10 (15.2%) |
| Maven | 50 | 49 | 76 (76) | 25 | 48 (63.2%) | 31 (40.8%) | 32 (42.1%) | 2 (2.6%) |
| npm | 50 | 44 | 70 (70) | 18 | 43 (61.4%) | 39 (55.7%) | 24 (34.3%) | 8 (11.4%) |
| NuGet | 50 | 28 | 56 (60) | 14 | 49 (81.7%) | 40 (66.7%) | 46 (76.7%) | 1 (1.7%) |
| pip | 50 | 45 | 61 (64) | 31 | 57 (89.1%) | 53 (82.8%) | 41 (64.1%) | 8 (12.5%) |
| RubyGems | 50 | 40 | 65 (65) | 20 | 49 (75.4%) | 43 (66.2%) | 24 (36.9%) | 1 (1.5%) |
| All | 350 | 285 | 465 (499) | 68 | 363 (72.7%) | 307 (61.5%) | 260 (52.1%) | 32 (6.4%) |

1. Advisories consist of 25 CVEs and 25 non-CVEs.
2. Numbers in parenthesis indicate distinct advisory-release pair

The analyzed packages in our study included security fixes in a new release within 4 days of the fix at the median. However, one-fourth of the releases in our data set still came at least 20 days after the corresponding security fix. Such a delay creates a window of opportunity for the observant attackers, who will know about the vulnerability when a security release is still not available to the client projects.

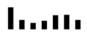
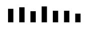
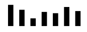
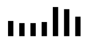
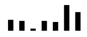
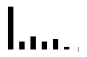
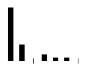
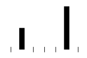
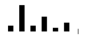
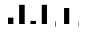
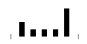
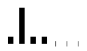
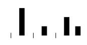
### 4.3.2 RQ2: Documentation

Table 4.5 shows how many security releases have a release note, and if the release note mentions the security fix, unrelated changes, and breaking changes. We identified eight categories of information that are documented in the release notes for security fixes and five categories of information for breaking changes. Table 4.6 lists the categories of information on the security fixes and breaking changes along with the sub-codes used during open coding, relative frequency over the seven ecosystems, and an example for each of the categories. Below, we discuss our findings for each of the sub-questions of RQ2.

#### 4.3.2.1 How many of the security releases contain a release note? What are the sources to find the release notes?

We could locate a release note for 363 (72.7%) cases in our data set. We found changelog to be the most common source (201), followed by GitHub release note (93). In 45 cases,

Table 4.6: Information categories documented in the release notes regarding security fixes and breaking changes (RQ2)

| **Category (count)** | **Explanation**[1] | **Ecosystem histogram**[2] | **Example**[3] |
|---|---|---|---|
| | | **How security fixes are documented** | |
| Security notice (198) | Explicit mention of the keywords: *security* and *vulnerability*. |  | *[SECURITY] Security release.* |
| Fix reference (181) | Link to fix commit, pull request, bug issue; commit message, pull request title; summary of fix. |  | *Remove unnecessary escape. in Regex. (#14)* |
| Vulnerability description (150) | Vulnerability type, description, severity; risk explanation; attack types. |  | *Fix yaml potential remote execution issue.* |
| Advisory reference (121) | CVE identifier; link to corresponding advisory. |  | *Backported fix for CVE-2020-6468.* |
| Affected component (79) | Vulnerable API, code components such as functions, classes. |  | *This fixes a security issue with sequelize.json() for MySQL.* |
| Affected versions (33) | List of versions affected by the vulnerability. |  | *The issue applies to Rancher versions v2.0.0-v2.0.15, v2.1.0-v2.1.10, v2.2.0-v2.2.4.* |
| Exploit (26) | Proof-of-concept; exploit instruction; exploit condition; attack vector. |  | *An attacker can set up a domain whitelistedXexample.com that will pass the whitelist filter.* |
| Affected configuration (3) | Configuration required for vulnerability to manifest. |  | *Fixed a security vulnerability for Windows users that have dcmtk installed.* |
| | | **How breaking changes are documented** | |
| Breaking change notice (22) | Explicit mention of breaking change or backward incompatibility. |  | *Breaking changes: Run onSend hooks when sending a stream.* |
| Affected API (20) | List of breaking APIs, code components; deprecated insecure API; affected code behavior. |  | *–cassandra.tls is replaced by –cassandra.tls.enabled.* |
| Action required (9) | Action required to mitigate breaking changes. |  | *Those affected must upgrade to a newer Docker version.* |
| Code change reference (9) | Link to commit, pull request inducing breaking change. |  | *GPU metrics provided by kubelet are now disabled by default (#95184).* |
| Affected configuration (7) | Dropping support; configuration change. |  | *Support for the CoreOS OS distribution has been removed.* |

1. Explanation and sub-codes of the category used during open coding.
2. Histogram of relative frequency for each category, plotted in the order: Composer, Go, Maven, npm, NuGet, pip, RubyGems.
3. Excerpt from release notes exemplifying the corresponding category of information.

we located the release note on the package homepage website. In 23 cases, we considered the tag message as the release note. In 1 case, we found a separate security notice for the release. In 9 cases, we noticed the security release listed by Snyk was incorrect, as we were able to locate the fix information in a different release.

#### 4.3.2.2 How many of the release notes mention the security fix? What information regarding the security fix is present in the release notes?

We identified the mention of security fix for the corresponding advisory in 307 cases (61.5%). If we disregard the releases without a release note, we find that the release notes document the security fix 84.5% of the time. We find that Python packages are most likely to mention the security fix in the release notes (82.8%), while Maven packages are the least likely to do so (40.8%). One possible reason for not mentioning security fixes in the release notes is that the omission is intentional, and the fix may be announced in a private channel. Another possible explanation is the security aspect of the fix was not recognized by the package maintainers at the time of the release.

For the release notes where the security fix was mentioned, we identified eight categories of information to be documented about the security fix. Table 4.6 lists the identified categories. Below, we explain the categories in the order of their frequencies. A single release note can contain multiple information categories, therefore, the total frequency of all the categories is greater than 100%.

1. **Security notice** (39.6%) refers to cases where the release note notifies the reader about a security fix through an explicit mention of the keyword ‘security’ or ‘vulnerability’. While an explicit signal can be helpful for the client projects, we observe that release notes can be long and readers may miss the signal if not properly highlighted.

2. **Fix reference** (36.2%) refers to cases where the release note either points to the fix commits or summarizes the fix. However, when the security is not explicitly mentioned, the readers may fail to understand that the fix addresses a potential security issue and not just a general bug.

3. **Vulnerability description** (30.1%) refers to cases where the release note describes the vulnerability, such as the vulnerability type and the explanation of potential risk.

4. **Advisory reference** (24.2%) refers to cases where either the CVE identifier is mentioned or an external link to a related advisory is provided. We find that the majority of the advisory references are provided in the case of CVEs (89.2%).

5. **Affected component** (15.8%) refers to the cases where the release note mentions which component of the package is affected by the vulnerability. Release notes can mention which API or function is vulnerable. Such information can be helpful for the client projects to evaluate if their projects actually use the vulnerable code of the package and prioritize fixes based on that.

6. **Affected versions** (6.6%) refers to cases where the release note states the range of versions affected by the vulnerability. Such information can be helpful for client projects who are using a much older version of a package than the latest available and evaluate if the old version they are using is also affected by the vulnerability or not.

7. **Exploit** (5.2%) refers to cases where the release note explains how the vulnerability can be exploited. Vulnerabilities with the exploit available are deemed to be of higher risk and can help the client projects to assess how to prioritize the fixes.

8. **Affected configuration** appears in only 3 release notes (0.6%) where the release notes state the configuration in which the vulnerability is manifested. This information can also help the client projects understand whether they are affected by the vulnerability or not.

While all eight categories are related to a security fix, we find 44 cases (0.9%) where the release note only lists the fix reference and does not provide any other information. While we could identify the fix reference with help from the reference URLs provided with the Snyk advisory, a typical reader is unlikely to understand the security aspect of the fixes and may ignore prioritizing the update.

#### 4.3.2.3 How many of the release notes mention changes unrelated to the security fix?

We find that in 260 cases (52.1%), unrelated changes to the security fix are mentioned in the release note (71.6% of the time when we could locate a release note). Only in 91 (18.2%) cases, the release note only mentioned the corresponding security fix and nothing else. Conversely, 48 (9.6%) times the release note mentions unrelated changes but not the security fix. We observe that the release notes typically mention general bug fixes, new features, and enhancements besides the security fix. Further, only in 1 case, we found multiple changes to be mentioned in the release notes, and all of them were security fixes.

#### 4.3.2.4 How many of the release notes mention breaking changes? What information regarding the breaking change is present in the release notes?

We find that in 32 (6.4%) cases, the security releases documented breaking changes in their release notes. However, in only 5 of these cases, the breaking change was introduced due to the security fix itself. We identified five categories of information to be documented about the breaking change in the notes for security releases, as shown in Table 4.6. Similar to 4.3.2.2, multiple categories may appear in a single release note. Below, we explain the categories in the order of their frequencies.

**Breaking change notice** (22) refers to cases where "breaking change" or "backward incompatibility" is explicitly mentioned. In 20 cases, the **affected APIs** are listed due to the breaking change. In 9 cases, the release note mentions the **actions required** by the client projects to adjust to the breaking change. **Code change reference** (9) refers to cases where the release notes refer to the commits that have caused the breaking change. Finally, in 7 of the cases, the release notes mention the **affected configuration** under which the breaking change manifests.

#### 4.3.2.5 Impact of vulnerability severity

We investigated if vulnerability severity impacts if and how the security fix will be mentioned in the release notes. For the comparison, we use Pearson's Chi-Squared test (Plackett 1983) with Bonferroni correction ($\alpha = .025$), and consider only the data points where we could locate a release note [39].

**Severity level does not impact how the security fix will be mentioned in the release notes.** We observed no statistical difference between high and medium severity advisories if the corresponding fix will be mentioned in the release notes ($X^2(1, N = 343) = 0.004, p > .95$) or if the release note will provide a security notice ($X^2(1, N = 343) = 2.718, p > .09$). We do not test low-severity advisories, as they only constitute 27 data points in our dataset.

**CVEs are more likely to have a security notice in the release notes than non-CVEs.** For the security releases where we could locate a release note, we find no statistical difference between CVEs and non-CVEs in the likelihood of the security fix being mentioned ($X^2(1, N = 363) = 3.011, p > .08$). However, we find that CVEs are more likely to come with a security notice, as categorized in this study, than non-CVEs ($X^2(1, N = 363) = 7.647, p < .006$). In 60.9% of the cases, the CVEs came with a security notice in the release notes, while the

[39]We do not measure if vulnerability severity impacts, if the security release will contain a release note or not, as publishing release notes, is more likely to be a package specific practice.

Table 4.7: Analysis of code change in a security release and code change type according to semantic versioning (RQ3)

| Ecosystem | Advisories | Packages | # of unique CWE types | Security Releases | Patch release | Major release | Unstable release | # of Files (median) | LOC change (median) |
|---|---|---|---|---|---|---|---|---|---|
| Composer | 742 | 198 | 63 | 784 | 85.3% | 0.4% | 1.8% | 11.0 | 212.0 |
| Go | 147 | 111.0 | 42.0 | 166 | 61.4% | 1.8% | 23.5% | 3.0 | 59.5 |
| Maven | 589 | 293.0 | 77.0 | 754 | 66.8% | 1.2% | 3.1% | 13.0 | 326.5 |
| npm | 690 | 486 | 79 | 734 | 54.6% | 6.9% | 19.1% | 3.0 | 58.0 |
| NuGet | 190 | 44 | 20 | 110 | 87.3% | 0.9% | 2.7% | 17.0 | 447.0 |
| pip | 467 | 231 | 75 | 425 | 53.9% | 1.4% | 21.2% | 7.0 | 121.0 |
| RubyGems | 203 | 112 | 46 | 265 | 64.5% | 3.0% | 13.6% | 3.0 | 34.0 |
| All | 3,028 | 1,475 | 155 | 3,238 | 67.1% | 2.5% | 10.7% | 6.0 | 131.0 |

non-CVEs came with a security notice only in 45.6% of the cases.

The analyzed packages in our dataset documented security fixes in the release note 61.5% of the time. However, the security implication (of the fixes) was explicitly mentioned only in 39.6% of the cases. The lack of documentation may result in client projects remaining unaware of the available security fixes.

## 4.3.3 RQ3: Code Change Characteristics

In this section, we present our findings for RQ3:

#### 4.3.3.1 Quantitative findings

Table 4.7 shows the median count of files and LOC change in open source security releases. The table also shows how many of the security releases were formatted as patch, major, or unstable releases as per SemVer formatting.

We find that open source security releases contain a median of 131 LOC changes, modifying a median of 6 source code files. We find RubyGems security releases to have the smallest median code change, while NuGet has the highest. However, as explained in Section 4.2.4.2, we applied a heuristic-based approach to locate package-specific source code within the repository for NuGet packages (unlike the other ecosystems where we download source code directly from the package registries), and therefore, there can be cases of over-approximation as the repository may contain non-package files as well.

In our dataset, 67.1% of the releases were patch releases, indicating the releases only contain bug fixes. Conversely, 13.2% of the releases indicated backward incompatibility, as 2.5% were major releases and 10.7% were unstable releases. We find that Go, npm, pip, and RubyGems packages are more likely to have security vulnerabilities during the initial development phases (version below 1.0.0) and release fixes in the subsequent unstable releases. For the rest of the dataset, 14.7% were formatted as minor releases indicating the introduction of new functionalities, while 5% were pre-release. We observe that the pre-releases typically come before a major or a minor release. Pre-release to a major release can also indicate breaking changes are being introduced alongside a security fix, depending on when the vulnerability was introduced.

#### 4.3.3.2 Understanding large code changes

To investigate if security releases bundle only security fixes or not, we look at 10 randomly-sampled releases from each ecosystem that had more than 131 LOC changes (the median across all ecosystems). However, Go and RubyGems only had 3 releases each with LOC changes over the overall median. Therefore, we manually investigate 56 data points in total, 3 from Go and RubyGems each, and 10 each from the rest of the five ecosystems. Specifically, we look at the commits made between the two releases on a package's source repository and try to locate when the fix commit for the corresponding advisory was made through available metadata.

In 21 cases, we were unable to identify the fix commit. However, (in these 21 cases) the changes between the two releases incorporated many commits where the commit messages indicate various types of changes, and the releases appeared to follow the regular release cycle of the respective package. In 16 cases, the release came at least 7 days after the security fix commit with newer code changes, and the releases appeared to follow the regular release cycle. In 16 cases, we found the release came soon after the security fix (within 7 days). However, the release included all the changes made to the package since its last release. In 1 case, the security fix itself was large in code change size. In 1 case, the release listed by Snyk (and NVD) was wrong, and the security fix was already included in an earlier release. In another case, a release was made immediately after the fix in the source repository (by tagging a new version, along with earlier changes) but was not published on the npm registry.

Li and Paxon found security fixes to contain 7 LOC changes at median (Li and Paxson 2017). However, we find the security releases in our data set contain 134 LOC changes at the median. The explanation behind this difference is that the packages bundle unrelated functional changes alongside the security fixes in these releases. Our explanation is supported by prior work (Chinthanet et al. 2021) and RQ2 findings.

Further, we observe that even if a release is published immediately after the security fix, the release is still likely to contain all the changes made to the corresponding repository branch since the last release on that branch. Therefore, the code change size of a security release may be determined by the typical release cycle and the development activity level at the time for a given package. Our findings suggest that open-source packages do not follow the recommendation (from the prior work (Pashchenko et al. 2020c)) of dedicating separate releases for security fixes.

#### 4.3.3.3 Impact of vulnerability severity

Similar to RQ1 (the methodology explained in Section 4.2.2.4), we investigated the impact of vulnerability severity on code change size in security releases. However, a single release can contain multiple security fixes. We assign a severity level to a security release according to the highest severity vulnerability it fixes. Similarly, when measuring the difference between CVEs and non-CVEs, if a security release contains fixes for any CVE, we consider it as a CVE fix release, and otherwise non-CVE fix release.

**No statistical difference observed for code change size of security releases between different severity levels of corresponding advisories:** Using the Mann-Whitney U test, we observed no statistical difference between security releases for high (136 LOC, 6 files changed at the median) and medium (127 LOC, 6 files changed at the median) severity advisories and medium and low (148 LOC, 8 files changed at the median) severity advisories ($p > 0.07$ in all the cases).

**No statistical difference in code change size of security releases for CVEs and non-CVEs:** We also observe no statistical difference (using Mann-Whitney U test) in code change size between CVE and non-CVE fix releases: (i) 6 files changed at the median for both the cases, (i) 127 LOC changed in CVE releases and 140.5 LOC changed in non-CVE releases at the median ($p > 0.22$ for both the cases).

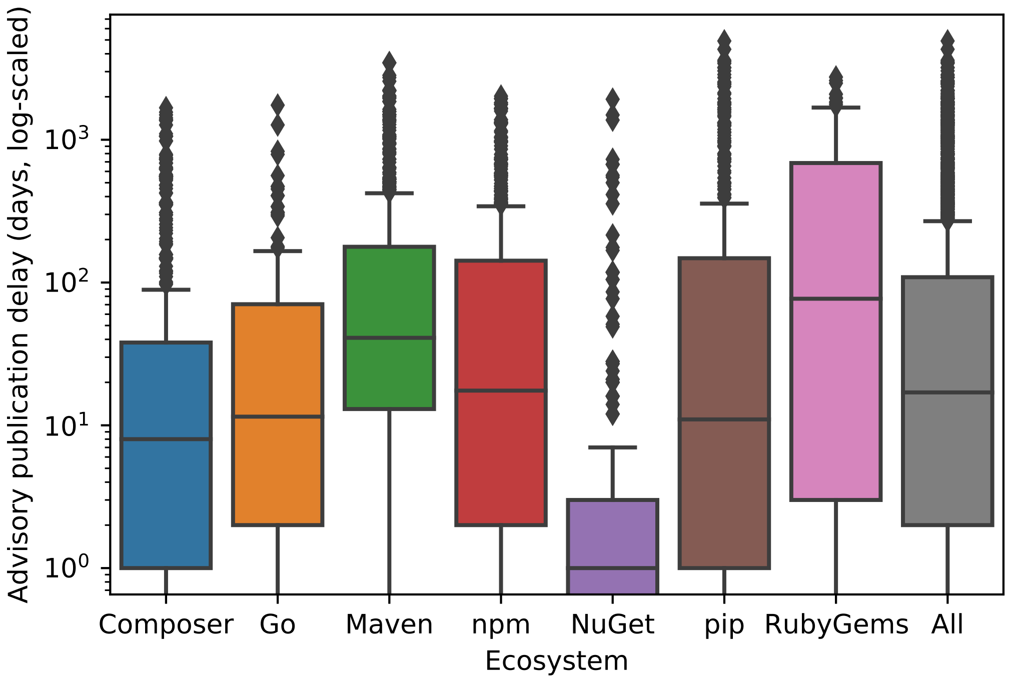


Figure 4.3: Box plot for the delay in advisory publication since the security release

The analyzed security releases in our data set contain 131 lines of code (LOC) changes and modify 6 source code files at the median. We observe that packages typically bundle unrelated functional changes alongside the security fix in these releases, violating the recommendation from prior work of dedicating separate releases for security fixes.

### 4.3.4 RQ4: Advisory publication delay

In this section, we present our quantitative findings followed by an analysis of RQ2 data points to understand the relationship between documentation of security fixes and advisory publication.

#### 4.3.4.1 Quantitative findings

Table 4.8 shows the number of advisories that were published after a security release and the median advisory publication delay for these advisories. Overall, 2,915 advisories (79.8%)

in our data set were published after the corresponding security release. For these advisories, we find the median delay between security release and advisory publication across all the ecosystems to be 17 days. Figure 4.3 shows a box plot of the delays. Further, Table 4.9 shows a breakdown of advisory publication delays over CVEs and non-CVEs.

We find that RubyGems has a larger median delay in the advisory publication than the other six ecosystems. Further, 29.8% of these RubyGems advisories are not reported on NVD as well (non-CVEs). Security researchers and SCA toolmakers may scan open source repositories to identify unreported security fixes (Zhou and Sharma 2017; Catabi-Kalman 2020), which may be a possible reason why vulnerabilities are reported long after a fix was already released. The non-CVEs in RubyGems have a median advisory publication delay of 916.5 days, while CVEs have a median delay of 8 days, supporting our explanation.

#### 4.3.4.2 Impact of vulnerability severity

Overall, 72.6% of the 2,915 advisories (that were published on or after the security release) are CVEs. While CVEs have a median advisory publication delay of 13 days, non-CVEs have a median delay of 26 days. We found the difference to be statistically significant through the Mann-Whitney U test ($U = 706054.0, p < .0001$).

We find no statistically significant advisory publication delay between high (median 9 days) and medium (median 22 days) severity advisories ($U = 83559.0, p = 0.4$). We only have 132 low-severity advisories in our RQ4 dataset, with a median delay of 12 days, and no statistically significant difference with the high or medium-severity advisories.

While advisories published before a security release cause a vulnerability to have widespread circulation without having a fix available, the delay in advisory publication after the security release suggests that: (i) there may be unknown security risks in using outdated dependencies (Decan et al. 2018a), as some security fixes may not have been reported (yet) to an advisory database; and (ii) open source packages need to employ standardized practices when announcing security releases, such as documenting in the release notes.

#### 4.3.4.3 Release note vs advisory publication

In RQ2, we qualitatively analyzed release notes of the security releases. From that data, we investigate the relation between the delay in advisory publication and if/how security fixes were documented in the release notes. We have 362 distinct advisory-security release pairs for whom (i) we manually analyzed the release notes (in RQ2); (ii) we have the publication

Table 4.8: Analysis of time lag between security release and advisory publication (either on NVD or Snyk)

| Ecosystem | Advisories | Advisories published on or after security release date | Median Advisory publication delay* (days) |
|---|---|---|---|
| Composer | 751 | 675 (89.9%) | 8.0 |
| Go | 211 | 160 (75.8%) | 11.5 |
| Maven | 937 | 715 (76.3%) | 41.0 |
| npm | 735 | 592 (80.5%) | 17.5 |
| NuGet | 241 | 208 (86.3%) | 1.0 |
| pip | 569 | 424 (74.5%) | 11.0 |
| RubyGems | 211 | 141 (66.8%) | 77.0 |
| All | 3,655 | 2,915 (79.8%) | 17.0 |

* For the advisories published after security release

Table 4.9: Comparison of advisory publication delay (median days) between CVEs and Non-CVEs that were published after the security release

| Ecosystem | CVEs | Publication delay for CVEs | Non-CVEs | Publication delay for Non-CVEs |
|---|---|---|---|---|
| Composer | 449 | 9.0 | 226 | 4.0 |
| Go | 130 | 11.5 | 30 | 12.5 |
| Maven | 620 | 35.0 | 95 | 99.0 |
| npm | 341 | 9.0 | 251 | 41.0 |
| NuGet | 179 | 1.0 | 29 | 3.0 |
| pip | 298 | 3.0 | 126 | 155.0 |
| RubyGems | 99 | 8.0 | 42 | 916.5 |
| All | 2,116 | 13.0 | 799 | 26.0 |

date of the release; and (iii) the advisory was published on/after the security release. We make the following observations based on these 362 data points:

**The security fixes that were not mentioned in the release notes have a median advisory publication delay of 161 days:** The 89 security releases where we could not locate a release note have a median advisory publication delay of 21 days. However, the 44 releases where we could locate a release note, but the note does not mention the security fix, have a median advisory publication delay of 161 days. A possible explanation is that the security aspect of the fixes was not known at the time, and the vulnerability was only recognized months later.

**Notification to the client projects may get delayed even when the security fix is explicitly mentioned in the release notes:** The 229 security releases where the release note mentions the security fix have a median advisory publication delay of 9 days. However, in 25% of these cases, the advisory was published at least 54 days after the security release, potentially creating a lag between the time of a security release and the time the client projects are notified. Further, 156 of these releases that explicitly mentioned security in the release notes have a median advisory delay of 8 days. However, in 39 of these cases (25%), the advisory was published at least 30 days after the security release.

We find that there can be a median time lag of 17 days between a security release and an advisory publication, resulting in client projects receiving delayed notification of the corresponding vulnerabilities. Such a delay further expands the window of opportunity for the attackers discussed in RQ1, as client projects may still remain unaware of the known vulnerabilities and the available security releases.

## 4.4 Discussion

In this section, we discuss (i) the possible cases of delay in the downstream propagation of security fixes; (ii) the possible regression risk for the client projects in adopting the security releases; and (ii) a comparison of security release practices between CVEs and non-CVEs. Afterward, we discuss four avenues for future work.

### 4.4.1 Delay in fix propagation

We have found the time lag between a security fix and the release that includes the fix to be four days at the median, which may be deemed reasonable given the developmental activities required before a release, such as maintainers' coordination, testing, etc. We discuss how security fixes may be kept private until release in Section 4.4.5 to reduce the risk of fix-to-release delay. However, a security release itself does not ensure the security of the client projects. Client projects need to be informed of the available security release and upgrade to the new version to adopt the security fix. We find that the release notes of the security releases lack documentation 38.5% of the time. Further, there can be a large delay in advisory publication as well. Such delays expand a vulnerability's exposure and increase the risk of security breaches, such as the Equifax data breach (Fruhlinger 2020).

While SCA tools like Dependabot are the primary means for the client projects for notification on upstream security fixes, we found in Chapter 3 that SCA tools suffer from both false positives and false negatives. In this study, we have found that there can be a time lag between a security release and the corresponding advisory publication, causing SCA tools like Snyk to send delayed notifications. Furthermore, we have observed that the security advisories may not contain information on the available security release as well. In our collected data from Snyk, 1,413 security advisories (20.3% of 6,956 advisories) did not have a security release listed. When manually examining 25 randomly-sampled advisories without a security release, we found that in 13 of these cases, a security release is actually available, but Snyk lists all versions of the affected package as vulnerable [40]. Inaccurate data, therefore, can result in SCA tools sending false vulnerability alerts to client projects.

Overall, reporting vulnerability data to advisory databases and subsequent updates of the data with information such as security releases primarily involves a manual reporting process. Due to this non-automated process, security releases may remain unreported. As a result, client projects, that are not using the latest versions of their dependencies, may remain unaware of vulnerabilities even when using an SCA tool. Conversely, clients can keep receiving false alerts even when they are using a fixed version. In Section 4.4.5, we discuss how employing standardized practices while announcing security releases can help us automate the notification process to client projects.

**User patch adoption period:** While some vulnerability disclosure policy includes an intentional delay between the security release and the advisory publication to facilitate user

[40]For the rest, in 7 cases, the affected packages are unmaintained. In 3 cases, a fix was made but was not released in a new version yet. In 2 cases, we were unable to identify if a fixing version exists or not.

patch adoption (Willis 2021), how well the policy fits with the open source development model needs to be studied. In such a policy, the package maintainer should be careful to not discuss the security fixes in either release notes, bug trackers, or commit messages. Otherwise, the security fix will not remain secret as intended anyway. Further, how to notify client projects of open source packages across the world and ensure the vulnerability remains secret should also be studied. In practice, we have observed that Snyk and GitHub security advisory databases do not incorporate such user adoption period and publish advisory soon after a vulnerability is reported on their platforms.

### 4.4.2 Regression risk when upgrading to a security release

Dependency updates may introduce changes that cause the client projects' code to stop working or to have deteriorated performance. Prior research has found that such regression risk is a major reason developers do not want to update their dependencies, even when they are aware of existing vulnerabilities (Pashchenko et al. 2020c; Kula et al. 2018). The analysis in RQ2 and RQ3 is partially motivated by an objective to estimate the regression risk carried by security releases. Based on our findings, we present our observations:

**Packages under active development do not (typically) dedicate a separate release for security fixes:** Based on developer interviews, Paschenko et al. (Pashchenko et al. 2020c) recommend packages to dedicate separate releases for security fixes (i.e. not including unrelated functional changes in the same release) to help fast adoption by the client projects, as developers may view functional changes to be a potential avenue for introducing a regression. Chinthanet et al. (Chinthanet et al. 2021) have analyzed 231 security releases from the npm ecosystem and found that the median release contains 219 lines of code (LOC) change [41]. However, upon manually analyzing the commits, they identified that only 10 LOC changes (at a median) in the releases were related to the security fix.

In our work, we found that the median open source security release contains 134 LOC changes. In our manual analysis (in RQ2 and RQ3), we found that the releases may contain code changes unrelated to the security fix. We observed that some packages follow their typical release cycles when releasing security fixes, resulting in large code changes in the release. Even when a package issues a new release immediately after a security fix, the release may still come bundled with all the code changes made since the last release. Overall, our

[41] In our study, we found that the median npm security release contains only 58 LOC change. (Chinthanet et al. 2021) measured code changes by comparing releases on GitHub repositories which may cause an over-approximation in the measurement as repositories often contain multiple packages or test files that are not included in the published package.

analysis suggests that open source packages do not follow the recommendation of issuing separate releases for security fixes.

**Package maintainers may not (always) strictly follow SemVer formatting (Preston-Werner 2021) when making breaking changes:** SemVer versioning is a guideline for package maintainers to indicate the code change type in a certain release. However, there is no technical method to measure if the maintainers are accurately following the guideline. Prior work has found that the introduction of breaking changes in non-major releases (in violation of SemVer policy) is prevalent among Maven packages (Raemaekers et al. 2014). In our analysis of release notes, we also find non-major releases to mention breaking changes. Specifically, out of 32 security releases that mention breaking changes in the release notes, 6 were patch releases and 12 were minor releases. Including breaking changes in non-major stable releases validates developers' fear of regression when updating dependencies.

### 4.4.3 CVEs vs non-CVEs

In Chapter 3, we found six of the nine studied tools to have reported unique non-CVEs that were not reported by the other tools in the study. However, while the CVE data is widely accepted within the security community, and well studied in the literature (Frei et al. 2006; Shahzad et al. 2012; Nakajima et al. 2019; Shahzad et al. 2019; Ruohonen et al. 2020; Li and Paxson 2017), vulnerabilities without a CVE identifier are still under-studied, although they are not uncommon as we have seen in Chapter 3. Approximately one-fourth of the advisories in our dataset are non-CVEs which provides us with a unique opportunity to compare CVEs and non-CVEs.

In the measurement for RQ1, RQ2, and RQ3, we do not find any difference between CVEs and non-CVEs besides our finding that the CVEs are more likely to contain an explicit security mention in the release notes than the non-CVEs. Further, we find that the non-CVEs have a higher delay in being published on advisory databases than the CVEs. As noted before, security researchers and SCA toolmakers often scan open source packages to discover (previously) unreported security fixes, which may explain the higher advisory publication delay.

However, we do not find any evidence in our data that CVEs and non-CVEs get treated differently by the package maintainers, indicating that both of them carry similar security risks. A possible explanation behind some vulnerabilities not having a CVE identifier is that the package maintainers may have viewed the CVE assignment process as an extra workload (Edge 2017). Therefore, a standardized process to keep track of security releases

Table 4.10: RQ1, RQ3, RQ4 measurements for the top 5 CWE types in our dataset

| CWE type | fix-to-release delay | LOC change in release | Advisory publication delay |
|---|---:|---:|---:|
| CWE-79 | 4 | 222.0 | 28 |
| CWE-400 | 3 | 88.0 | 4 |
| CWE-94 | 3 | 188.0 | 7 |
| CWE-200 | 3 | 251.0 | 27 |
| CWE-284 | 2 | 225.5 | 12 |

in the open source packages should be practiced. Further, vulnerability data should also be open source (instead of commercial databases) and be easily queryable by client projects, e.g., through package managers, to prevent any reliance on commercial tools.

### 4.4.4 The impact of CWE types

Table 4.2 shows that our study data set contains vulnerabilities across 172 CWE types. The most frequent CWE types are CWE-79 (Cross-site scripting) with 1,214 advisories, CWE-400 (Uncontrolled Resource Consumption) with 653 advisories, CWE-94 (Code Injection) with 633 advisories, CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor) with 518 advisories, and CWE-284 (Improper Access Control) with 226 advisories. These 5 CWE types comprise 69.3% of the advisories in our data set.

When investigating the RQs of this study at the granularity level of individual CWEs, we find similar results for the top CWE types (in our data set) as the overall findings, as shown in Table 4.10 (we do not show findings from RQ2 where we worked only with a subset of the data). However, note that our methodology may not be suitable to draw any conclusion on individual CWE types, as a particular ecosystem may have more advisories of a certain CWE type than the others. Further, only 14 CWE types (8.2%) contain more than a hundred advisories in our dataset.

### 4.4.5 Recommendations

In this section, we discuss our recommendations and potential avenues for future research:

#### 4.4.5.1 Private forks for security fixes

While we find open source packages to be typically fast in bundling a security fix in a new release, one-fourth of the releases in our dataset still came at least 20 days after the corresponding security fix. While there may be valid reasons why packages need time before the next release, e.g., testing, contributor coordination, etc., the commit messages and issue discussion in open source make security fixes easy to spot as we have observed in our manual analysis.

Therefore, a solution to mitigate the security risk of fix-to-release delay is to keep the security fixes private until the next release. GitHub offers a feature where open-source package maintainers can create a private fork to collaborate on the fix [42]. However, once the fix is finalized and merged into the main codebase, the fix, and the associated commit message become public. Future work can look at the implementation possibility of a private fork for the security fixes that remain private until the next release of the project. Currently, GitLab follows a similar approach to avoid the leak of security fixes in their public repositories (Speicher 2021).

#### 4.4.5.2 Standardized practice for announcing security release

When a package issues a new security release, the client projects should get notified immediately. However, manually reporting vulnerability data to advisory databases may create a lag in the process, as shown in RQ4. Further, many security releases may remain unreported. We recommend open source packages follow a standard process in tagging a version as a security release, where the process would also help automate subsequent notifications to the client projects.

A version may be tagged as a security release in multiple possible ways, e.g., (i) add metadata indicating security release in the package manifest files where the version number gets updated; (ii) add a security release label in the release note, or (iii) tag a version commit with standard headers such as *[Security Release]*. A standardized process would make package managers identify security releases automatically and notify the client projects during the clients' next developmental builds. Package managers can also automatically update advisory databases. SCA tools, like Dependabot, can then notify projects without recent activity as well.

[42]`https://docs.github.com/en/code-security/security-advisories/collaborating-in-a-temporary-private-fork-to-resolve-a-security-vulnerability`

#### 4.4.5.3 Security patch for older versions

We have discussed the code change size in security releases and the possible regression risk these releases may carry. However, the analysis may be relevant only for client projects who are using the latest version of a package in a specific release branch. Projects with outdated dependencies – that is – using a version of the package older than the latest available may face greater migration effort in upgrading to a security release (Kula et al. 2018). Therefore, popular packages may release a security patch, at least for critical vulnerabilities, that can be applied to the past versions of the package.

For example, if a package has the latest version at 3.2.14, and the security release comes at 3.2.15, the package can also release a security patch that can be applied to any version starting from 3.0.0 (e.g., from 3.0.0 to 3.0.0.1 indicating a security patch adoption). This way, client projects can immediately adopt a security fix in case of regression risks. Snyk, an SCA tool, provides such security patches for selected vulnerabilities for Node.js (JavaScript) projects [43]. Similarly, node developers can patch the dependencies locally using tools like *patch-package* [44]. Future research can look at how to automate such a process – that is – (i) identifying the fix commits of a vulnerability; (ii) converting the fix commits into a stand-alone patch; (ii) identifying all the older releases where the patch can be automatically applied without any regression; and (iii) applying the patch in the client project and keep a record of the fix.

#### 4.4.5.4 Security metrics for dependency selection

Ecosystem administrators can provide automatically measured security metrics with packages to aid security-critical projects in dependency selection. Prior work has found SCA tools to provide a security rating for packages over simple metrics such as vulnerabilities discovered in the past. The Security Scorecards project [45] provides security health metrics for selected packages, such as if the package uses static analysis testing, code review, signed releases, etc.

How packages dealt with security releases in the past can be another metric that signals the reliability of a package in case a vulnerability is found. Specifically, the metrics in RQ1-4 can be utilized in this regard. For example, how quickly the fix was released, if the release note mentioned the security fix, and if an advisory was immediately published after the

[43]https://docs.snyk.io/fixing-and-prioritizing-issues/starting-to-fix-vulnerabilities/snyk-patches-to-fix-vulnerabilities

[44]https://github.com/ds300/patch-package

[45]https://github.com/ossf/scorecard

release. Similarly, future research can look at (i) if providing security metrics nudge packages to employ better practices; and (ii) if such metrics help the client projects in dependency selection and maintain their own projects' overall security health.

Further, SCA tools may give a package a poor rating if the package had known vulnerabilities in the past. Such a penalty may discourage packages to report vulnerabilities in public databases. We argue that if a package follows best practices in handling security fixes, the package should be rewarded rather than penalized in its security ratings.

## 4.5 Threats to validity

**Data accuracy:** One threat to the validity of our findings is the accuracy of our data set. We leverage the Snyk advisory database that is specifically curated for vulnerabilities in open source packages. Still, Masecci suggests providing a manual validation for a data set that is constructed through automated approaches (Massacci 2022). Following the suggestion, we have provided a data accuracy check for both RQ1 and RQ3 (RQ2 analysis is manual, while RQ4 depends on the advisory publication date). Further, following prior work (Nguyen et al. 2016), we have provided Agresti and Coull's score confidence interval for our data accuracy checks, which measures the accuracy to be between 92.5% to 100% with a 95% confidence level. Nonetheless, we have observed that the outlier data points are more likely to be inaccurate. Therefore, we believe the threat to our overall findings and conclusions is minimal.

For the analysis of RQ1, advisory references may point to fix commits applied to different release branches. However, we considered the latest of these commit dates as the fix commit date for all the branches, which may not be accurate in all cases. We believe the threat is minimal, as we find security releases in all the branches are likely to come on the same day, 51.8% of the time. Similarly, in measuring advisory publication delay, we only consider two databases (Snyk and NVD) while there exist other popular databases as well, such as GitHub and npm security advisory databases. Nonetheless, our analysis shows the risk of delayed notification when using Snyk as an SCA tool.

**Generalizability threat:** Another threat to our study is whether the analyzed data generalize to all open source packages and all types of vulnerabilities (and their security releases). We study seven popular package ecosystems, which provide a broad picture of the open-source packages in general. Further, a database may be biased toward certain types of vulnerabilities and software packages, depending on the vulnerability reporting and data collection process (Li and Paxson 2017; Christey and Martin 2013). We address this threat by leveraging the Snyk database that is specifically curated for vulnerabilities in the open source packages and includes non-CVEs and CVEs from NVD. However, we worked only with the publicly available data from the Snyk database, which may introduce unknown biases.

Further, we had to exclude 300 packages from the collected data from Snyk because we could not identify their repositories. Similarly, the analysis for RQ1 incorporates vulnerabilities only where the fix commit is known and may be biased. Further, we only worked with a randomly-sampled subset of 350 advisories to answer RQ2, which may also suffer from the threat to generalizability. We report the count of distinct packages and CWE types while reporting our findings for each RQs to indicate the generalizability of our findings.

**Subjectivity in the qualitative analysis for RQ2:** We adopt a manual qualitative analysis to answer RQ2, which involves subjectivity. To ensure the reliability of the qualitative analysis, two of the authors performed the analysis. In cases where the two authors disagreed in their analysis, the conflict was resolved through discussion.

**The security risks posed by the studied vulnerabilities:** Not all dependency vulnerabilities impact the client projects or have a known exploit. Therefore, the type of security risk posed by a vulnerability may impact how package maintainers would handle the security release. Further, the impact of a dependency vulnerability depends on how the client projects use the dependency package (e.g., does the client project use the vulnerable functionality?) and will vary from case to case (Plate et al. 2015). However, note that all the vulnerabilities in our data set were validated by Snyk (Catabi-Kalman 2020), and the client projects (who use Snyk) get notified of these vulnerabilities regardless of their perceived security risk. The only reliable metric we have for the vulnerabilities is their severity rating. For all the RQs, we provide analysis broken down across severity analysis. We also provide a short discussion on the CWE types of the studied vulnerabilities in Section 4.4.4.

## 4.6 Conclusion

In this chapter, we study fix-to-release delay, documentation, code change characteristics, and advisory publication delay for past security releases across seven open source package ecosystems, namely Composer, Go, Maven, npm, NuGet, pip, and RubyGems. We find that the analyzed packages are typically fast in releasing security fixes, as the median release comes within 4 days of the corresponding security fix. However, one-fourth of the releases still have a delay of at least 20 days. While an advisory may be published only after the security release, the commit messages and issue discussion within the code repository makes it possible for potential attackers to discover the vulnerability and gain a window of opportunity to plan attacks on the client projects. Further, even after the security release, the client projects may remain unaware of the security fix, as we find a median delay of 17 days before the publication of an advisory on Snyk and NVD.

We analyzed the release notes of the security release to investigate if and how the release note documents the security fixes. We found that 38.5% of the security releases either do not have a release note, or the release note does not mention the security fix. Conversely, we find that even when the release note explicitly mentions the security fixes, 25% of the security releases still take at least 30 days before a corresponding advisory is published. Such a time lag can result in delayed notifications to the client projects through an SCA tool, and can further increase the window of opportunity for the attackers. To address these security threats, we recommend package maintainers work in private forks for security fixes and follow a standardized practice when announcing security releases so that the client projects are notified automatically and immediately.

Lastly, we find that security releases contain 131 lines of code changes and modify 6 source code files at the median. We observe that the security releases may also come with unrelated functional changes and even breaking changes, as packages typically do not follow the recommendation of dedicating a separate release only for the security fixes. Further, the potential migration effort may be higher for client projects that are using an older version than the latest available of a dependency package. In this regard, we recommend packages release security patches, at least for critical vulnerabilities, that can be applied to older versions of the package.

To conclude, the reliance on open source packages in modern software development makes the client projects' security health to be directly correlated with the security health of their dependency packages (Cox 2019). Therefore, projects should select dependencies keeping a package's security practices in mind (Pashchenko et al. 2020c). Ecosystem administrators may provide metrics that signal the security health of a package to aid the client projects in this regard. One such signal can be how security releases in the past were handled by the packages. Such a signal will not only help projects in dependency selection, but it may also nudge the package maintainers in following better developmental practices.

CHAPTER

5

# CODE REVIEW COVERAGE IN DEPENDENCY UPDATES

## 5.1 Motivation & Research Questions

The use of open source packages has opened up new attack vectors, as vulnerable and even malicious code can sneak into software through these third-party dependencies (Ohm et al. 2020). Further, practitioners are recommended to keep dependencies up to date with the latest version (iba 2019), resulting in developers pulling in new code through frequent updates, often automatically and without a security review (Mirhosseini and Parnin 2017).

Recent times have seen popular open source packages be compromised and push malicious updates, such as the attack through *ua-parser-js*, an npm package used by major software corporations (Flowers 2021). Attacks through dependencies, such as the preceding example, are categorized as “supply chain attacks” by security practitioners (Deming 2020). While the existence of malicious packages in different ecosystems is well-known (Ohm et al. 2020), popular packages from reliable sources can also be compromised, such as through: 1. hijacking of a maintainer’s account; 2. a maintainer going rogue; 3. account handover through social engineering; and 4. build system compromise. (Ohm et al. 2020; Zahan et al. 2021)

Therefore, practitioners are now recommended to review dependency updates before merging them into the codebase (Yang et al. 2021), as the responsibility of security lies on the consumer when using free open-source code (Washburn 2017). However, manually reviewing all the code changes in each update may not be a practical solution, as projects may have hundreds of direct and transitive dependencies (Synopsys 2021). Further, actively maintained packages get frequent updates, over-burdening any project that would employ such strict measures. Therefore, we propose that *dependency updates should go through automated security and quality checks before being merged*, which can act as the first line of defense and can aid developers in securely accepting dependency updates.

Recently, the software industry has proposed multiple frameworks to define the compliance standards for using open source packages, such as the "Supply chain Levels for Software Artifacts (SLSA)" [1] framework. Specifically, SLSA provides a checklist of standards and controls to prevent tampering, improve integrity, and secure packages and infrastructure. Among others, SLSA requires a dependency package to employ a two-person code review. However, while projects like the "Security Scorecards" [2] exist to aid in dependency selection, little research has been done yet on employing automated checks during each subsequent update (Vu et al. 2021). The goal of this study is to aid developers in securely accepting dependency updates by measuring if the code changes in an update have passed through a code review process.

The check for code review ensures that each line of code change in the update can be traced back to at least two persons. Such a requirement can guard against the threat scenarios where 1. only one maintainer's account has been hijacked by the attacker(s); a single maintainer has gone rogue; or 2. attackers have compromised the publishing infrastructure to inject (unowned) malicious code. Further, prior research has shown that code review helps ensure higher code quality (McIntosh et al. 2016) and can prevent the introduction of new security vulnerabilities (Bosu et al. 2014). Ladisa et al (Ladisa et al. 2022) have developed an attack taxonomy for open source supply chain attacks, where they have mentioned *code review* as a safeguard against the attack vector *inject into sources of legitimate package.*

However, the feasibility of employing an automated check for the code review requirement during dependency updates has not been studied yet. Packages get bundled and

[1] `https://slsa.dev/`

[2] `https://github.com/ossf/scorecard`

distributed in different ways and may use different code hosting platforms and code review tooling. Such differences in the maintenance and the distribution of open source packages may create challenges in reliably auditing the changes in each new update. Further, no empirical study exists on the code review practices among top open source packages to understand the practicality of employing this SLSA requirement currently in practice. This study aims to address these gaps in the research of securely using open source dependencies.

We implement Depdive which measures the ***code review coverage (CRC)*** (McIntosh et al. 2014) in a dependency update. We define *CRC* as the proportion of the code changes in an update that have gone through a code review process. Depdive works for four package registries, namely Crates.io for Rust, npm for JavaScript, PyPI for Python, and RubyGems for Ruby. We choose these registries because they follow a similar package distribution model, where the maintainers upload the package code to the registry. We further scope our implementation to GitHub repositories as we leverage the platform to determine if certain code changes have been reviewed or not.

In summary, Depdive maps code changes between two versions of a package uploaded in the registry to the corresponding commits in the package's source repository that made these changes and identifies if there was a reviewer for the mapped commits through four GitHub-based checks. Along the process, Depdive also identifies the files and lines of code that cannot be mapped from the registry to the source repository, which we refer to as ***phantom artifacts***, following the definition in prior work (Vu et al. 2021). While one approach can be to consider the phantom artifacts as non-reviewed in the denominator of the *CRC* measurement, these artifacts may exist in the form of binaries or programmatically-generated files, as will be shown in this study. Therefore, they will require a provenance tracking mechanism to audit if changes in them were reviewed or not. With Depdive, we first filter out the phantom artifacts and output them separately from the *CRC* measurement.

For an empirical evaluation, we run Depdive over the latest ten releases of the most downloaded 1000 packages in Crates.io, npm, PyPI, and RubyGems. Based on Depdive's output, we answer the following two research questions:

> **RQ1:** To what extent do phantom artifacts exist in the updates of the most downloaded packages?
>
> **RQ2:** Excluding phantom artifacts, what is the code review coverage (*CRC*) in the updates of the most downloaded packages?

Besides the answer to the above two research questions, the contributions of our work include a working tool, Depdive, that outputs details on the phantom artifacts and the code review data for a dependency update. The code and data for this study are anonymously available at `https://tinyurl.com/depdive`.

## 5.2 Selected Package Registries

Package registries are package hosting services to store and distribute packages. In this chapter, we work with Crates.io, npm, PyPI, and RubyGems registry, the primary package registries for Rust, JavaScript, Python, and Ruby language, respectively. Crates.io and RubyGems refer to packages as *crates* and *gems*, respectively, while npm and PyPI simply call them *packages*. These registries follow a similar package distribution model where developers can upload their package source code to the registry alongside any required artifacts such as data files, pre-compiled binaries, etc. in a registry-specific distribution bundle (e.g., *wheel* for PyPI). The client software can then download and install packages from these registries. It is expected that the same source code in the package's development repository, e.g., the code hosted on GitHub, is distributed via the registry. However, the registry contains its own copy of the package code, which may not be identical to the one in the repository.

## 5.3 Depdive - methodology steps

In this section, we describe the implementation of the update audit tool, Depdive. Figure 5.1 shows a high-level workflow of the tool.

Depdive takes four arguments as input: 1. registry name; 2. package name; 3. current version; and 4. update version. Depdive works with four package registries, namely Crates.io, npm, PyPI, and RubyGems. The input can be an update from any version to another version available in the registry. Depdive begins the analysis by identifying the source code repository of the given package from metadata available on the registry. We scope Depdive's implementation to GitHub repositories, as we rely on data available through the GitHub platform to determine if a commit was code-reviewed, as will be explained in Section 5.3.5.

Depdive then collects package code both from the registry and the repository and compares them to identify the phantom artifacts. Phantom artifacts in the registry do not map to any corresponding artifact in the repository, and therefore, cannot be audited

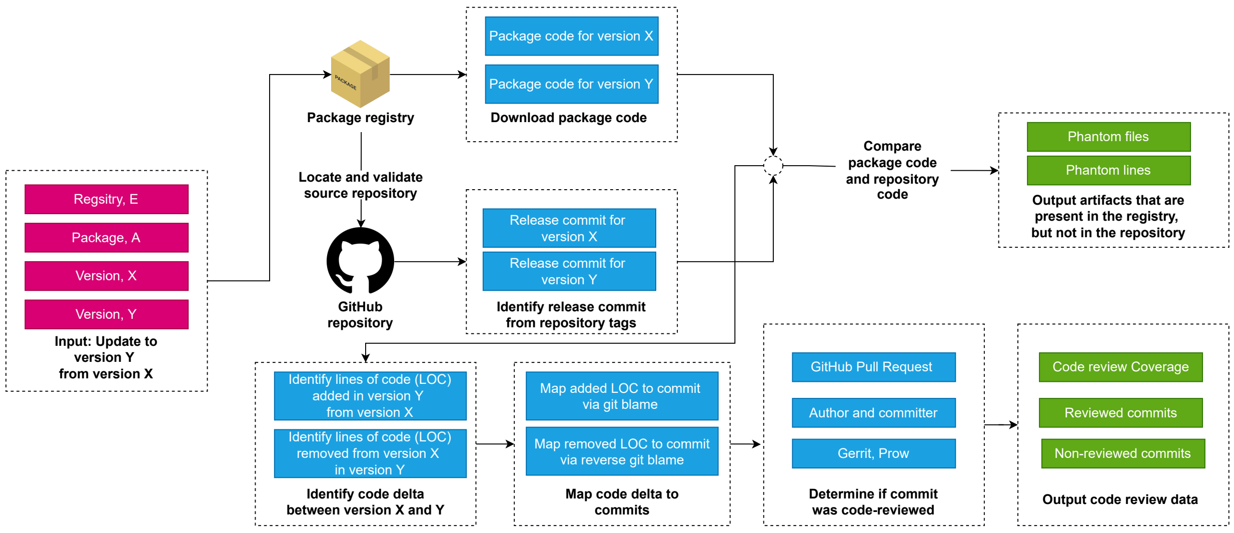

Regsitry, E
Package, A
Version, X
Version, Y
Input: Update to version Y from version X
Package registry
Locate and validate source repository
GitHub repository
Package code for version X
Package code for version Y
Download package code
Release commit for version X
Release commit for version Y
Identify release commit from repository tags
Compare package code and repository code
Phantom files
Phantom lines
Output artifacts that are present in the registry, but not in the repository
Identify lines of code (LOC) added in version Y from version X
Identify lines of code (LOC) removed from version X in version Y
Identify code delta between version X and Y
Map added LOC to commit via git blame
Map removed LOC to commit via reverse git blame
Map code delta to commits
GitHub Pull Request
Author and committer
Gerrit, Prow
Determine if commit was code-reviewed
Code review Coverage
Reviewed commits
Non-reviewed commits
Output code review data

Figure 5.1: Depdive workflow

for code review without tracking the provenance of these artifacts first. Hence, Depdive outputs data on phantom artifacts separately from code review analysis. Algorithm 1 lists pseudocode for determining phantom files and lines in an update.

Afterward, Depdive maps each line of code changed in the update to its corresponding commit in the repository that made the change. Algorithm 2 lists the pseudocode for identifying the code changes between two versions and mapping them to their corresponding commits. Depdive then determines if the commit was code-reviewed through four GitHub-based checks. In the following subsections, we explain each step in detail.

### 5.3.1 Collect package code

#### 5.3.1.1 Download package code from the registry:

Depdive downloads package code for both the current and the update version from the respective registries. While Crates.io, npm, and RubyGems provide every package in a uniform format, PyPI packages can be available in multiple formats in the registry. When multiple formats are available, we prefer the default *wheel* distribution.

#### 5.3.1.2 Locate package repository:

We collect the repository of a package from the metadata provided by the registry. Afterward, we locate the directory path of the package within the repository.

A repository can contain source code for multiple packages. Therefore, we need to know the directory path of a package within the repository for an accurate one-to-one mapping of files between the registry and the repository. For example, the filepath *CHANGELOG.md* in the Rust package *tokio* maps to the filepath *tokio/CHANGELOG.md* in the repository.

For Crates.io, npm, and RubyGems, Depdive identifies the directory path by locating the manifest file (*Cargo.toml*, *package.json*, and *Gemspec*) of the package in the repository. While PyPI packages do not contain a uniform manifest file, we locate the directory by matching the filepaths in the registry with the filepaths in the repository. Through directory path locating, Depdive also validates the retrieved repository for a package. Further, a repository can contain submodules that point to different repositories. We obtain commit history recursively for all the submodules in the repository.

#### 5.3.1.3 Map file path from the registry to the repository:

For each file in the registry, we obtain the repository filepath by combining the package directory and the registry filepath. We follow this simple heuristic of filepath matching based on the assumption that packages are bundled by maintaining the same directory structure in the repository [3]. Further, we also resolve the cases where a filepath contains a symbolic reference to another file within the repository.

### 5.3.2 Identify release commit

To compare the package code in the registry and in the repository, and to identify the commits between two versions in the repository, we need to identify the head commit from which a certain version was built and uploaded in the registry. We refer to this commit as the ***release commit*** for a package version.

Depdive identifies the release commit through the git tags in the corresponding repository. Tagging the release commit with the version number is a recommended developmental practice (Zoric 2018) and was followed in prior research work (Goswami et al. 2020). Depdive identifies the release tag and the associated commit for a version through a regular expression match of the tag name with the version number and the package name. If the repository does not contain a single tag to match the given version, Depdive fails to analyze the corresponding update.

[3]We have found this assumption to be generally true in our study.

While not every repository annotates the release commit via git tags or may do it inaccurately, the alternative way is to compare the repository code with the registry code at all the commits in the history and take the commit with the smallest difference (registry code may contain phantom artifacts which will not be present in the release commit, and therefore may not return an exact match with any single commit). Vu et al. developed LastPyMile (Vu et al. 2021) to identify phantom artifacts in a PyPI package by comparing the registry code with every commit in the repository. However, their approach cannot identify the specific release commit for a given version. Therefore, we take the simple heuristic to identify release commits via repository tags.

### 5.3.3 Identify phantom artifacts

Depdive then compares the code diff between the package code of the current and the update version fetched from the registry, and the code diff between the two versions' corresponding release commits in the repository. Here, by ***code diff***, we refer to the diff between two codebases obtained by *git-diff*. However, files and lines of code changes can be present in the registry, but not present within the two release commits in the repository. We refer to such files and code changes as phantom artifacts, following the definition of the term in prior work (Vu et al. 2021). Depdive outputs phantom artifacts in two categories, *phantom files*, and *phantom lines*:

#### 5.3.3.1 Phantom file:

We define phantom files in an update as files that are present in the update version in the registry, but not present in the corresponding repository filepath at the release commit (line 8-14 in Algorithm 1).

#### 5.3.3.2 Phantom line:

We define phantom lines in an update as the lines of code changes that are present in the code diff between the current and the update version, but not present in the code diff in the repository between the corresponding release commits (line 15-24 in Algorithm 2).

### 5.3.4 Map code delta to commits

Besides phantom files and lines, the rest of the code diff between two versions in the registry maps to the code diff between the corresponding release commits in the repository.

**Step 1** Identify Phantom artifacts in an update

```
Require: registry name: $E$
Require: package name: $P$
Require: current version: $X$
Require: update version: $Y$
 1: $P_X = DownloadPackageCodeFromRegistry(E, P, X)$
 2: $P_Y = DownloadPackageCodeFromRegistry(E, P, Y)$
 3: $F_{YP} = GetAllPackageFilepaths(P_Y)$
 4: $R, D = LocateRepositoryAndDirectory(E, P)$
 5: $C_X = IdentifyReleaseCommit(R, X)$
 6: $C_Y = IdentifyReleaseCommit(R, Y)$
 7: $F_{YR} = GetAllPackageFilepathsAtGivenCommit(R, D, C_Y)$
 8: Set of phantom files, $H_{PF} = \emptyset$
 9: for all $f \in F_{YP}$ do
10:     $f_r = GetRepositoryFilePath(f, D)$
11:     if $f_r \notin F_{YR}$ then
12:         $H_{PF} = H_{PF} \cup f$
13:     end if
14: end for
15: Map of phantom lines to files: $M_{PL}$
16: for all $f \in F_{YP} - H_{PF}$ do
17:     $f_r = GetRepositoryFilePath(f, D)$
18:     Code changes in the registry, $D_P = Diff(f, P_X, P_Y)$
19:     Code changes in the repository, $D_R = Diff(f_r, R, C_X, C_Y)$
20:     phantom lines, $P_f = D_P - D_R$
21:     if $P_f \neq \emptyset$ then
22:         $M_{PL}.insert(f, P_f)$
23:     end if
24: end for
25: return Phantom files and phantom lines, $H_{PF}, M_{PL}$
```

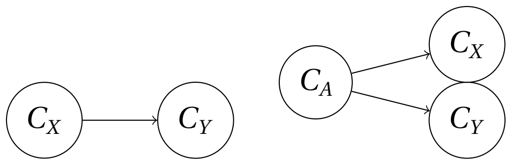
**Step 2** Map code delta to corresponding commits in the repository

```
Require: Release commit of current version: $C_X$
Require: Release commit of update version: $C_Y$
Require: Files changed in update: $F_{XY}$
 1: Commits between $C_X$ and $C_Y$, $L_{XY} = git\_log(C_X..C_Y)$
 2: Common ancestor, $C_A = ParentOf(Oldest(L_{XY})$
 3: Map of added lines to commit: $A_{XY}$
 4: Map of removed lines to commit: $R_{XY}$
 5: for all $f \in F_{XY}$ do
 6:     Git blame: $B_f^Y = git\_blame(f, C_Y)$
 7:     for all $(l, c) \in B_f^Y$ do
 8:         if $c \in L_{XY}$ then
 9:             $A_{XY}.insert(f, (l, c))$
10:         end if
11:     end for
12:     Reverse git blame: $RB_f^X = git\_blame(f, C_A, reverse = True)$
13:     for all $(l, c) \in RB_f^X$ do
14:         if $c \neq C_Y$ then
15:             $rc = findRemovalCommit(f, c, l)$
16:             if $rc \in L_{XY}$ then
17:                 $R_{XY}.insert(f, (l, rc))$
18:             end if
19:         end if
20:     end for
21: end for
22: return Code changes to commit map, $A_{XY}, R_{XY}$
```

However, not every line of change in the code diff between two commit points can be mapped to a specific commit. To explain why, consider the two commit history graphs shown in Figure 5.2, where $C_X$ and $C_Y$ refer to the release commit for version X and Y, respectively. In Figure 5.2a, both $C_X$ and $C_Y$ lie on the same development branch, where $C_X$ is a direct ancestor of $C_Y$. In this case, each line of change in the diff between $C_X$ and $C_Y$ can be mapped to a single commit between the two commits. Contrarily, in Figure 5.2b, $C_X$ and $C_Y$ lie on two different branches with a common ancestor at commit $C_A$. In this case, a line of code that was added in $C_X$, but was not present in $C_A$ and subsequently in $C_Y$, will be shown as a deleted line in the diff between $C_X$ and $C_Y$ (and between the two versions fetched from the registry). However, that line of code was not deleted in any commit, rather just did not appear between $C_A$ and $C_Y$.

Therefore, we define ***code delta*** in an update as code changes that can be mapped to a commit that is present in the update version, but not present in the current version. Following this definition, we only consider the commits between $C_A$ and $C_Y$ in Figure 5.2b as the commits between $C_X$ and $C_Y$ (double-dot commit range as per git syntax). The rationale behind this approach is that when analyzing an update, Depdive assumes that the current version is already trusted by the user, and therefore, focuses analysis only on the code changes between the current and the update versions. Consequently, in Figure 5.2b, we assume that if $C_X$ is trusted by a user, any ancestor commit of $C_X$ is also trusted by the user. Therefore, when updating from $C_X$ to $C_Y$, we only focus on the code changes between their common ancestor $C_A$ and the update version $C_Y$. In the simpler case, where both $C_X$ and $C_Y$ lie on the same branch as shown in Figure 5.2a, $C_X$ itself is the common ancestor of the two versions.

Algorithm 2 shows Depdive's implementation of identifying the code delta between two versions and mapping each line of code in the delta to a corresponding commit in the repository. To identify the newly-added lines in the update version, Depdive runs *git blame* on each package file at commit $C_Y$ to obtain the commits that added each line in the file. If a line was added in a commit that lies between $C_X$ and $C_Y$, we take the line as a newly-added line in the update version (line 6-10 in Algorithm 2). To identify the lines that have been removed in the update, we run *reverse git blame* on each package file at commit $C_A$, to obtain the last commit where each line in the file was present. If a line was still present in $C_Y$, the line was not removed in the update version. For the rest of the removed lines, we identify the removal commit by traversing forward from the blamed commit up to $C_Y$. If the removal commit lies between $C_X$ and $C_Y$, we take the line as a removed line in the



(a) $C_X$ and $C_Y$ are on the same branch where $C_X$ is the common ancestor for both $C_X$ and $C_Y$

(b) $C_X$ and $C_Y$ are on different branches, diverging from the common ancestor commit, $C_A$

Figure 5.2: The relative position of the release commit of version X ($C_X$) and the release commit of version Y ($C_Y$) in the repository commit history.

update (line 12-20 in Algorithm 2). [4]

Along the process, we also obtain the commit mapping for each newly added and removed line of code in the update. Note that, in this and the preceding step, Depdive also handles corner cases where a file has been renamed; a file that already existed in the repository but was newly included in the registry; and a file within a submodule pointing to a different remote repository.

### 5.3.5 Determine if a commit was code-reviewed

After obtaining the commits involved in an update, we need to determine if the commit was code-reviewed. Git does not store data on code review for a commit. Therefore, we focus on the projects that use GitHub as their development platform and look for evidence on GitHub to determine if a commit had been code-reviewed. Specifically, we apply four checks to look for evidence of code review on GitHub. Note that, we adopt these checks from “Security Scorecards” [5], a tool that analyzes the last 30 commits of a GitHub project and generates a score for a project’s adherence to code review. Below, we explain the four checks:

1. GitHub review: We check if a commit belongs to a GitHub pull request and if the pull request was reviewed through GitHub’s native code review tooling.

[4]In the case where the two release commits lie on the same branch (Figure 5.2a), the code delta in the update will be equal to or greater than the code diff between the versions fetched from the registry. The code delta can be greater in cases where a line was modified after the release commit of the current version but was reverted before the update version. Such a line will not appear in the code diff between versions from the registry, but will appear in the code delta calculated by Depdive.

[5]`https://github.com/ossf/scorecard`

Table 5.1: Updates successfully analyzed by Depdive from the selected packages

| Registry | No. of Selected Packages | No. of Selected Updates | No. of Packages analyzed by Depdive | No. of Updates analyzed by Depdive |
|---|---|---|---|---|
| Crates.io | 990 | 8,434 | 833 (84.1%) | 6,326 (75.0%) |
| npm | 989 | 8,158 | 919 (92.9%) | 7,178 (88.0%) |
| PyPI | 992 | 8,657 | 788 (79.4%) | 6,089 (70.3%) |
| RubyGems | 991 | 8,646 | 674 (68.0%) | 5,351 (61.9%) |
| Total | 3,962 | 33,895 | 3,214 (81.1%) | 24,944 (73.6%) |

2. Different merger: We check if a commit belongs to a GitHub pull request and if the pull request was opened and merged by two different GitHub accounts. We assume the action of merging as an implicit approval of the commit by the merger.

3. Different committer: We check if a commit was authored and committed by two different accounts. We assume the commit was approved by the committer.

4. Third-party tools: Besides GitHub's native code review tooling, a project can use external code review tools, e.g., Gerrit, or custom bots to handle code review, e.g., Prow. When a commit is reviewed on Gerrit, the commit message contains metadata for the Gerrit review. When a pull request on GitHub is handled by Prow, Prow adds a label to the pull request to indicate if it was code-reviewed. Following Security Scorecards' implementation, we also check for evidence for Gerrit or Prow review in Depdive.

We determine a commit as code-reviewed if any one of the four checks is met. When checking for different merger or committer, we do not consider a commit to be code-reviewed if both the author and the reviewer are bot accounts on GitHub. We further exclude cases where GitHub's own bots are involved (e.g., GitHub actions). However, in cases where a pull request has been opened by a bot, e.g., Dependabot, and merged by a non-bot account, we consider the commit as code-reviewed.

## 5.4 Dataset

In this section, we explain the packages and their updates we selected for this study, and the updates that Depdive could successfully analyze based on which we complete our empirical analysis.

### 5.4.1 Package Selection

We select the latest ten updates of the most downloaded 1000 packages in each of the four package registries, namely Crates.io, npm, PyPI, and RubyGems. For Crates.io and RubyGems, we download the official data dumps that are updated daily and then select the top packages in the order of the total download count across all the versions. For PyPI, we use the dataset from `https://zenodo.org/record/5812615#.ZESDkOzMJb8` that is updated on a monthly basis. We collected the data for these three registries at the end of December 2021. For NPM, we use the dataset from (Zahan et al. 2021) that was constructed in August 2021. While the download count can be inflated in different ways, including through CI/CD tooling, sampling the most downloaded packages to study a package ecosystem is an established approach (Vu et al. 2021; Bommarito and Bommarito 2019) and provides an estimation of the most used packages in a registry.

We then collect the list of the available version releases on the registry for each of the selected packages at the end of December 2021. Depdive can run on an update from any version to another (e.g., 1.8.3 $\rightarrow$ 1.8.4, 1.5.1 $\rightarrow$ 1.9.0, 1.0.0 $\rightarrow$ 3.0.3 etc.). However, for this empirical study, for each version, we consider an update only from its prior version according to SemVer (Preston-Werner 2021) ordering. For example, for *tokio@1.8.4*, we consider an update from *tokio@1.8.3* to *tokio@1.8.4* to pass as input to Depdive. We choose this approach to avoid any data duplication, that is, the same code changes and commits to appear in multiple updates of a package selected in this study. Further, we exclude any pre-release version in our data set.

We also restrict our data set to only the most recent ten updates for each of the packages. The rationale behind this is that Depdive makes multiple GitHub API calls for each commit in an update for code-review checking. However, GitHub limits API calls to 5000/hr for each user, which puts a constraint on how much analysis we can do within an hour. However, choosing at least ten updates will provide us the data to measure the consistency of a package in its code review coverage in each update. In the case where a package has less than ten updates, we consider all of them in our study.

Out of the 4000 packages, 26 packages had only one release listed on the respective registry. Further, 12 packages had only one regular release available, while the other available versions were pre-releases. We exclude these 38 packages as there were no updates available to analyze. Finally, we select 33,895 updates from 3,962 packages to run Depdive on. Table 5.1 shows a breakdown of these initially selected updates across the four registries.

### 5.4.2 Depdive analysis

Out of the selected 33,895 updates, Depdive successfully analyzed 24,944 (73.6%) updates. In Section 5.5, we present our empirical analysis based on Depdive's output for these 24,944 updates. Table 5.1 shows a breakdown of the dataset of this study. While Depdive outputs details on each commit, phantom artifact, and lines of code changed in an update, we collect the following metrics for each update to conduct our empirical analysis: 1. No. of phantom files, 2. No. of files with phantom lines, 3. No. of added phantom lines [6], and 4. Code Review Coverage (*CRC*).

Note that, we measure the *CRC* of an update as the proportion of the update's code delta that has been code-reviewed. Here, the code delta, as explained in Section 5.3.4. is the code changes in an update that can be mapped to a commit in the repository and therefore, can be classified if code-reviewed or not

### 5.4.3 Why Depdive failed

As shown in Table 5.1, Depdive only successfully analyzed 73.6% of the initially selected updates. The primary reason that Depdive could not analyze an update was the absence of git tags in the repository pointing to the release commit of a version, which was the case for 5,747 updates (17.0%) in our dataset. The next major reason is Depdive's inability to either locate or validate the repository of a package, which was the case for 2,669 of all the updates (7.9%). For 127 updates (0.4%), the listed repository was not on GitHub. For the rest of 1.1% of the updates, Depdive failed for various reasons, including not being able to read a file containing non-Unicode characters, a private submodule within the repository, and version code not being available on the registry.

If Depdive were to be deployed in a CI/CD pipeline, our empirical evaluation shows that the tool could fail in 26.4% of the cases. However, the failure rate may be reduced if the package repositories followed the best developmental practices, such as annotating a git tag for each released version. We recommend that package manager tools add a feature to include metadata on the repository, package directory, and release commit in the package bundle during the build process to aid in a third-party audit of each update.

[6]While Depdive also outputs phantom removal, we observe that the removed lines are often also phantom lines from the old version.

Table 5.2: Updates with phantom files, i.e., files present in the registry but not in the repository.

| Registry | Total packages | Total Updates | Packages with phantom files | Updates with phantom files | Median phantom file count |
|---|---|---|---|---|---|
| Crates.io | 833 | 6,326 | 42 (5.0%) | 123 (1.9%) | 3.25 |
| npm | 919 | 7,178 | 413 (44.9%) | 2,489 (34.7%) | 2.00 |
| PyPI | 788 | 6,089 | 355 (45.1%) | 2,088 (34.3%) | 1.50 |
| RubyGems | 674 | 5,351 | 71 (10.5%) | 323 (6.0%) | 2.00 |
| Total | 3214 | 24944 | 881 (27.4%) | 5023 (20.1%) | 2.00 |

Table 5.3: Updates with phantom lines, i.e., changes in a file in the registry but not in the repository

| Registry | Total packages | Total Updates | Packages with phantom lines | Updates with phantom lines | Median file count with phantom lines | Median count of added phantom lines |
|---|---|---|---|---|---|---|
| Crates.io | 833 | 6,326 | 63 (7.6%) | 87 (1.4%) | 1.0 | 2.0 |
| npm | 919 | 7,178 | 265 (28.8%) | 1,733 (24.1%) | 1.0 | 1.0 |
| PyPI | 788 | 6,089 | 157 (19.9%) | 471 (7.7%) | 1.0 | 3.0 |
| RubyGems | 674 | 5,351 | 105 (15.6%) | 185 (3.5%) | 1.0 | 2.0 |
| Total | 3,214 | 24,944 | 590 (18.4%) | 2,476 (9.9%) | 1.0 | 2.0 |

Depdive could successfully analyze 73.6% of the updates in our dataset. The primary reason behind Depdive's failure was the absence of a release tag for a package version. We recommend package maintainers include the metadata on release commit in the package bundle to aid third-party audit tools.

## 5.5 Empirical Analysis

Based on Depdive's output, we answer the following two research questions for the four studied registries: ***RQ1:*** To what extent do phantom artifacts exist in the updates of the most downloaded packages? and ***RQ2:*** Excluding phantom artifacts, what is the code review coverage in the updates of the most downloaded packages? In the following two subsections, we present our findings:

### 5.5.1 Phantom artifacts

Table 5.2 shows the occurrence of phantom files, and Table 5.3 shows the occurrence of phantom lines in updates across the four studied registries. Overall, 24.9% of the updates contained either a phantom file or a phantom line (3.2%, 44.3%, 38.5%, and 9.1% in the case of Crates.io, npm, PyPI, and RubyGems updates, respectively). Below, we discuss our findings:

#### 5.5.1.1 Phantom files:

We find that 20.1% of the analyzed updates and 27.4% of the analyzed packages had at least one phantom file. We find that npm and PyPI updates are more likely to contain a phantom file (34.7% and 34.3%, respectively) than Crates.io and Ruby Gems updates (1.9% and 6.0%, respectively). Overall, we identified 306,940 phantom files across all the updates. Below, we provide a crude characterization of the phantom files that we identified:

Across all the npm updates, we identified 123,000 phantom files. We observed that for some npm packages, the code in the registry can be a transpiled version of the source code in the repository. For example, the source code may be written in TypeScript, whereas the package contains transpiled JavaScript code. Similarly, npm packages can also contain minified JavaScript, or JavaScript transpiled through transcompilers such as *babel*. Overall, 89.4 % of the phantom files in the npm updates are JavaScript files (*.js, .d.ts, .cjs, .mjs, .min.js, .map, .flow* files) which may be code transpiled during the package build process. Besides transpiled code, 9.5% of the npm phantom files are *.json* files which are either data or configuration files, and possibly were git-ignored in the repository through the *.gitignore* file.

We identified 146,564 phantom files across all the PyPI updates. The majority of these files (46.1%) are machine-generated header files for C/C++ (*.h, .hpp, .inc* files), while 7.4% of the phantom files are compiled binaries (*.so, .jar, .dylib*). The core engine of many Python packages, e.g., *tensorflow*, are written in languages like C/C++, while the Python files only provide a front-end to communicate with the engine. While the repositories of these packages contain the source code for the engine written in different languages, the Python packages, in their default *wheel* distribution, only contain the compiled binaries and the machine-generated header files alongside the Python source files. The header files and the binaries come from 118 distinct packages in our dataset (15.0% of the analyzed PyPI packages). Moreover, we find 16.3% of the phantom files to be Python files, where one-third of them (32.1%) are *__init__.py* files which are presumably machine-generated. Finally, in PyPI updates, we have also observed non-Python code files (7.1% are *.js, .ts* files), data files (5.0% are *.dat* files), and git-ignored files (e.g., *.pyc, .pyi, .py* files) as phantom files.

Table 5.4: Top 10 phantom file types in Crates.io packages

| File extension | Count |
|---|---|
| .html | 24,017 (80.3%) |
| .def | 668 (2.2%) |
| .md | 590 (2.0%) |
| .js | 570 (1.9%) |
| .c | 502 (1.7%) |
| .LICENSE | 421 (1.4%) |
| .h | 408 (1.4%) |
| .rs | 231 (0.8%) |
| .txt | 227 (0.8%) |
| .py | 172 (0.6%) |

Table 5.5: Top 10 phantom file types in npm packages

| File extension | Count |
|---|---|
| .js | 61056 (49.6%) |
| .ts | 23387 (19.0%) |
| .map | 13898 (11.3%) |
| .json | 11688 (9.5%) |
| .flow | 11095 (9.0%) |
| .npmignore | 420 (0.3%) |
| .cjs | 315 (0.3%) |
| .html | 302 (0.2%) |
| .mjs | 232 (0.2%) |
| .md | 102 (0.1%) |

Across all the RubyGems updates, we found 7,483 phantom files. However, 83.2% of these files come from 7 packages where the package either (i) bundled its own Ruby dependency packages, (ii) bundled non-Ruby dependency code, or (iii) included log files that were git-ignored in the repository. Across Crates.io updates, we identified 29,893 phantom files. However, 84.4% of these files come from a single package that contained the source code for its project website hosted in a repository branch. Overall, across all the four registries, we have observed files that are git-ignored in the repository, to be included in the registry (e.g., *.DS_Store, .npmignore, .editorconfig, setup.cfg* files). Table 5.4, 5.5, 5.6, 5.7 show the top 10 phaton file types in each of the four studied registries.

**Accidental vs legitimate phantom files:** Broadly, we can characterize the phantom files into two categories: (i) phantom files due to legitimate reasons such as compiled binaries in PyPI *wheel* distribution, transpiled JavaScript, auto-generated files, and third-

Table 5.6: Top 10 phantom file types in PyPI packages

| File extension | Count |
|---|---:|
| .h | 59917 (40.9%) |
| .py | 23983 (16.4%) |
| .so | 7738 (5.3%) |
| .dat | 7479 (5.1%) |
| .js | 5527 (3.8%) |
| .ts | 5050 (3.4%) |
| .inc | 4486 (3.1%) |
| .map | 3170 (2.2%) |
| .hpp | 3139 (2.1%) |
| .pyi | 2681 (1.8%) |

Table 5.7: Top 10 phantom file types in RubyGems packages

| File extension | Count |
|---|---:|
| .rb | 3064 (40.9%) |
| .cache | 1033 (13.8%) |
| .h | 753 (10.1%) |
| .c | 655 (8.8%) |
| .md | 156 (2.1%) |
| .txt | 135 (1.8%) |
| .rbc | 120 (1.6%) |
| .rdoc | 90 (1.2%) |
| .y | 86 (1.1%) |
| .cc | 69 (0.9%) |

Table 5.8: Code review coverage of the analyzed updates

| Registry | Total Package | Total Updates | Median LOC changes in updates | Median code review coverage (*CRC*) in updates | Updates with 100% *CRC* | Updates with 0% *CRC* |
|---|---|---|---|---|---|---|
| Crates.io | 830 | 6,293 | 75.75 | 42.9% | 915 (14.5%) | 2,068 (32.9%) |
| npm | 918 | 7,147 | 25.00 | 5.9% | 87 (1.2%) | 3,497 (48.9%) |
| PyPI | 780 | 5,861 | 76.75 | 52.7% | 1,425 (24.3%) | 1,760 (30.0%) |
| RubyGems | 672 | 5,309 | 61.25 | 31.0% | 269 (5.1%) | 1,663 (31.3%) |
| Total | 3,200 | 24,610 | 51.00 | 27.2% | 2,696 (11.0%) | 8,988 (36.5%) |

Table 5.9: Packages with all updates fully code-reviewed, or not code-reviewed at all.

| Registry | Total Packages | Packages with all updates having 100% *CRC* | Packages with all updates having 0% *CRC* |
|---|---|---|---|
| Crates.io | 830 | 99 (11.9%) | 217 (26.1%) |
| npm | 918 | 5 (0.5%) | 386 (42.0%) |
| PyPI | 780 | 162 (20.8%) | 185 (23.7%) |
| RubyGems | 672 | 23 (3.4%) | 161 (24.0%) |
| Total | 3,200 | 289 (9.0%) | 949 (29.7%) |

party dependency code; and (ii) phantom files presumably put in mistakenly, such as the git-ignored files.

For the legitimate phantom files, we need to track their provenance, and then determine if changes in the origin files have been code-reviewed or not, and incorporate that into the overall *CRC* of the update. We discuss current research efforts and our recommendations on handling legitimate phantom files in Section 5.8. On the contrary, we recommend package maintainers issue a new clean release in case of accidental phantom files, as the *last mile* between the repository and the registry has been used as an attack vector to sneak in malicious code in the past (Vu et al. 2021; Ohm et al. 2020).

Overall, while there may be legitimate reasons behind phantom files, we cannot audit the changes in these files through our proposed approach. Package users may manually review these files before accepting an update, especially when the current version does not have any phantom files, but the new update does. In our dataset, 461 packages (52.1% of all the packages with phantom files) had phantom files only in a subset of all their updates. We presume that these packages may have accidentally put in phantom files in some of their updates.

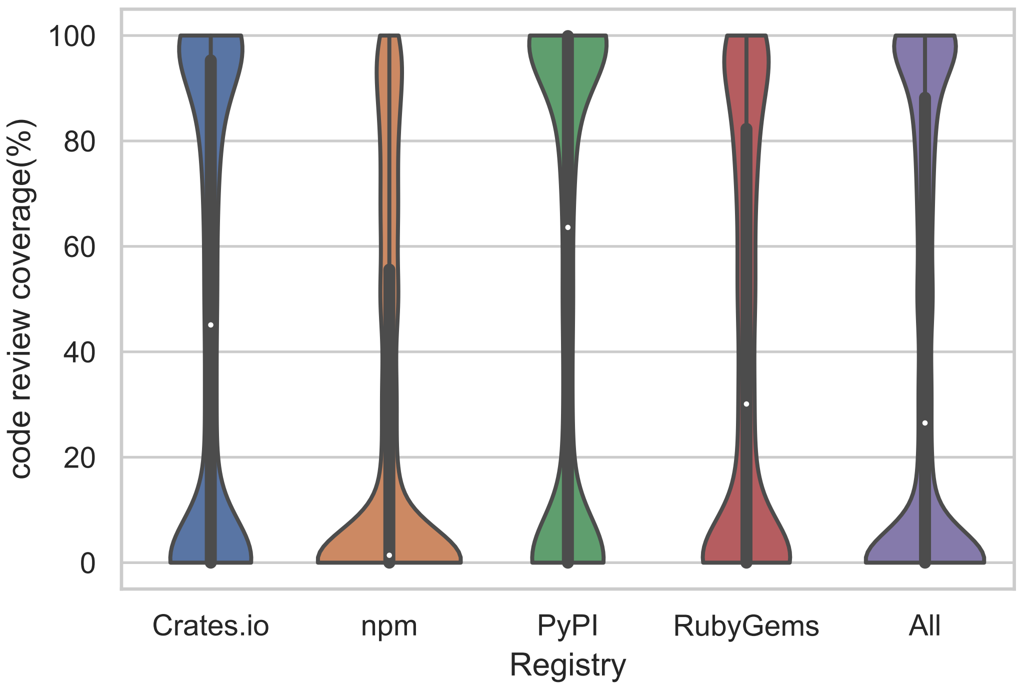


Figure 5.3: Violin plot of code review coverage across updates

#### 5.5.1.2 Phantom lines:

We find that 9.9% of the analyzed updates across 18.4% of the packages had code changes in an update that were not present in the changes of the corresponding files in the repository. We find that phantom lines are generally small changes, with 2 added lines in 1 file at the median. We also find that npm updates are more likely to contain phantom lines than updates in the other three studied registries.

We identified phantom lines in 1,733 (24.1%) npm updates, where 93.9% of these updates contained phantom lines in the manifest file, *package.json*, that was presumably dynamically generated with added data such as the release commit. Further, as explained when discussing phantom files, the npm package code in the registry can be transpiled from the code in the repository. Therefore, the same filepath may have different code content in the registry and the repository, resulting in many updates with phantom lines.

Across PyPI packages, we identified 471 updates (7.7%) with phantom lines, where 51.6% of the updates had phantom lines in a file named *_version.py* that was presumably dynamically generated during the package build process with added data such as the release commit, the build date, and the version number. Further, *__init__.py* files can also be generated dynamically during the build process and may differ in content from the repository copy, which resulted in phantom lines in 117 updates (24.8%). For both Crates.io and RubyGems, we do not find any common pattern in the identified phantom lines. However, false positives may appear in the case where a release commit was inaccurately tagged.

While some files can be dynamically generated during the package build process and incorporate phantom lines, overall, we find that phantom lines are less likely to occur due to legitimate reasons in Crates.io, PyPI, and RubyGems updates, and package users should manually review these lines before accepting an update.

We find that phantom artifacts are not uncommon in the updates (20.1% of the analyzed updates had at least one phantom file). The phantoms can appear either due to legitimate reasons, such as in the case of programmatically generated files, or from accidental inclusion, such as in the case of files that are ignored in the repository. However, without provenance tracking, we cannot audit if the changes in these phantom artifacts were code-reviewed or not.

## 5.5.2 Code review coverage

In this section, we present our findings on code review coverage (*CRC*) for the analyzed updates. Note that, we exclude phantom artifacts in *CRC* measurement, and only consider the changes in the package code that can be mapped to the repository. Table 5.8 presents the median lines of code (LOC) changes and the median *CRC* across updates in the four registries. We excluded 334 updates from the 24,944 analyzed updates as they contained zero code delta as measured by Depdive. These updates either only made changes in the non-package files in the repository, or only made changes in the phantom files. Further, in our dataset, we identified 110,657 commits as code-reviewed, of which 60.1% were GitHub review, 31.5% were Different merger, and 8.4% were Different committer, as per the checks explained in Section 5.3.5.

The table also shows the portion of the updates with 100% and 0% *CRC* across the four registries. We find that 11.0% of the updates were fully code-reviewed, while 36.5% of the updates were not code-reviewed at all. The rest of the 52.5% of the updates were only partially code-reviewed. Further, we find that the median *CRC* across all the analyzed updates stands at 27.2%.

We find the npm packages to have the lowest median *CRC*. In Section 5.5.1, we explained that npm packages may contain transpiled JavaScript, resulting in many phantom artifacts. In these cases, Depdive's *CRC* measurement would be limited to only a subset of the package files. For example, for the *no-case* package, Depdive could only audit the package manifest files for code review. To address this limitation, we looked at the *CRC* for the 3,947 npm updates that did not contain any phantom artifacts with a presumption that such updates are less likely to contain transpiled code. However, we still observe a low median *CRC* of 6.1% for these updates. While many npm packages are small in size and maintained by a few maintainers resulting in many code changes remaining non-reviewed, our analysis in this study may be an under-approximation for npm, as the packages that contain transpiled JavaScript, e.g., packages with source code in TypeScript, may also be more likely to follow developmental best practices such as code review. We have observed npm packages from reputed organizations like *babel* and *facebook* to develop their source code in TypeScript and adhere to code review in the majority of the commits in the corresponding repositories. However, Depdive could only audit a subset of the files in these packages, and therefore, we may have got an under-approximated *CRC* measurement.

For Crates.io, PyPI, and RubyGems packages, we find the median *CRC* to be 42.9%, 52.7%, and 31.0% respectively. Figure 5.3 shows a violin chart of the *CRC* across the updates.

We observe an *hourglass* shape in the violin chart, suggesting that updates tend to have either very high *CRC* or very low *CRC*. Open source packages are often maintained by a small group of maintainers. While the contributions from an outsider get code-reviewed by one of the maintainers, the code changes from the maintainers themselves may remain non-reviewed, therefore resulting in low-to-medium *CRC* overall. On the contrary, we have found updates that can have very high *CRC* but are not fully code-reviewed. For example, we have found 2,442 updates (9.9%) that are larger than 51 LOC changes (overall median) and have greater than 90% but less than 100% *CRC*.

While most of the commits in these updates were code-reviewed, we found some commits did not go through the code review process, possibly due to the following two primary reasons as per our observation: (i) *Non-critical changes:* the commit only changed configuration or documentation files; (ii) *Cherry-picked commits:* the commit cherry-picked a commit from a different branch where it was code-reviewed either to backport a fix or to import changes to a release branch from the master branch. Note that, SLSA requires context-specific approval during a code review, which means the cherry-picked commits require their own separate review. Nonetheless, this phenomenon shows that non-reviewed code changes may sneak in even in packages that attempt to adhere to code review. A few notable packages where we found high *CRC* but not 100% include *numpy, ansible-core* in PyPI, *nokogiri* in Ruby, *openssl* in Crates.io, etc.

Table 5.9 shows packages for whom all the analyzed updates in our data set were measured to have 100% *CRC* or all the updates were measured to have 0% *CRC*. We find that only 9.0% of the packages in our dataset had all their updates fully code-reviewed, most of them coming from PyPI and Crates.io. We observed that packages from reputed organizations such as *google, Azure, rust-lang, tokio-rs*, and *aws* are likely to consistently have fully code-reviewed updates. On the contrary, 29.7% of the packages had none of their updates code-reviewed at all. We have observed that these packages are typically maintained by a small group of maintainers who typically do not review each other's code.

We find that the updates are typically only partially code-reviewed (52.5% of the time). Further, only 9.0% of the packages had all their updates in our data set fully code-reviewed, indicating that even the most used packages can introduce non-reviewed code in the software supply chain. We also observe that updates either tend to have very high *CRC* or very low *CRC*, suggesting that packages at the opposite end of the spectrum may require a separate set of treatments.

## 5.6 Maintainer Agreement Survey

We conducted a survey to evaluate whether the studied packages' maintainers agreed with the Depdive analysis. We emailed maintainers of the packages and provided an analysis report of a randomly-selected update (of that package) and asked two Likert scale questions based on the analysis: 1. *Do you agree with our analysis?* 2. *How often do you require code review?* We also gave options for the respondents to explain their answers.

We sent out the survey through email to the maintainers of 945 packages. We received 96 responses, with a response rate of 10.2%. We did not survey packages where 1. we could not collect the maintainers' valid email addresses; 2. we already emailed the maintainer querying on a different package they also own; and 3. the package had phantom artifacts resulting in incomplete analysis on our part.

Figure 5.4 shows a pie chart of maintainer agreement to our analysis. 47 (49.0%) respondents fully agreed to our analysis, while 33 (34.4%) respondents partially agreed. The remaining 16 (16.7%) respondents disagreed. In 48 cases, the maintainers provided the reasoning behind their disagreement. We classified the reasoning into the following categories:

1. **Non-functional changes (15):** Maintainers noted that the unreviewed commits did not change the source code of the package, rather only configuration files, e.g. package manifest files, CI/CD scripts, and documentation files, e.g. README.md, CHANGELOG.md. The maintainers disagreed with our analysis, noting that changes in these files should not be included in the *CRC* analysis.

2. **Single maintainer (8):** Maintainers disagreed on the rationale that *CRC* analysis is invalid for their packages, as they are the sole maintainer.

3. **Review outside GitHub (7):** In 7 cases, maintainers noted that the commits flagged as unreviewed were indeed reviewed, either through an internal code review tool, or through discussion on GitHub.

4. **Misunderstanding of the analysis (7):** There were 7 cases where maintainers disagreed for reasons that appear to be a miscommunication on our part regarding what the tool should do. For example, maintainers noted that Depdive missed some commits in the surveyed update. When investigating those updates, we found that the omitted commits were on non-package files, e.g., test files or CI/CD scripts, which were not present in the registry, and therefore, were not included in the *CRC* analysis.

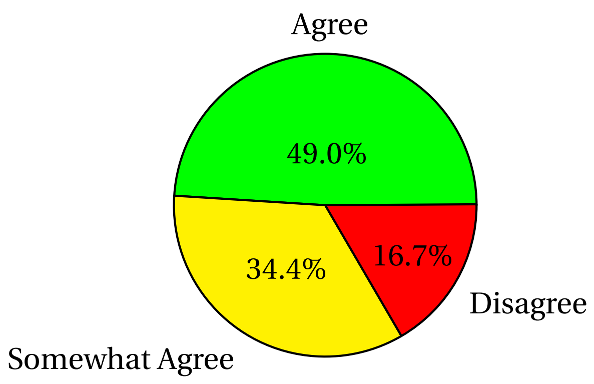


Figure 5.4: Maintainers' agreement rate with Depdive analysis.

5. **Trivial changes (5):** Maintainers noted that the unreviewed commits only consisted of trivial changes, e.g., code re-formatting, and therefore, does not require a code review. They viewed flagging such commits as unreviewed as false alarms.

6. **Others (6):** While we marked dependabot PRs merged by a developer as reviewed, two maintainers opined that they should be marked as unreviewed. In one case, the maintainer disagreed that the backported commits needed to be reviewed again. In another case, the unreviewed commit was only a merge commit. In one case, the package only included auto-generated content. In another case, the package is now unmaintained.

Finally, Figure 5.5 shows maintainers' responses to how often they require code review for new contributions in their package. We observe that the responses are in alignment with our empirical findings in Table 5.9. Only 12.5% of the packages always require code review, while 41.7% of the packages often require code review. On the other hand, 28.2% of the packages either never or rarely require a code review.

We found that maintainers largely agree with our analysis. While there are cases where the code review took place outside GitHub (and therefore, flagged as unreviewed by Depdive), the maintainers' disagreement primarily stemmed from their opinion that some changes (e.g., non-functional changes) should not be included in the *CRC* analysis.

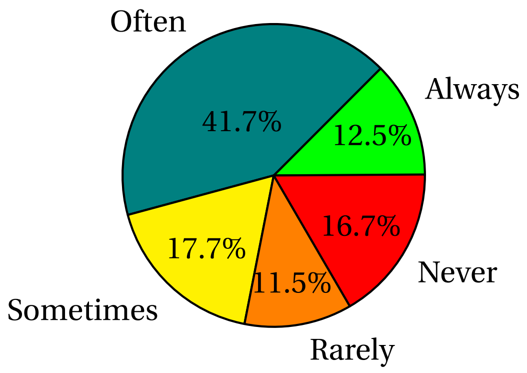


Figure 5.5: Maintainers' responses to code review requirement in their packages.

## 5.7 Limitations

**Depdive design decisions:** We adopt multiple heuristics in designing Depdive that may result in false-positive outputs. Firstly, we map package files in the registry to a file in the repository through filepath matching. However, package maintainers may choose a build process where the filepaths will be altered in the bundled package without altering the content of the file, in which case Depdive will output them as phantom files. Similarly, we may fail to validate a correct repository and locate the package directory when the directory structure in the registry code and the repository do not match. While an alternate approach could be a pairwise comparison of all the files, our simpler heuristic is based on a realistic assumption that bundled packages typically follow the same directory structure as that in the repository (unless the files are programmatically generated). Secondly, we leverage repository tags to determine the release commit for a package version, inaccuracy in which may incorrectly output phantom artifacts. We explained this design decision in Section 5.3.2 and recommend that packages themselves should contain the metadata for the release commit.

Overall, we chose our heuristics to keep the design simple while minimizing the possibility of false negatives. In this regard, our empirical analysis of the phantom artifacts may represent an over-approximation. Similarly, Depdive scopes its code-review checking to the GitHub platform, and therefore, may output false positives in the case where a project is hosted on the GitHub platform but uses an external or an atypical code review tool. While we only chose packages that list a GitHub repository in their provided metadata for the empirical study, our findings on code review coverage may represent an under-approximation in such cases. We aim to incorporate more checks in the future for code review and recommend maintainers list their code review tooling within the package metadata.

**Generalizability threat:** Our work studies the most downloaded packages in four package registries. However, as shown in Table 5.4, we failed to analyze 26.4% of the initially selected updates. The failure may introduce unknown sampling biases in our dataset for the empirical analysis. Further, the threat to the *CRC* analysis for the npm ecosystem due to the presence of transpiled code has been explained in Section 5.5.2. Nonetheless, we believe our empirical findings provide an evaluation of our proposed approach and offer many insights into the existence of phantom artifacts and *CRC* among the most downloaded packages.

## 5.8 Discussion

### 5.8.1 Depdive design philosophy

One driving philosophy behind Depdive is that every line of code in the software supply chain should be traceable to at least two owners. To achieve that, we need to map the package code downloaded from the registry to the commits in the package's repository and then reliably determine the author and the reviewers of the mapped commits. However, we face two challenges in the process:

**Provenance of the legitimate phantom artifacts:** We find that legitimate phantom artifacts, such as the programmatically generated files, can appear in a package. Especially, transpiled JavaScript code in the npm packages and compiled binaries in the PyPI packages make Depdive's audit incomplete without ensuring the integrity of the origin of these files. Downstream projects can address this issue by retrieving package code directly from the source repository and then building the package by themselves. PyPI already has a feature to provide a source distribution alongside the default *wheel* distribution that contains the source code to compile the required binaries and machine-generated files on the client's end. Clients then can run Depdive on the source distribution for a complete *CRC* measurement. We recommend npm also consider this distribution model.

Another way is to divide the update audit into two parts: (i) have a framework, such as the *in-toto* (Torres-Arias et al. 2019) and the *Reproducible Builds*[7], to verify that the package has indeed come from a certain commit point in the repository, and (ii) measure *CRC*

[7]`https://reproducible-builds.org/`

directly from the repository (without the step in Algorithm 1). However, these approaches have their own pros and cons. We recommend future research to investigate designing a package distribution model that minimizes the efforts required from the stakeholders while ensuring the integrity of the packages.

**Reliability of code review:** While we pitch Depdive as an audit tool, a rogue/hijacked maintainer account can easily bypass Depdive checks by creating a pull request from a sock account and then reviewing and accepting that pull request. SLSA requirement states that two trusted persons should be involved in a code review. Therefore, a tool like Depdive should also consider the digital identity of the involved developer accounts and flag any suspicious review. Similarly, the diversity and the expertise of the reviewers should also indicate the overall reliability of a code review. We consider providing a reliability rating for each code review and flagging suspicious reviews as future work for Depdive.

**Difference with other tools:** There are existing tools that either (i) measure the code review adherence of a project, or (ii) identify phantom artifacts in a package. "Security Scorecard" [8] gives a score for a GitHub project's code review adherence by looking at the branch protection rules and the review history of the last 30 commits. While such a one-time check can help in selecting a dependency, we aim to measure the code review coverage during *each update*, which presents an added challenge of mapping the code changes from the registry to the repository commits. Further, Vu et al. (Vu et al. 2021) have developed LastPyMile to identify phantom artifacts in a PyPI package by comparing the package code with every commit in the repository. While Depdive's approach has methodological differences with LastPyMile as explained in Section 5.3.2, both the tools should return similar results (for phantom artifacts in PyPI updates). In a way, Depdive brings the above two tools' objectives into a single workflow and provides an isolated audit only for the changes in an update. Such an audit will help package users focus their review effort on the incremental changes in each update.

## 5.8.2 The state of the package ecosystems

Through our empirical evaluation of Depdive, we also present our findings on the code review coverage in the recent updates of the most downloaded packages. Below, we discuss some implications of our findings:

**The *hourglass* phenomenon:** We have seen that packages either tend to have very high *CRC* or very low *CRC*, as depicted by an hourglass shape in Figure 5.3. The packages

[8] `https://github.com/ossf/scorecard`

with high *CRC* should enforce strict branch protection in the GitHub project to reject any non-reviewed commits, or not include any non-production files that have loose restrictions on code-review in the package. On the other end of the spectrum, packages with low *CRC* may be in need of more manpower. Future research may look at recommending reviewers for packages that are highly used but maintained by a small group of maintainers.

**Post-release code vetting:** The trade-off that comes with code review is a slowed-down development, while maintainers of some packages may not welcome reviewers in their projects. Further, multiple maintainers can collude in developing a package and wait until the package becomes sufficiently popular before pushing in a backdoor as part of a long-term cyberattack. An alternate approach to code review can be the post-release crowd-vetting of the code changes in a new update. Package registries can provide a system where developers from all around the world can review and approve each new release of a package, while the client projects can wait a certain period until an update has garnered enough approval before accepting it in their codebase.

## 5.9 Conclusion

We implement Depdive, a dependency update audit tool for Crates.io, npm, PyPI, and RubyGems packages, that first (i) identifies the files and the code changes in an update that cannot be traced back to the package's source repository, i.e., *phantom artifacts*, and then (ii) measures what portion of the changes in the update excluding the phantom artifacts has passed through a code review process, i.e., *code review coverage*. Depdive can help package users in focusing their review effort on the phantom artifacts and the non-reviewed code when pulling in a new update, while also providing a quality estimate of the incoming changes. We ran Depdive over the latest ten updates of the most downloaded 1000 packages in each of the four above-mentioned registries, of which Depdive could successfully analyze 73.6% of the updates.

Overall, from our empirical evaluation over 24,944 updates across 3,214 packages, we present interesting insights regarding the studied packages ecosystems. We find that phantom artifacts are not uncommon in the updates, either due to legitimate reasons, such as in the case of programmatically generated files, or from presumably accidental inclusion, such as in the case of git-ignored files. While phantom artifacts are rare in Crates.io and RubyGems updates, we find that npm and PyPI updates can commonly have phantom artifacts in the form of transpiled JavaScript code, compiled binaries, and other machine-generated files.

Regarding code review coverage, we find that updates are typically only partially code-reviewed (52.5% of the time). Further, only 9.0% of the packages in our dataset had all their updates fully code-reviewed, highlighting the fact that even the most used packages ship non-reviewed code. We also observe that updates tend to have either very high *CRC* or very low *CRC*, indicating that packages at the opposite end of the spectrum require different treatments. Overall, this study provides an empirical evaluation of our proposed approach to auditing a dependency update and an ecosystem-level analysis of code review coverage among the latest updates of the most downloaded packages.

CHAPTER

6

# SOCIAL NETWORK-BASED CENTRALITY RATINGS FOR RUST DEVELOPERS

## 6.1 Motivation & Research Questions

In Section 5, we have proposed a quality control method during dependency updates that measures code review coverage in a package update to determine what portion of changes in the update has been reviewed by a second developer other than the author. Further, SLSA (Supply chain Levels for Software Artifacts), a security framework for using open source packages, also suggests a two-person code review requirement as an industry best practice [1]. Software practitioners have developed tooling so that third parties can review code in already published packages, such as *cargo-crev* [2] and *cargo-vet* [3]. However, the mere presence of a review approval in the open source world may not ensure that the involved developers, both authors, and reviewers, are trustworthy.

A user account management system can help authenticate the users and ensure their

[1]https://slsa.dev/
[2]https://github.com/crev-dev/cargo-crev
[3]https://github.com/mozilla/cargo-vet

trustworthiness within an organizational context. However, the open source world has no centralized account management system (Hawthorne and Perry 2005). Brewer et al. (Brewer et al. 2021) noted that a lack of user authentication might make code reviews in open source unreliable, as developer accounts on online coding platforms such as GitHub cannot be assumed to be independent or trustworthy. They called for a federated model for developer digital identities (Brewer et al. 2021).

However, to the best of our knowledge, we do not know of any such identity model for open source developers to date. A rating or ranking attributed to each developer that reflects their standings within a community can help establish such an identity model. In this work, we conjecture that the public development history of open source packages can be leveraged to build such a rating system. We propose constructing a collaboration graph of the developers using the development history, referred to as developer social network (DSN) in the literature (Herbold et al. 2021). We can then rate the developers based on their positional properties in the graph. The rating (or ranking within the community) can help understand the trustworthiness of open source developers. For example, code changes that are authored and reviewed by top-ranked developers may be more trustworthy. On the other hand, changes coming from newcomers or fringe developers should be reviewed more carefully by the downstream projects.

The goal of this study is to help downstream project developers prioritize review efforts for upstream code by providing a social network-based centrality rating for the authors and reviewers of that code. To that end, we study the Rust ecosystem in this paper —that is— the packages hosted on Crates.io [4]. We conduct our study by investigating the following research questions:

> **RQ1:** To what extent are developers from the most downloaded Rust packages interconnected through collaboration?

Towards RQ1, we construct a social network of developers from the most downloaded Rust packages based on their collaboration activity within these packages, such as author-reviewer collaboration, and investigate how interconnected the developers from the different packages are. If the developers are interconnected, a social network analysis can provide meaningful signals for an individual developer's position in the network (Sherchan et al. 2013).

> **RQ2:** What is the distribution of the developer ratings based on their network centrality?

[4]`https://crates.io/`

Towards RQ2, we propose rating developers using an aggregation of five centrality measures computed from our constructed network. Prior work in social networks, including developer social networks, has demonstrated the use of centrality measures to quantify the influence, trust, social capital, and reputation of individual actors in a network (Bosu and Carver 2014; Asim et al. 2019). Specifically, centrality measures have been used in various social network-based trust models (Asim et al. 2019; Ceolin and Potenza 2017; Meo et al. 2017; Zahi and Hasson 2020; Şimşek and Meyerhenke 2020) where trust is defined as "a measure of confidence that an entity will behave in an expected manner, despite the lack of ability to monitor or control the environment in which it operates" (Sherchan et al. 2013; Singh and Bawa 2007). We hypothesize that a centrality-based rating system can help identify the developers whose code may be perceived as more trusted by the community and, therefore, be accepted by the downstream projects with less scrutiny.

> **RQ3:** Are developers with higher centrality ratings likely to have their code reviewed less carefully by the downstream project developers?

Towards RQ3, we survey the Rust developers in our constructed network to understand if the authors' and reviewers' identities impact how carefully they review upstream code. We designed our survey questions to evaluate if our proposed rating correlates with developers' perception of the trustworthiness of another developer, in terms of code contributed by them getting accepted by downstream project developers with less scrutiny.

Overall, we make the following contributions in this paper:

1. A social network-based rating of Rust developers derived from an aggregation of five centrality measures;

2. A survey-based evaluation of our rating approach in terms of a correlation between a developer's centrality rating and the level of review their code may face from the downstream project developers;

3. Empirical insights on the developer community structure of Rust ecosystem, highlighting the interconnected nature of developers from different packages;

4. A dataset on collaboration activity among developers in the Rust package ecosystem.

The Institutional Review Board has approved our study design (*IRB*). Our dataset will be published at `https://tinyurl.com/58bx2erd`. The rest of the paper is structured as follows: Section 6.2 explains our methodology to construct a social network of Rust

Table 6.1: An overview of collected data from October 2020 to October 2022 which we use to construct a social network of Rust developers.

| | |
|---|---|
| Crates | 1,644 |
| Repositories | 1,088 |
| Commits | 109,512 |
| Reviewed Commits | 57,592 (52.6%) |
| Rejected Pull Requests | 2,975 |
| Developers | 6,949 |
| Authors | 6,616 (95.2%) |
| Reviewers | 2,891 (41.6%) |
| Relationships | 26,448 |
| File Co-edition Relationships | 18,461 (69.8%) |
| Author-Reviewer Relationships | 14,363 (54.3%) |

developers and compute a centrality rating for each developer. We also present our survey questionnaire in this section, where we seek to understand if developers with higher centrality ratings are more trusted by the community. Afterward, Section 6.3, 6.4, and 6.5 present our findings for the three research questions in this study. Section 6.6 and 6.7 discusses the implications and limitations of our study before we conclude this paper in Section 6.8.

## 6.2 Methodology

The Rust ecosystem has an active culture of collaboration and use of open source packages (Schueller et al. 2022). Many security-critical projects, like popular blockchain networks, are being developed in Rust (Yakovenko 2018; Wood 2016). Further, the Rust ecosystem has built multiple tooling frameworks to help the secure use of its packages [5]. Therefore, we chose to study the Rust ecosystem in this paper.

In this section, we explain the methodology for building a developer social network for the Rust community and providing a rating for the developers in the network by aggregating five centrality measures. We also explain the developer survey we conducted to evaluate if our proposed rating reflects the community's perception.

[5]`https://github.com/rust-secure-code/wg`

Table 6.2: Centrality measures used in this study to rate developers in the Rust community

| Metric | Definition | Interpretation |
|---|---|---|
| Degree Centrality | The fraction of the total nodes in the network it is connected to. | The extent of collaboration with other developers indicates the developers' direct trust relationships in the network. |
| Closeness Centrality | The reciprocal of the average shortest path distance from the node to all other reachable nodes in the network. | How quickly any developer can reach another developer may estimate the developer's influence within the network. |
| Betweenness Centrality | The fraction of all pairs in the network whose shortest paths pass through that node. | The developers who keep the network connected can be perceived as more influential. |
| EigenVector Centrality | Eigenvector centrality computes the centrality of a node based on the centrality of its neighbors. | The collaborators of a trusted developer may also be trusted. |
| PageRank | PageRank computes a ranking of the nodes in the graph based on the structure of the number and quality of the incoming links. | PageRank is used to measure the influence or trustworthiness of a node in the network. |

### 6.2.1 DSN construction

In this subsection, we explain our data collection and social network construction methodology.

#### 6.2.1.1 Data collection

We obtained the metadata for packages from Crates.io from the official data dump [6]. At the time of this study, Crates.io hosted 92,231 packages. For this study, we chose the most 1,000 downloaded packages and all their dependency packages, which resulted in 1,724 packages. While ideally, the developer network should include collaboration history from all the Crates.io packages and all open source projects collaborated by the package developers, we chose to work on a subset of the packages as our data collection methodology is constrained by GitHub API rate limit. While we discuss the limitations of our chosen network boundary in

[6]https://crates.io/

Section 6.7, prior work has found DSN metrics to be robust even with incomplete data (Nia et al. 2010).

Out of the 1,724 packages, we found valid GitHub repositories for 1,644 packages. We restricted our study to only GitHub repositories, as we identify distinct developers through GitHub accounts and collect code review information from GitHub. These 1,644 packages are hosted on 1,088 distinct repositories. To build the developer network, we considered code activities over a two-year period between October 2020 and October 2022. Specifically, we collected 109,512 commits from this period.

For each commit, we collected the GitHub user account of the author. We consider each distinct GitHub user as an individual developer, excluding the bot accounts [7]. We also determined if each commit was reviewed or not by developers other than the author. We consider a commit to be code reviewed if there is a review approval on the associated pull request on GitHub or the commit was merged into the codebase by a different developer (Imtiaz and Williams 2022) We identified 52.6% (57,592) of the studied commits to have been code reviewed. We also collected data on the rejected pull requests. If a non-merged pull request has a review from or was closed by a different developer, we consider that as a collaboration between the author and reviewer. In total, we obtained 2,975 pull requests rejected by a reviewer.

Overall, we found 6,949 distinct developers in our data set who are authors or reviewers of the studied commits. Of these, 6,616 developers have authored at least one commit. Similarly, 2,891 developers have reviewed a commit at least once. We construct a social network over these 6,949 developers in the following subsection. Table 6.1 provides an overview of the study dataset.

#### 6.2.1.2 Developer social network (DSN) construction

A DSN is a graph data structure where the nodes represent developers, and the edges may represent communication and collaboration among them (Herbold et al. 2021). We focus on developer collaboration in this study with the rationale that collaboration can be mined directly from the repository history, unlike communication which can spread across various channels, such as emails and Slack. We use two metrics to capture developer collaborations (Kerzazi and El Asri 2016; Meneely and Williams 2011):

1. **File Co-edition Collaboration:** If two developers work on the same file within a 30-day window, we consider that activity to indicate collaboration between them. This

[7] the bot accounts' usernames are suffixed by '[bot]' on GitHub

metric was validated by Meneely et al. (Meneely and Williams 2011).

2. **Author-Reviewer Collaboration:** If a developer has reviewed code changes submitted by another, we consider that activity to indicate collaboration between the author and the reviewer.

We identified 166,675 instances where two developers had made changes to the same file within a 30-day period, involving a relationship between 18,461 developer pairs. We also identified 62,320 cases where a developer had reviewed code from another developer, involving 14,363 relationships. Combining both types of relationships (the same developer pair can have both types of relationships), we have edges between distinct 28,448 developer pairs. Further, we set edge weights in the network as the number of collaborations following prior work (Kerzazi and El Asri 2016; Meneely and Williams 2011). Finally, we constructed an undirected, weighted graph based on our collected data using the Python package, *networkx* (Hagberg and Conway 2020).

### 6.2.2 Centrality measures for developer rating

Centrality measures are a common approach to estimating subjective concepts like trust, reputation, and influence (Bosu et al. 2014; Asim et al. 2019; Ceolin and Potenza 2017; Meo et al. 2017; Zahi and Hasson 2020; Şimşek and Meyerhenke 2020). Overall, centrality measures indicate who are the central actors in a network and, thus, can be used to rank the actors within the network (Riquelme et al. 2018; Cadini et al. 2009). In DSN literature, Bosu et al. (Bosu and Carver 2014) have studied the impact of developer reputation on code review outcomes by identifying the core developers in a network through six centrality measures. We incorporate five of those six metrics in our work, as these were also used in a trust model proposed by Asim et al. (Asim et al. 2019). One metric we excluded from Bosu et al.'s (Bosu and Carver 2014) work is eccentricity, which only works for graphs with a single connected component, and is unsuitable in our context. Table 6.2 explains the five metrics and provides our rationale behind using them in our study context.

We compute the five metrics using *networkx* package that considers the weights of the edges. We then aggregate them to avoid bias from any single measure (Bosu et al. 2014; Şimşek and Meyerhenke 2020; Sabah and Şimşek 2023; Asim et al. 2019; Fei et al. 2017). We take the normalized value of each of the five centrality measures listed in Table 6.2, then calculate the normalized sum of those values to get a final rating between 0 and 1. All five centrality metrics are ordinal attributes, so their aggregated rating will also be ordinal. Therefore, we use the rating to rank all the developers in our constructed network, where a higher rating indicates higher centrality for a developer. Our approach of a simple addition of normalized scores is commonly followed when combining multiple metrics for ranking (Tofallis 2014; Shabrina et al. 2022; Cody et al. 2018).

### 6.2.3 Developer survey

To evaluate our ranking approach (RQ3), we want to understand if developers with higher centrality ratings in our constructed network are perceived to be more trusted by the community in terms of their code facing less scrutiny from the downstream projects. To that end, we conduct a developer survey to (i) understand if authors' and reviewers' identities impact how carefully the respondents review upstream code, and (ii) collect data on the level of scrutiny the respondents would put in reviewing upstream code coming from different developers in the network. Finally, we use the data indicating the level of scrutiny assigned to different developers as a proxy measure of trust and conduct a correlation test with the developers' centrality ratings. We sent the survey to a subset of the developers in the network who have directly collaborated with at least five others in the network, with the rationale that the developers who only had a few interactions in the community may not be able to provide informed answers about other developers.

Our survey contained three required questions. Table 6.3 and Figure 6.1, 6.2 and 6.3 show the survey questions and corresponding Likert-scale options. We detail each of the questions below:

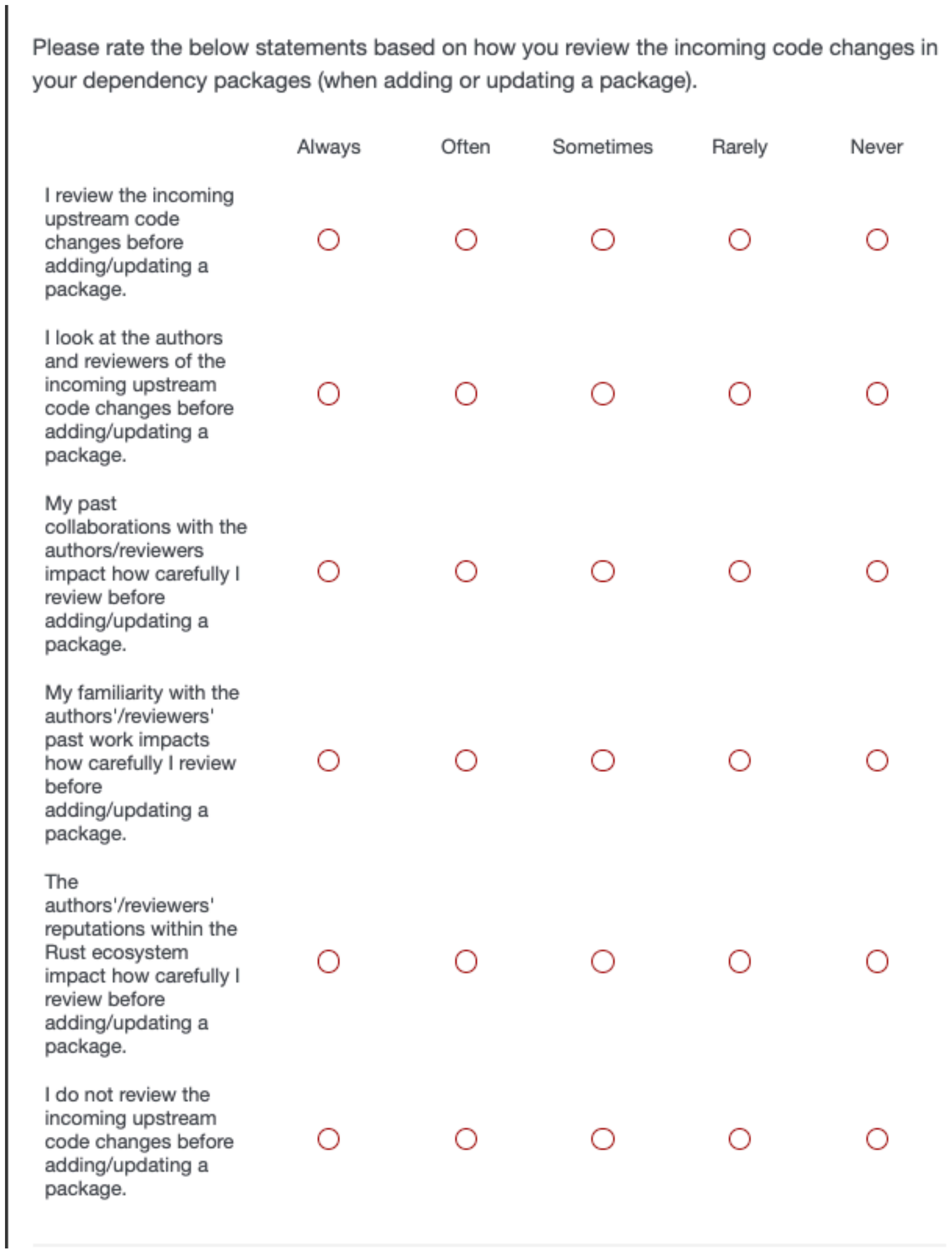
**Q(a)** The motivation behind Q(a) is first to understand how often developers review upstream code changes before merging them into their codebase. We also want to know if the authors' and reviewers' identities impact their review process. For example, we ask if the authors' and reviewers' reputations within the Rust community and the recipients' familiarity with their work impact their review process. Q(a) helps us to establish the motivation that a ranking system for Rust developers will be helpful for the downstream project developers to review upstream code.

**Q(b)** The motivation behind Q(b) is to understand if our constructed network reflects the true collaboration relationships between the studied developers. The responses of Q(b) help us validate the network we used to calculate the centrality measures and ratings for each developer. To this end, we sample ten developers from the network independently for each respondent and ask them how well they know these developers.

We sample ten developers for each respondent in the following ways: (i) five developers

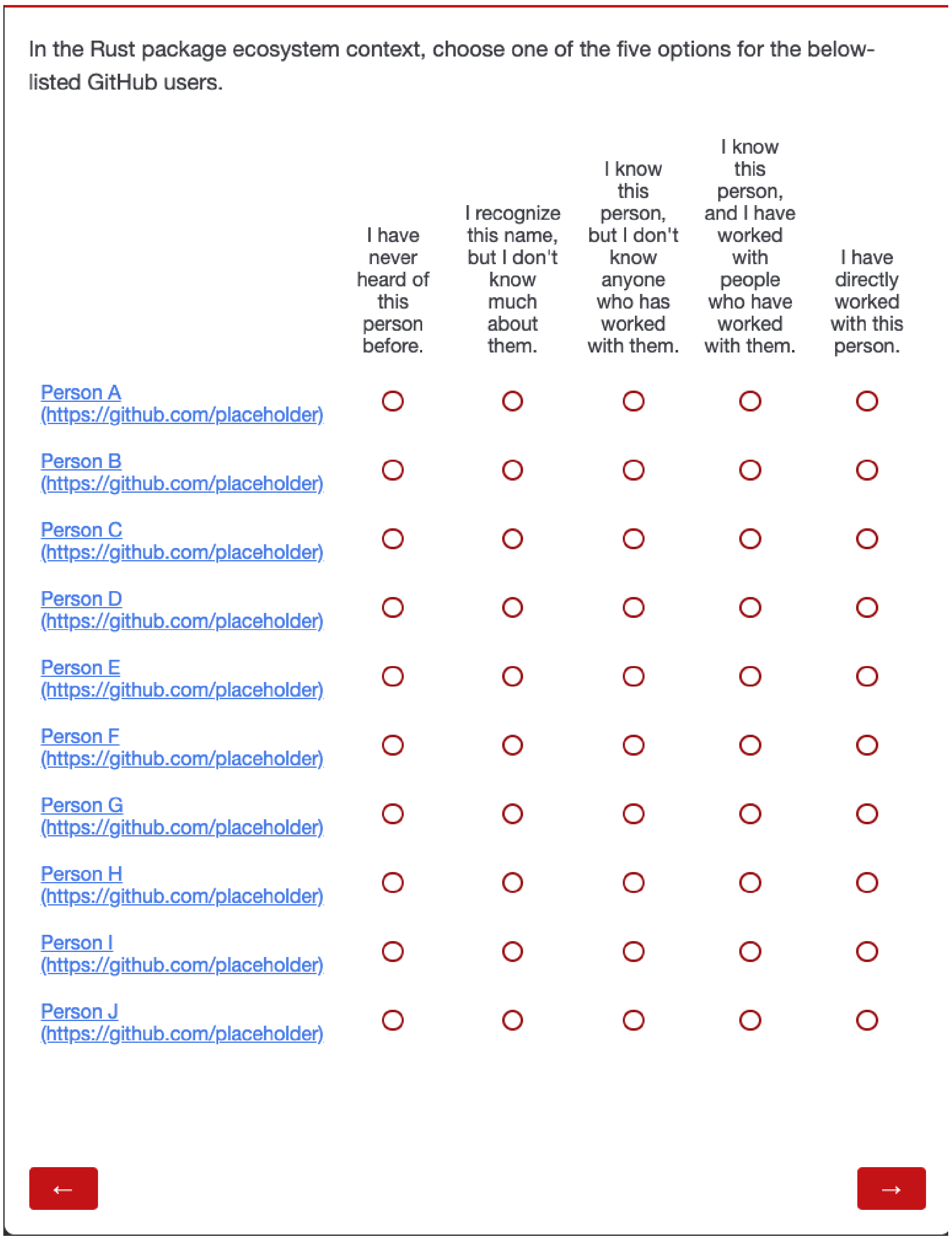
Table 6.3: Survey Questions

| Questions | | Options |
|---|---|---|
| **Q(a)** | Please rate the below statements based on how you review the incoming upstream code changes in your dependency packages (when adding or updating a package).<br>— The question presents six statements which are listed in Figure 6.7 when discussing the findings. | Always, Often, Sometimes, Rarely, Never |
| **Q(b)** | In the Rust package ecosystem context, choose one of the five options for the below-listed GitHub users.<br>— The respondent is presented with a list of ten GitHub users who are sampled separately for each respondent. The sampling strategy is explained in Section 6.2.3. | 1. I have never heard of this person before.<br>2. I recognize this name, but I don't know much about them.<br>3. I know this person, but I don't know anyone who has worked with them.<br>4. I know this person, and I have worked with people who have worked with them.<br>5. I have directly worked with this person. |
| **Q(c)** | When adding or updating a package, how carefully would you review upstream code changes coming from the below-listed GitHub users? Please choose one of the four options based on your knowledge/association with them. Note that the code changes are within the dependency packages of your project. For example, when updating a package, how do you review the new changes that come with the update?<br>— The respondent is presented with the same list of ten GitHub users from Q(b). | 1. I would not include code from this person in my project.<br>2. I would review each line of upstream code change coming from this person.<br>3. I would skim through the upstream code changes coming from this person.<br>4. I do not feel the need to review the upstream code changes coming from this person. |


Please rate the below statements based on how you review the incoming code changes in your dependency packages (when adding or updating a package).

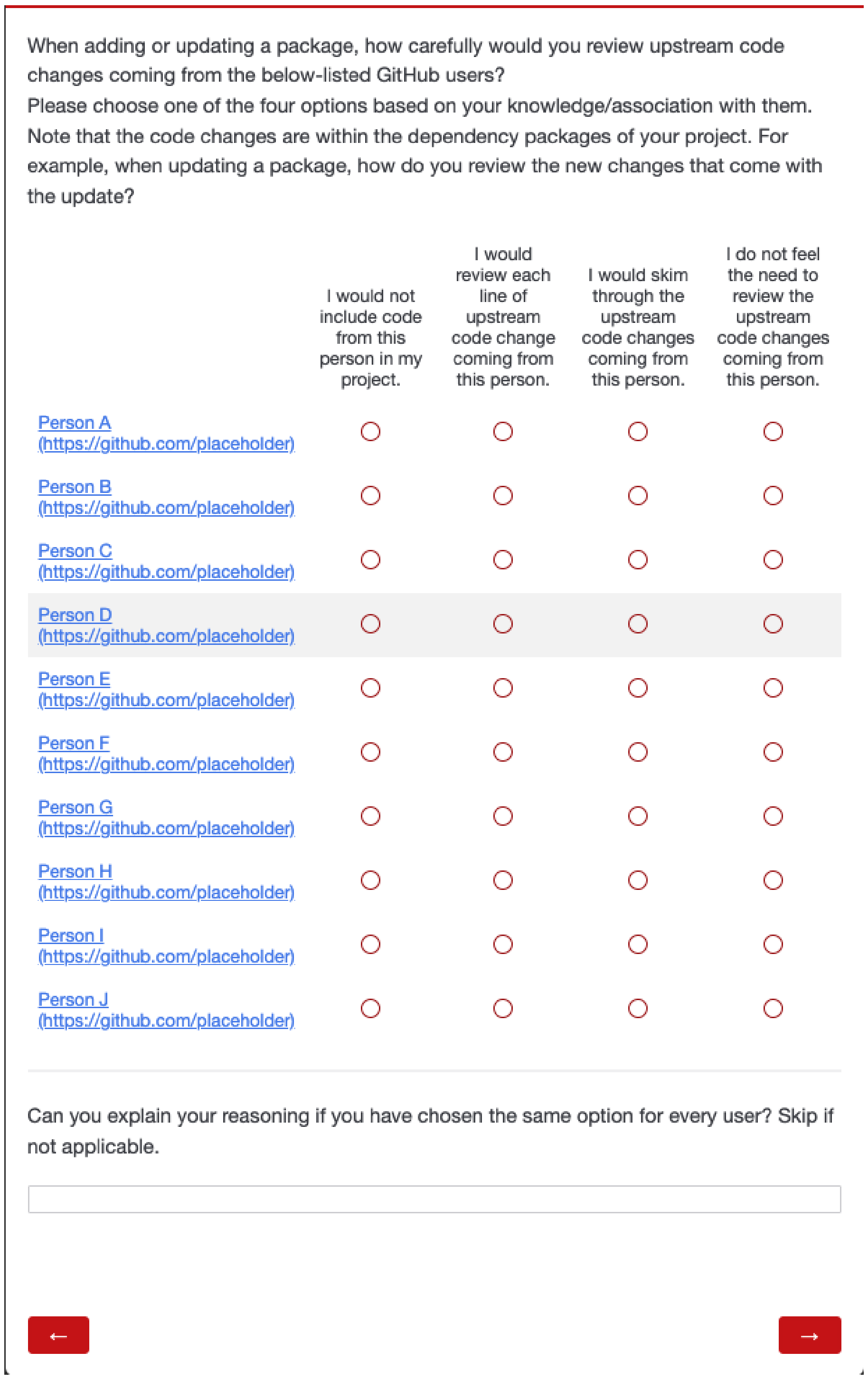
| | Always | Often | Sometimes | Rarely | Never |
|---|---|---|---|---|---|
| I review the incoming upstream code changes before adding/updating a package. | ○ | ○ | ○ | ○ | ○ |
| I look at the authors and reviewers of the incoming upstream code changes before adding/updating a package. | ○ | ○ | ○ | ○ | ○ |
| My past collaborations with the authors/reviewers impact how carefully I review before adding/updating a package. | ○ | ○ | ○ | ○ | ○ |
| My familiarity with the authors'/reviewers' past work impacts how carefully I review before adding/updating a package. | ○ | ○ | ○ | ○ | ○ |
| The authors'/reviewers' reputations within the Rust ecosystem impact how carefully I review before adding/updating a package. | ○ | ○ | ○ | ○ | ○ |
| I do not review the incoming upstream code changes before adding/updating a package. | ○ | ○ | ○ | ○ | ○ |

Figure 6.1: Survey Question - Q(a)


In the Rust package ecosystem context, choose one of the five options for the below-listed GitHub users.

| | I have never heard of this person before. | I recognize this name, but I don't know much about them. | I know this person, but I don't know anyone who has worked with them. | I know this person, and I have worked with people who have worked with them. | I have directly worked with this person. |
|---|---|---|---|---|---|
| Person A (https://github.com/placeholder) | ○ | ○ | ○ | ○ | ○ |
| Person B (https://github.com/placeholder) | ○ | ○ | ○ | ○ | ○ |
| Person C (https://github.com/placeholder) | ○ | ○ | ○ | ○ | ○ |
| Person D (https://github.com/placeholder) | ○ | ○ | ○ | ○ | ○ |
| Person E (https://github.com/placeholder) | ○ | ○ | ○ | ○ | ○ |
| Person F (https://github.com/placeholder) | ○ | ○ | ○ | ○ | ○ |
| Person G (https://github.com/placeholder) | ○ | ○ | ○ | ○ | ○ |
| Person H (https://github.com/placeholder) | ○ | ○ | ○ | ○ | ○ |
| Person I (https://github.com/placeholder) | ○ | ○ | ○ | ○ | ○ |
| Person J (https://github.com/placeholder) | ○ | ○ | ○ | ○ | ○ |

Figure 6.2: Survey Question - Q(b)



When adding or updating a package, how carefully would you review upstream code changes coming from the below-listed GitHub users?
Please choose one of the four options based on your knowledge/association with them.
Note that the code changes are within the dependency packages of your project. For example, when updating a package, how do you review the new changes that come with the update?

| | I would not include code from this person in my project. | I would review each line of upstream code change coming from this person. | I would skim through the upstream code changes coming from this person. | I do not feel the need to review the upstream code changes coming from this person. |
|---|---|---|---|---|
| Person A (https://github.com/placeholder) | ○ | ○ | ○ | ○ |
| Person B (https://github.com/placeholder) | ○ | ○ | ○ | ○ |
| Person C (https://github.com/placeholder) | ○ | ○ | ○ | ○ |
| Person D (https://github.com/placeholder) | ○ | ○ | ○ | ○ |
| Person E (https://github.com/placeholder) | ○ | ○ | ○ | ○ |
| Person F (https://github.com/placeholder) | ○ | ○ | ○ | ○ |
| Person G (https://github.com/placeholder) | ○ | ○ | ○ | ○ |
| Person H (https://github.com/placeholder) | ○ | ○ | ○ | ○ |
| Person I (https://github.com/placeholder) | ○ | ○ | ○ | ○ |
| Person J (https://github.com/placeholder) | ○ | ○ | ○ | ○ |

Can you explain your reasoning if you have chosen the same option for every user? Skip if not applicable.

Figure 6.3: Survey Question - Q(c)

Table 6.4: Structural overview of the social network of Rust developers

| | |
|---|---:|
| Nodes | 6,949 |
| Edges | 26,448 |
| Components | 132 |
| Isolates | 111 |
| Network density | 0.001 |
| Avg. clustering coefficient | 0.398 |
| Total communities | 164 |
| Communities with more than a hundred nodes | 21 |
| **Largest Component** | |
| Nodes | 6,789 (97.7%) |
| Edges | 26,417 |
| Avg. shortest path length | 3.674 |

from the direct collaborators of the recipient as indicated by an edge in our constructed network, and (ii) five developers from the rest of the network with whom the recipient does not have an edge with. Within these two groups (direct collaborators and others), we sample three from the top 50 according to the ranking we derive from centrality measures described in Section 6.2.2, and two from the rest of that group. We chose this sampling approach, so the participants have a higher chance of being familiar with at least one of the developers. Further, this approach ensures that the survey recipients have top, average, and low-ranked developers to provide their opinion.

**Q(c):** The motivation behind Q(c) is to evaluate if our proposed ranking approach reflects developer perception in the community. We present the survey recipient with the same ten developers from Q(b) and ask how the recipient would review upstream code changes coming from them. We provide four Likert-scale options to indicate the level of scrutiny the recipient would put into upstream code from each developer. Using the responses to Q(c), we statistically test if the recipients' answers correlate with the developers' centrality rating.

## 6.3 RQ1 Findings

**To what extent are developers from the most downloaded Rust packages interconnected through collaboration?** Our constructed network graph consists of 6,949 nodes (developers) and 26,448 edges (collaborations). The largest connected component[8] in the network

[8]A connected component in a graph is a group of nodes where every two nodes have a direct or indirect path (via other nodes) between them.

consists of 6,789 nodes (97.7%), showing the majority of the studied developers are interconnected. The rest of the developers are isolated or connected within a small group of collaborators. The density [9] of our network is 0.001, and the average clustering coefficient [10] of the nodes is 0.4, which are typical of real-world social networks (Walsh et al. 1999). The low density and clustering coefficient values indicate the graph consists of many tightly-knit clusters.

Further, the average shortest path length in the largest connected component is 3.7 meaning any two developers are connected via approximately 4 other developers, on average, indicating a small-world phenomenon (Walsh et al. 1999; Sherchan et al. 2013). Further, we found 164 communities [11] using the Louvain community detection algorithm (Que et al. 2015). However, most developers are members of 21 communities that consist of at least 100 developers. The community structure further establishes the interconnected nature of the Rust developer network. Table 6.4 summarizes the properties of the network. Figure 6.4 visualizes the network, where the nodes in the communities with larger than 100 developers are assigned distinct colors.

Overall, this RQ reveals the properties of our constructed Rust developer network. We find Rust developers to be interconnected within tightly-knit clusters. The network shows characteristics of a small world, which establishes the ground for centrality analysis (Watts and Strogatz 1998; Wasserman and Faust 1994), presented in the following Section 6.4.

## 6.4 RQ2 Findings

**What is the distribution of the developer ratings based on their network centrality?** As explained in Section 6.2.2, we aggregate five centrality measures to provide a single rating to each developer in the network. Figure 6.5 shows a histogram of the aggregated rating for the developers in our network. The histogram shows a skewed distribution of the aggregated centrality ratings. We find 5.3% of the developers to have a rating above 0.2, indicating their core positions in the network. Conversely, 2.6% of the developers have a rating below 0.1, indicating their peripheral position. The ratings for the rest of the 92.1% of the developers are distributed between 0.1 and 0.2. While the aggregated rating offers a granular ordering among the developers, the skewed distribution can be visually interpreted

[9]The ratio of existing edges to all possible edges.

[10]The clustering coefficient of a node is a measure of the extent to which its neighbors are also connected to each other.

[11]A subset of nodes within the graph such that connections between the nodes are denser than connections with the rest of the network.

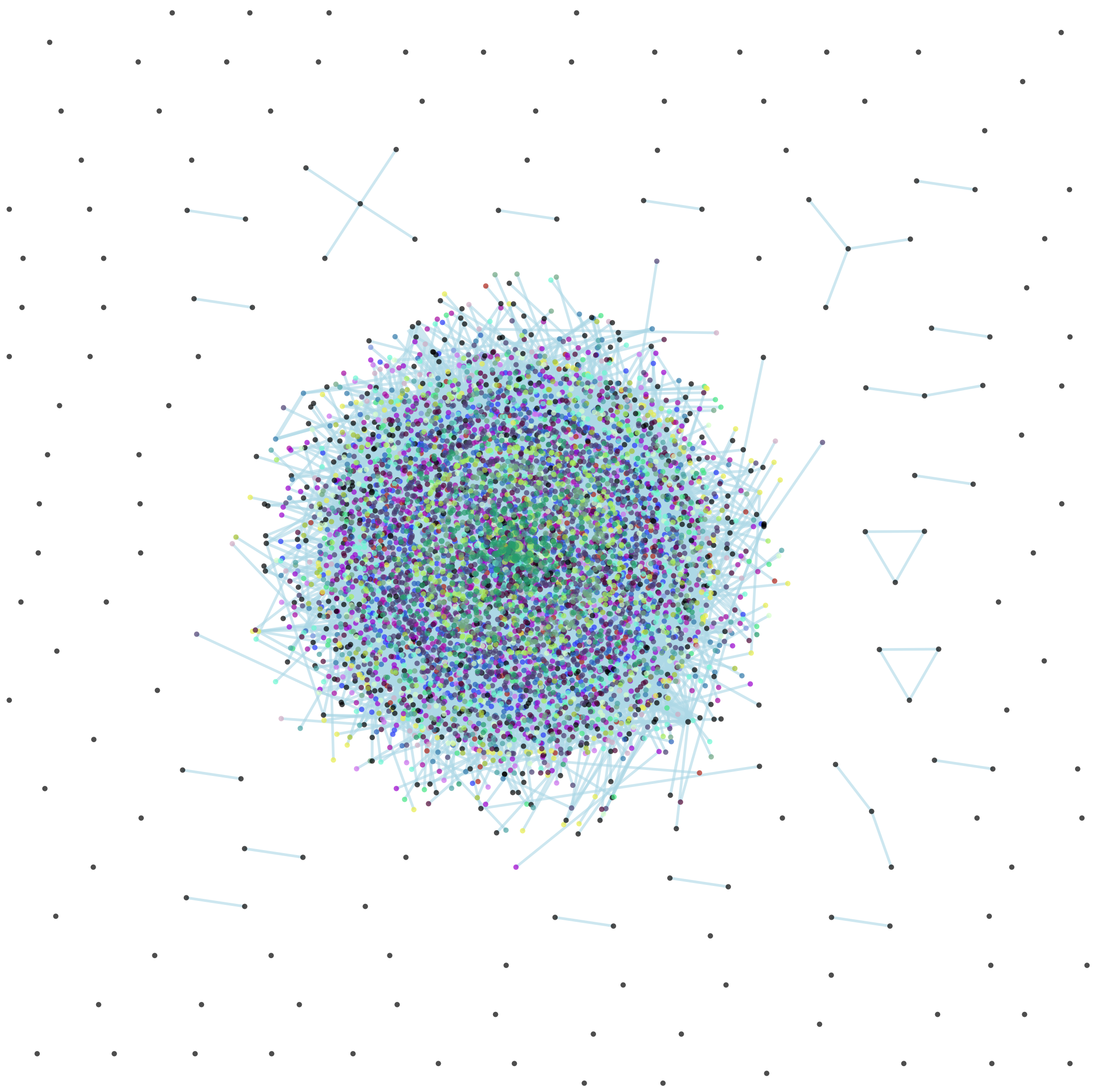


Figure 6.4: Social network graph of Rust developers

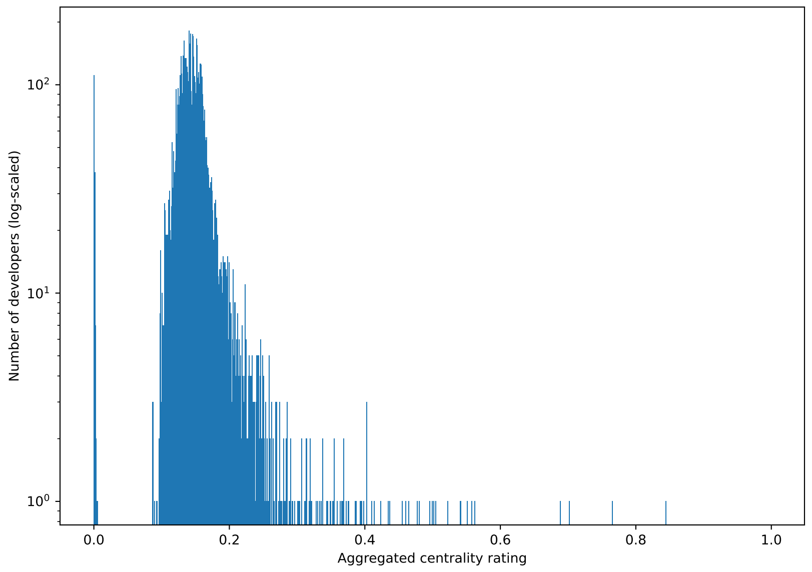
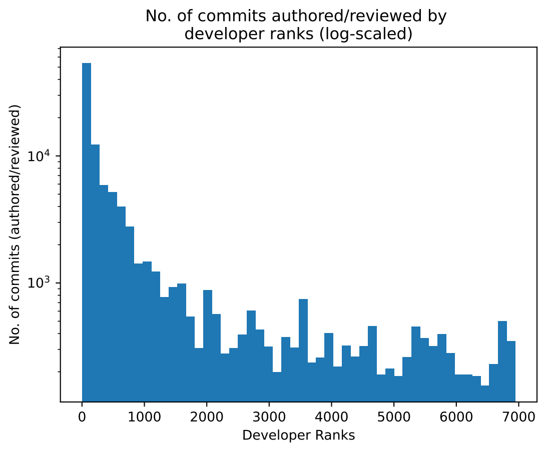


Figure 6.5: Histogram of aggregated centrality rating of Rust developers

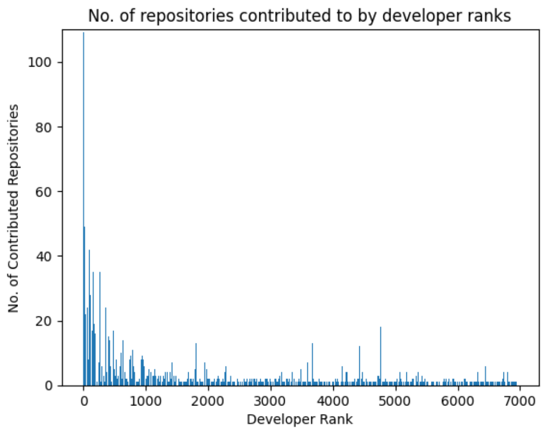
to categorize them into high (>0.2), low (<0.1), and average (0.1-0.2) ranks based on their network centrality.

We investigated developers' characteristics in terms of the relationship between their centrality ratings and (a) the number of commits they contributed either as an author or a reviewer; (b) the number of different repositories they contributed to; and (c) the number of developers they directly collaborated with. Figure 6.6 shows a bar chart on each characteristic across developers from different centrality ratings (in the x-axis, developers are sorted in descending order based on their centrality ratings, i.e., ranked first to last, where the first ranked developer has the highest centrality.). Our results show that developers with high centrality ratings are also the ones to participate (as authors or reviewers) in the most number of commits and repositories and have collaboration with the most number of developers in the network.

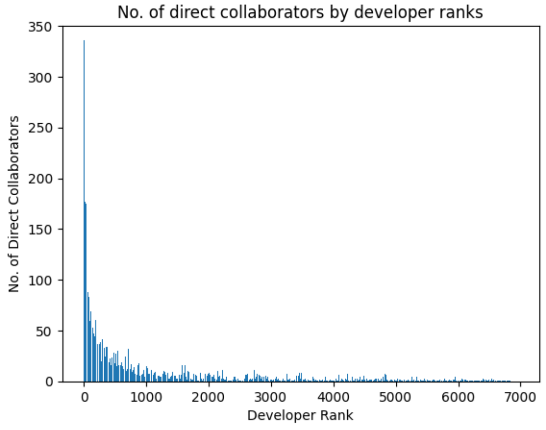
When looking at the commits specifically in Figure 6.6a, our dataset has 51,154 unreviewed commits from 883 distinct authors. Our results show that the top 10% of developers based on network centrality have authored 78.2% of these unreviewed commits. For 57,592 reviewed commits (52.6% of the total), more than one developer is involved in the commit. Considering both authors and reviewers of these commits, the top 10% developers were



(a)



(b)



(c)

Figure 6.6: Bar chart for developer rank vs. (a) no. of commits contributed either as an author or a reviewer, (b) no. of repositories the developers contributed to, and (c) no. of direct collaborators. In the x-axis, developers are sorted in descending order based on their centrality ratings, i.e., ranked first to last, where the first-ranked developer has the highest centrality.

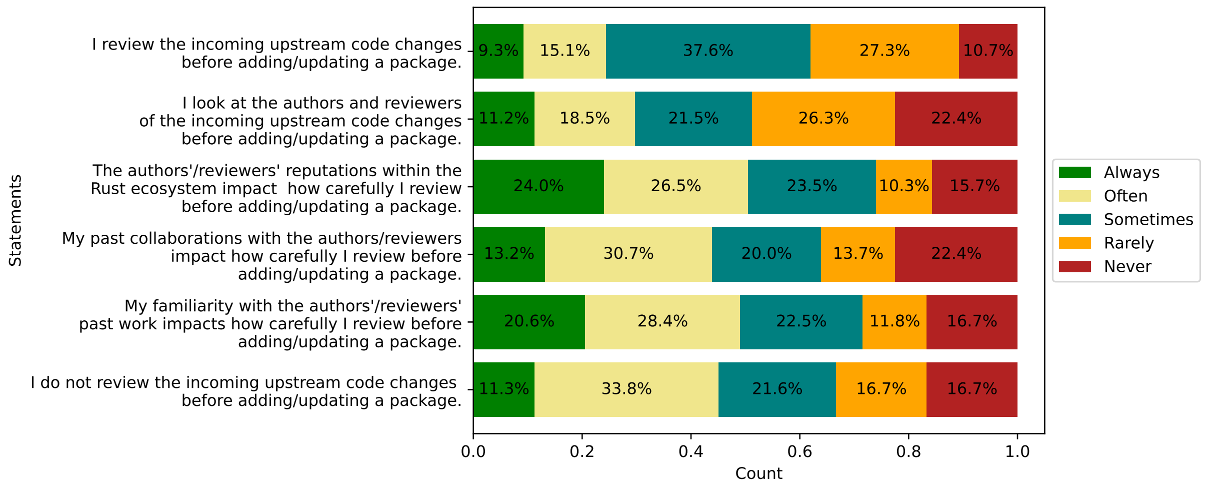


Figure 6.7: Likert scale responses on how developer identity impacts review process of upstream code.

involved in 89.7% of these reviewed commits. Overall, our results show that the top 10% of the developers were involved in 84.2% of the commits in our dataset, either as an author or a reviewer. However, note that if we scale our social network to all 92K packages hosted on Crates.io, a similar pattern between the centrality ratings and the number of commits authored/reviewed may not be found. If a developer has published many packages that are not widely used and has little collaboration with others in the network, that developer will receive a low centrality rating despite having a high code contribution.

## 6.5 RQ3 Findings

**Are developers with higher centrality ratings likely to have their code reviewed less carefully by the downstream project developers?** We investigate RQ3 through responses from our developer survey, consisting of three questions in Table 6.3. The questions help us to investigate if (i) authors' and reviewers' identities impact how carefully developers review upstream code; (ii) if our constructed DSN accurately captures the collaboration relationship between developers; and (iii) if developers with higher centrality ratings are perceived to be more trusted by the community, in terms of how much their code can get scrutinized by the downstream developers.

We sent our survey to 1,995 developers from our network who collaborated with at least five others according to our collected data. We received 206 responses, with a response rate of 10.3%. We present findings from the responses in this section.

### 6.5.1 Survey responses - Q(a)

In Q(a), we investigate whether the authors' and reviewers' identities impact how carefully downstream project developers review upstream code changes (i.e., review scrutiny) through a Likert-scale matrix question of six statements. Figure 6.7 lists the six statements and the response distribution.

The survey participants have responded that they are unlikely to review the upstream code changes before merging them into their projects. Only 24.4% of the respondents answered that they (always or often) review the upstream changes. Similarly, only 27.7% of the respondents answered that they (always or often) look at the authors/reviewers of the upstream changes before adding/updating a package. In the survey, we also gave options to provide additional comments. In these open-ended responses, the respondents indicated that they are most likely to review a package before adding it for the first time and may not review the changes in subsequent updates. Further, respondents have indicated they may only read the *changelog* or *release notes* [12] during an update and not the actual code changes. The respondents have also mentioned that the context of their projects, e.g., how security-critical the project is, and the nature of the upstream package, e.g., if the package code is internet-facing, impact if they review the changes in a new update.

While the survey respondents are less likely to review all upstream code changes, they indicated that the authors' and reviewers' reputations in the Rust community impact their decision process before adding or updating a package. **More than half of the respondents (50.5%) suggested that the reputations of the upstream developers impact (always or often) how carefully they review upstream changes.** In contrast, only 25.9% of the respondents indicated that the developer's reputation rarely or never impacts their review process. Similarly, respondents indicated that familiarity with the upstream developers' work (49.0% of the responses were always or often) and past collaborations (43.9% of the responses were always or often) impact their review process.

### 6.5.2 Survey responses - Q(b)

An objective behind asking this question was to investigate if our constructed social network accurately reflects developer collaborations in real life. Considering ten responses for ten

[12] a natural text documentation of the changes in a new version

given developers from each respondent in Q(b), we have 2,060 total data points. In 989 of these cases, our constructed DSN has an edge between the respondent and the given developer. However, only in 185 of these cases (18.7%), the respondent answered that they directly worked with the given developer. In another 424 cases (42.9%), the respondent indicated some familiarity with the given developers (any option other than *I have never heard of this person before*). In the remaining 380 cases (38.4%), the respondent stated they had never heard of the person before.

While the file co-edition relationship may not ensure the two developers are, indeed, familiar with each other, we find that the survey respondents failed to recognize developers in the case of author-reviewer relationships as well. Out of the 989 cases, there are 531 cases where our DSN shows an author-reviewer relationship between the recipient and the given developer. However, only in 164 (30.9%) of these cases, the respondents indicated a direct collaboration, while in another 328 (61.8%) cases, the respondents showed some familiarity. In the remaining 39 cases (7.3%), the respondent stated they had never heard of the person before, even though our data shows an author-reviewer relationship between them. Conversely, there are 1,071 cases where our constructed DSN does not show an edge between the survey respondent and the given developer. In only 23 cases (2.1%), however, the respondent indicated they had worked with the developer.

Given the distributed nature of open-source development, a developer may not be familiar with all others working on the same project at the same time. Further, a developer may not remember every collaborator's name as well. Overall, **the survey responses show that our constructed DSN may overestimate collaboration relationships between two developers.**

‘

### 6.5.3 Survey responses - Q(c)

We hypothesize that developers with higher centrality ratings are more trusted in the community in terms of their code facing lesser scrutiny from the downstream project developers. We test our hypothesis through responses for Q(c). Out of the 206 respondents, 10 did not fill out this question. We present our analysis for the remaining 196 responses in this subsection.

**We find that 118 respondents (60.2%) chose the same option for all the developers. On the other hand, 78 respondents chose different options for different developers, indicating that the developer identity may impact the level of review they employ for upstream**

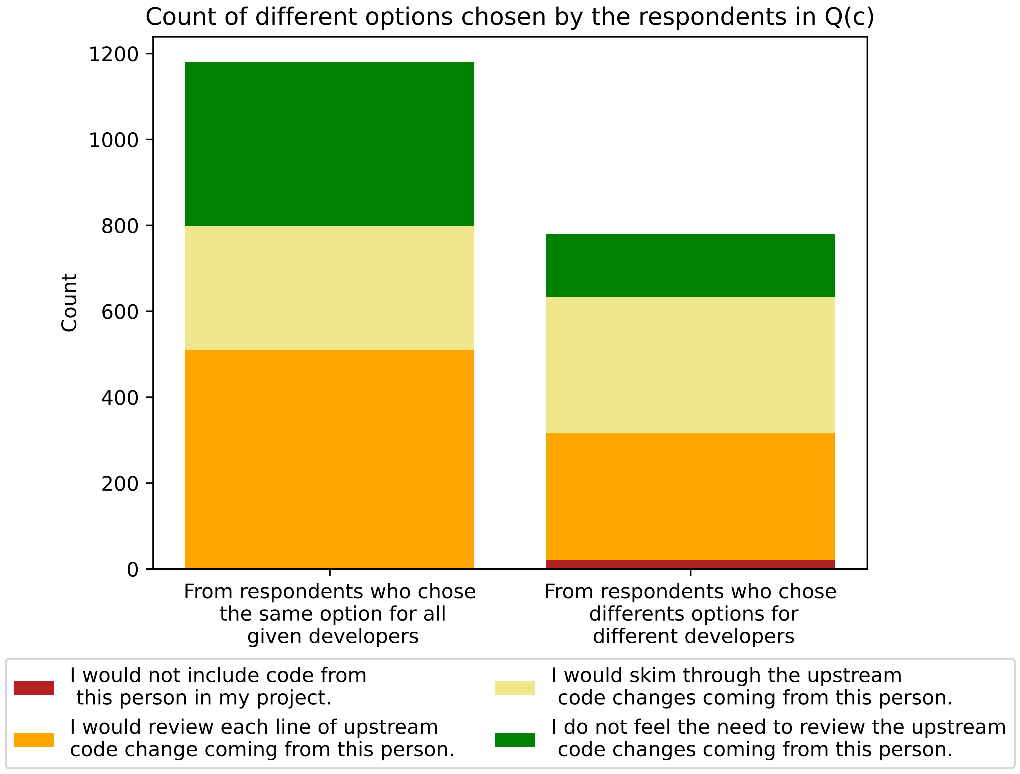


Figure 6.8: Count of different options chosen by the respondents in Q(c)

**changes.** Figure 6.8 shows the response distribution from both types of respondents.

In our survey, we asked the respondents to provide an optional explanation if they chose the same option for every developer in Q(c). We received an explanation from 97 respondents. In 30 of these cases, the respondents appeared to misinterpret our question and answered based on how they would review *pull requests* in their own projects, rather than changes in upstream code. Consequently, these respondents selected that they would review each line of changes. The rest of the respondents either mentioned that they do not review upstream dependency code (26 respondents), or are not familiar enough with the given developers (5 respondents). The rest of the 27 respondents noted they would employ a similar level of review, irrespective of who they are coming from, and may decide the level of review based on other factors other than the developer identity, such as popularity and API changes.

For the 78 respondents who chose a different level of review for different developers in Q(c), we measured if a developer's aggregated centrality rating correlates with the level of the review indicated by the respondents in Q(c), with 1 being the highest scrutiny and 4 being the least scrutiny. The corresponding Likert-scale options for each level of review are listed in Table 6.3.

In our analysis, we control for the fact that there can be multiple responses for the same developer coming from different respondents. To this end, we run a mixed effect regression analysis (Faraway 2016) on the 78 responses, where the respondent chose different levels of review for different developers. Mixed effect linear regression (MELR) models the association between some fixed effect variables and dependent variables when clusters or categorizations are present in the data. The regression model eliminates the impact of the clusters (that are called random effects) and represents only the correlation between the fixed effect and the dependent variables. In our model, we define the level of review indicated by the respondents for a given developer as the dependent variable and the aggregated centrality rating of the developer as the fixed effect variable. We add the developer identity as a random effect variable in the regression analysis since multiple responses for the same developer can introduce clusters in the data (780 responses are distributed over 484 distinct developers).

The p-value obtained from the MELR model shows a significant correlation between the aggregated centrality rating of a developer and the level of review or scrutiny their code may face from the downstream project developers (N=780, No. of Groups = 484, Coef. = 1.033, $p \approx 0.0$) [13] — that is—**developers with higher centrality ratings are likely to be subject to less scrutiny from the downstream project developers.** We have also visualized the correlation in Figure 6.9 through a box plot of the distributions of the developers' centrality rating under each level of review, as indicated by the survey respondents. The box plots show that as the median of centrality rating decreased, the level of review scrutiny increased (from level 4 to level 1). The box plot also shows that the developers whose code the respondents chose to accept without a review (note the box plot for level 4) have a higher median centrality rating than the developers for whose code the respondents warranted comparatively more scrutiny.

Further, we investigated if the level of familiarity of the respondent with a given developer, as indicated by the responses in Q(b), correlates with the level of review or not. However, we do not find any significant correlation ($p = 0.843$) in the MELR model. We also did not find the past collaboration of the respondent with the given developer to be an influential factor ($p = 0.682$). Our findings indicate that centrality ratings computed from the social network can predict how carefully downstream projects review upstream

[13]We also find a significant correlation when considering responses from all 196 respondents (N=1960, No. of Groups = 991, Coef. = 0.524, $p \approx 0.0$).

changes coming from a developer and, therefore, can be used as a proxy for the trust rating of a Rust developer within the community.

## 6.6 Discussion

In this section, we discuss the implications of our findings:

**Developer social network (DSN) at a package ecosystem level.** Prior work on DSN only focused on single large-scale projects like Linux and Firefox. In the context of a single project, a rich body of prior work showed how DSN could be leveraged for many use cases, including identifying core and peripheral developers (Bosu et al. 2014), predicting defects (Abreu and Premraj 2009) and post-release failures (Meneely et al. 2008). Ours is the first work to show that developers at a package ecosystem level are also interconnected. However, we only studied a limited number of packages to construct a DSN. How the DSN structure may evolve when scaled to more packages is unknown. Similarly, popular packages from other ecosystems, like npm and PyPI, may not exhibit similar inter-connectedness as the Rust ecosystem. While our study shows promise in studying DSN at a package ecosystem level, future studies are required to understand how well the approach scales or generalizes to other ecosystems.

**The use case of network-based centrality rating towards an identity model for open source developers.** Prior research has proposed building trust in upstream code by tracing the humans behind that code (Imtiaz and Williams 2022). However, in the open source, anybody can open an account on GitHub and start contributing. Therefore, only knowing which account on an online coding platform has authored/reviewed certain code may not be sufficient for downstream project developers to trust that code. In that case, the network-based centrality rating may be used towards the development of an identity model for open source developers. With this rating approach, one can determine not only who has authored/reviewed code in a specific package, but also estimate how reputed the developer is in the community.

Such a rating system may also help establish trust in post-release code vetting (Zimmermann et al. 2019; Imtiaz and Williams 2022). Tools like *cargo-crev* [14] offer a distributed code review system for developers to vet published packages so that not all downstream projects have to review the package themselves. However, establishing trust in the developers who will perform the code vetting on behalf of the entire community is also important. Our

[14] `https://github.com/crev-dev/cargo-crev`

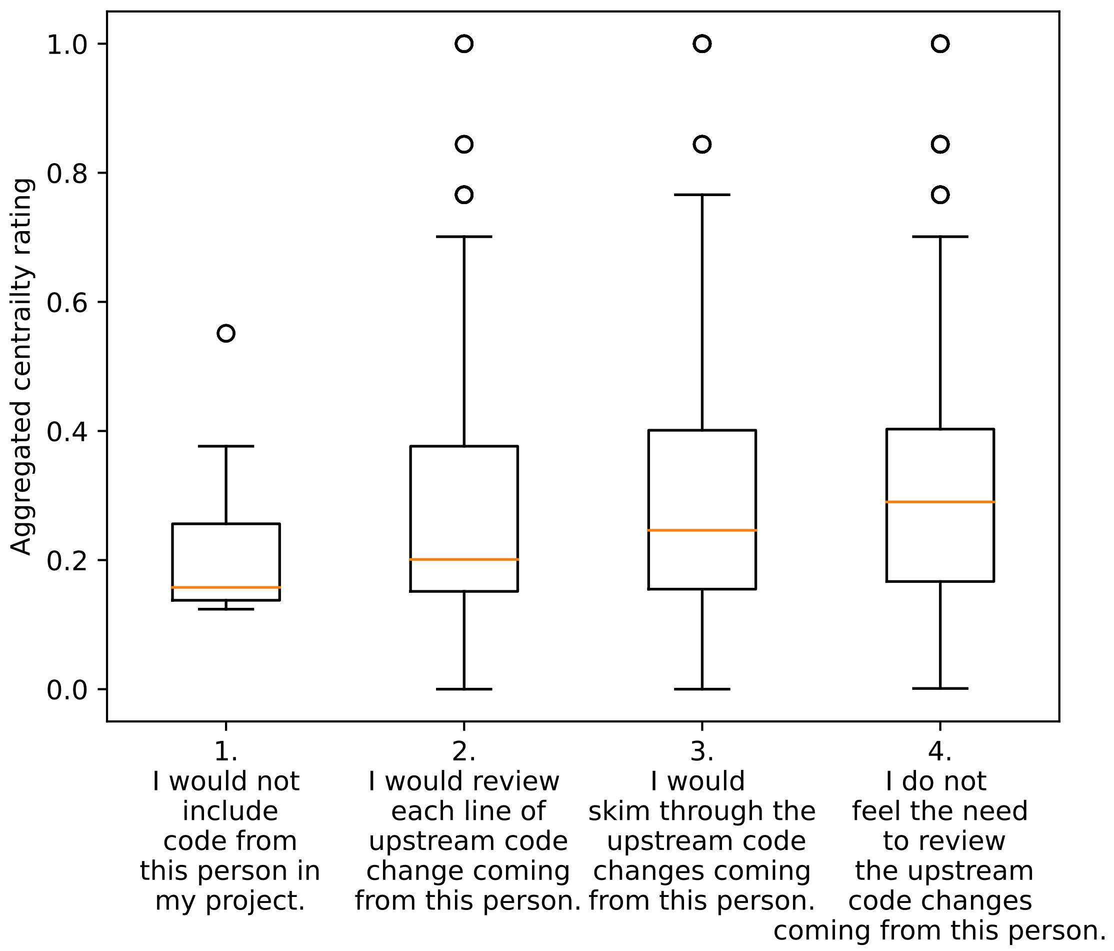


Figure 6.9: Box plot of aggregated centrality ratings for developers across different levels of review chosen by respondents ($N = 78$) in Q(c), with 1 being the highest scrutiny and 4 being the least scrutiny.

social network-based centrality rating system can be helpful in these cases to identify the most reputed developers in a community and make crowd-based code vetting objectively trustworthy.

**Package managers and audit tools can incorporate developer centrality ratings in their reporting of package metadata.** Package managers like Crates.io and audit tools like *OSSF Scorecard*[15], *packj*[16], and *Depdive* (Imtiaz and Williams 2022) provides various analysis over a package and its updates to help downstream project developers be aware of potential security, quality, and maintainability risks. For example, these tools provide the download count, code review coverage, and the maintainers' email domain validity to help developers review a new update. These tools can also incorporate centrality ratings of the authors and reviewers of upstream changes, as our work has shown such a rating can be a helpful signal.

Developers' centrality ratings can be leveraged to identify potentially malicious activities. For example, code review coverage proposed by Imtiaz and Williams (Imtiaz and Williams 2022) can be gamed by opening a fake account to stamp a review on malicious code pushed by a compromised developer account. Further, a group of developers may be colluding, in which case code review coverage would make little sense. Developer centrality ratings and their collaboration patterns may be leveraged to identify such potentially malicious activities and prevent future supply chain attacks (Ohm et al. 2020).

**Can we measure trust at an organizational level?** One respondent in our survey noted that measuring trust at the individual developer level, as has been done in this paper, may not scale. One alternate approach to rating individual developers is to rate organizations in the community. Many open source developers contribute on behalf of large organizations, such as Mozilla, Facebook, and Google. We can leverage social network analysis to identify the top organizations in a community, similar to how we identified the top developers. Future studies may build on our work in this paper and investigate if trust based on an organizational level may be appropriate at larger scales.

**Correlation between five centrality measures that are aggregated to provide developer ratings:** We aggregate five centrality metrics to provide a single rating for each developer in the network. Table 6.10 shows Pearson correlation (Cohen et al. 2009) among the five centrality metrics and the aggregated rating. We find weak to moderate to high correlation among different measures. Further, all the centrality metrics, individually, come out significantly correlated with the review scrutiny indicated in Q(c) responses through MELR

[15]`https://github.com/ossf/scorecard`
[16]`https://github.com/ossillate-inc/packj`

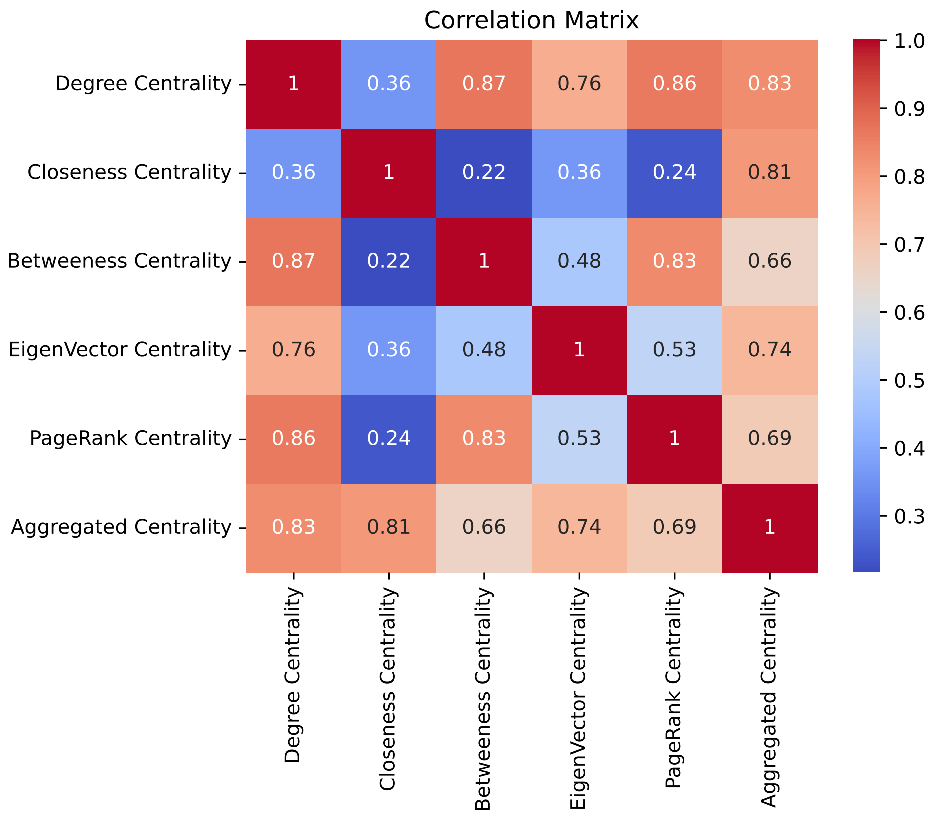


Figure 6.10: Pearson correlation matrix for five centrality measures used to provide an aggregated rating for Rust developers

model discussed in Section 6.5.3 ($p <= 0.001$ for all five metrics). The correlation is expected as all the metrics measure the influence of a node in the network, albeit through different measurements. Prior work (Asim et al. 2019; Bosu et al. 2014; Sabah and Şimşek 2023; Fei et al. 2017; Şimşek and Meyerhenke 2020) in social network analysis literature combined multiple centrality measures to estimate the social capital of a node, such as trust and reputation, to avoid bias coming from any single metric. For example, a developer may have a direct connection with many other developers, all of whom may not be highly trusted in the network. In such a case, the developer will have high direct centrality but low eigenvector centrality. Our correlation matrix in Figure 6.10 supports this rationale as we find not all the centrality metrics are highly correlated with each other. The variance in different metrics can be mitigated through our aggregation approach when providing a single rating for each developer.

## 6.7 Limitations

In this section, we discuss the limitations of our study:

**DSN construction.** The construction of DSN in this paper has multiple limitations: (i) we only considered packages hosted on GitHub; (ii) we only considered developers with a GitHub account; (iii) we only collected code review information based on the evidence present on GitHub; (iv) we did not consider non-code based interactions between developers, like Slack and email, and (v) relationships between upstream and downstream project developers. Further, we may not have captured all existing developer relationships due to the limited data chosen to construct the DSN. When calculating edge strength between a pair of developers, we did not differentiate between different packages, the type, and size of contributions in each commit, and the complexity of different changes. Further, we use a simple addition of normalized scores to aggregate five centrality measures. A more complex method may provide a more accurate proxy trust rating for Rust developers. While a DSN built upon more elaborate data involving more complex modeling may have given us more insights into the community structure and the developers' centrality ratings, we rely on simple design mechanisms following prior work in the literature (Herbold et al. 2021). Being the first work of its kind, our goal was to evaluate the feasibility of constructing DSN at a package ecosystem level and understand if a centrality-based rating can help estimate the trustworthiness of the developers.

**Generalizability Threat.** Our work focuses on the top 1,644 packages within the Rust ecosystem. Our work may not generalize to other package ecosystems like npm and PyPI. Further, our findings may not generalize when scaled to all 92K packages hosted on Crates.io. Additionally, we only worked with data collected from October 2020 to October 2022, which may pose a generalizability threat to our findings if the data is scaled to a more extensive period.

**Misinterpretation of survey questions and reporting bias.** When discussing the answers for Q(c) in our survey, we noticed that 30 respondents appeared to have misinterpreted our questions and answered based on how they would review pull requests in their own projects. While we were able to spot such misinterpretation through the explanation provided by the respondents, there could be other unknown misinterpretations that we are unaware of. We only ask about ten developers to each survey recipient, chosen through a stratified sampling approach. Our sampling approach may introduce unknown biases in our evaluation of RQ3. Further, the response rate in our survey is only 10.3%. While a similar response rate is typical in developer surveys (Smith et al. 2013), our findings may be subject to unknown biases based on who chose to participate in our survey and who did not.

## 6.8 Conclusion

In this paper, we demonstrate a social network-based centrality rating for developers within the Rust community. The rating is aimed to reflect how much a developer's code is trusted by the community so that it can help downstream project developers prioritize their review efforts for changes in upstream packages. To develop such a rating, we construct a social network based on collaboration across the most downloaded 1,644 Rust packages. We show that developers from different packages are interconnected, where each developer is connected to another via only 4 developers on average. Building on this interconnected nature of Rust developers, we propose a global centrality rating for each developer, aggregated over five centrality measures computed from the network. Finally, we conduct a survey among the Rust developers to evaluate if our proposed rating reflects the perceived trustworthiness of developers within the community. Survey responses ($N = 206$) show that the respondents are more likely to not differentiate between developers in deciding how to review upstream changes (60.2% of the time). However, when they do differentiate, our results indicate that the code from the developers with higher centrality ratings is likely to face lesser scrutiny (as per the MELR-based correlation testing). To summarize, the social network-based centrality rating approach shown in this paper can be leveraged to estimate the trustworthiness of a developer in the community and may help downstream project developers decide what level of review new upstream changes may require.

CHAPTER

# 7 CONCLUSIONS

In this dissertation, we studied how to manage open-source security. We study both (a) **reactive**: when a vulnerability is discovered in a dependency, the clients should react to any potential threat; and (b) **proactive**: before pulling in new dependency code, the clients should make an informed decision about the security of the code. In this section, we discuss the summary of our findings and possible future work.

## 7.1 Summary of Findings

In Chapter 3, we present an in-depth case study by comparing the analysis reports of 9 industry-leading software composition analysis (SCA) tools on a large web application, OpenMRS, composed of Maven (Java) and npm (JavaScript) projects. We find that the tools vary in their vulnerability reporting. The count of reported vulnerable dependencies ranges from 17 to 332 for Maven and from 32 to 239 for npm projects across the studied tools. Similarly, the count of unique known vulnerabilities reported by the tools ranges from 36 to 313 for Maven and from 45 to 234 for npm projects. Our manual analysis of the tools' results suggests that the completeness, accuracy, and up-to-dateness of the vulnerability databases are key differentiators for SCA tools.

In Chapter 4, we study (1) the time lag between fix and release; (2) how security fixes are documented in the release notes; (3) code change characteristics (size and semantic versioning) of the release; and (4) the time lag between the release and an advisory publication for security releases over a dataset of 4,377 security advisories across seven package ecosystems. We find that the median security release becomes available within 4 days of the corresponding fix and contains 131 lines of code (LOC) change. However, one-fourth of the releases in our data set still came at least 20 days after the fix was made. Further, we find that 61.5% of the security releases come with a release note that documents the corresponding security fix. Still, Snyk and NVD, two popular databases, take a median of 17 days (from the release) to publish a security advisory, possibly resulting in delayed notifications to the client projects. We also find that security releases may contain breaking change(s) as 13.2% indicated backward incompatibility through semantic versioning, while 6.4% mentioned breaking change(s) in the release notes. Based on our findings, we point out areas for future work, such as private fork for security fixes and standardized practice for announcing security releases. Implementing our recommended practices would help SCA tools maintain an accurate and up-to-date vulnerability database.

In Chapter 5, we work on building trust in dependency updates by quantifying the aspect of human evaluation of the code within the dependency packages. We implement Depdive, an update audit tool for packages in Crates.io, npm, PyPI, and RubyGems registry. Depdive first (i) identifies the files and the code changes in an update that cannot be traced back to the package's source repository, i.e., *phantom artifacts*; and then (ii) measures what portion of changes in the update, excluding the phantom artifacts, has passed through a code review process, i.e., *code review coverage*. Using Depdive, we present an empirical study across the latest ten updates of the most downloaded 1000 packages in each of the four registries. We find that phantom artifacts are not uncommon in the updates (20.1% of the analyzed updates had at least one phantom file). The phantoms can appear either due to legitimate reasons, such as in the case of programmatically generated files, or from accidental inclusion, such as in the case of files that are ignored in the repository. Regarding code review coverage (*CRC)*, we find the updates are typically only partially code-reviewed (52.5% of the time). Further, only 9.0% of the packages had all their updates in our data set fully code-reviewed, indicating that even the most used packages can introduce non-reviewed code in the software supply chain. We also observe that updates either tend to have high *CRC* or low *CRC*, suggesting that packages at the opposite end of the spectrum may require a separate set of treatments.

Finally, we study if a social network-based centrality rating for the authors and reviewers of package code can help client projects review upstream changes during dependency updates. We built a social network of 6,949 developers across the collaboration activity from 1,644 most downloaded Rust packages. We then compute a rating for each developer based on five centrality measures extracted from the network. Further, we survey the developers in the network to evaluate if code coming from a developer with a higher centrality rating is likely to be accepted with lesser scrutiny by the client projects and, therefore, is perceived to be more trusted. Our results show that 97.7% of the developers from the studied packages are interconnected via collaboration, with each developer separated from another via only four other developers in the network. Our survey responses ($N = 206$) show that the respondents are more likely to not differentiate between developers in deciding how to review upstream changes (60.2% of the time). However, when they do differentiate, statistical analysis showed a significant correlation between developers' centrality ratings and the level of scrutiny their code might face from the client projects, as indicated by the respondents. Overall, our findings show that automated techniques can identify if upstream changes have been authored and reviewed by trusted developers in the community and, therefore, may help client projects decide what level of review dependency updates may require. Overall, our findings show that automated techniques can identify if upstream changes have been authored and reviewed by trusted developers in the community and, therefore, may help client projects decide what level of review dependency updates may require.

## 7.2 Future Work

In Chapter 3, we point out two research directions in the SCA space: i) establishing frameworks and metrics to identify false positives for dependency vulnerabilities; and ii) building automation technologies for continuous monitoring of vulnerability data from open-source package ecosystems. As all dependency vulnerabilities may not affect the client project itself, automated techniques need to be researched that can accurately measure the risk of dependency vulnerabilities. In Chapter 4, we recommend standardized practices for announcing security releases to automate the vulnerability notification process to client projects, future work can implement novel notification models based on our recommendations. Further, researchers can look at how clients react when notified of a new dependency vulnerability and what more can be done to help clients fix these vulnerabilities.

In Chapter 5 and ref 6, we explain our research on how to trace the authors and reviewers of the changes in dependency updates, measure code review coverage, and estimate the trustworthiness of these authors and reviewers to help client projects review dependency updates. Our work opens up a series of possible future work. Future work can look at how to implement reviewer signing within git repositories and how packages can publish metadata on their development history so that changes within a dependency update can be reliability audited for code review coverage. Further, future work can look at how to implement third-party code vetting for already published packages and how such a vetting can be incorporated into our proposed code review coverage metric.

In Chapter 6, we show how the social networks of Rust developers can be leveraged to estimate the trustworthiness of individual developers within the ecosystem. Future work can look at how our proposed technique generalizes to other package ecosystems, such as npm and PyPI. Further, future work can develop a trust model for dependency updates that not only incorporate trust ratings for individual developers but also other possible features, such as security practices followed by a package and the download count of the new update.

# REFERENCES

0patch.com (2017). Security patching is hard. `https://0patch.com/files/SecurityPatchingIsHard_2017.pdf`.

Abebe, S. L., Ali, N., and Hassan, A. E. (2016). An empirical study of software release notes. *Empirical Software Engineering*, 21(3):1107–1142.

Abreu, R. and Premraj, R. (2009). How developer communication frequency relates to bug introducing changes. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 153–158.

Agresti, A. and Franklin, C. (2007). The art and science of learning from data. *Upper Saddle River, New Jersey*, 88.

Alali, A., Kagdi, H., and Maletic, J. I. (2008). What's a typical commit? a characterization of open source software repositories. In *2008 16th IEEE international conference on program comprehension*, pages 182–191. IEEE.

Alfadel, M., Costa, D. E., and Shihab, E. (2021a). Empirical analysis of security vulnerabilities in python packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 446–457. IEEE.

Alfadel, M., Costa, D. E., Shihab, E., and Mkhallalati, M. (2021b). On the use of dependabot security pull requests. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 254–265. IEEE.

Amir-Mohammadian, S., Chong, S., and Skalka, C. (2016). Correct audit logging: Theory and practice. In *International Conference on Principles of Security and Trust*, pages 139–162. Springer.

Asim, Y., Malik, A. K., Raza, B., and Shahid, A. R. (2019). A trust model for analysis of trust, influence and their relationship in social network communities. *Telematics and Informatics*, 36:94–116.

Bacchelli, A. and Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE.

Berger, A. (2021). What is log4shell? the log4j vulnerability explained (and what to do about it). `https://www.dynatrace.com/news/blog/what-is-log4shell/`.

Bi, T., Xia, X., Lo, D., Grundy, J., and Zimmermann, T. (2020). An empirical study of release note production and usage in practice. *IEEE Transactions on Software Engineering*.

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

Bommarito, E. and Bommarito, M. (2019). An empirical analysis of the python package index (pypi). *arXiv preprint arXiv:1907.11073*.

Bosu, A. and Carver, J. C. (2014). Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10.

Bosu, A., Carver, J. C., Hafiz, M., Hilley, P., and Janni, D. (2014). Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 257–268.

Brewer, E., Pike, R., Arya, A., Bertucio, A., and Lewandowski, K. (2021). Know, prevent, fix: A framework for shifting the discussion around vulnerabilities in open source. `https://security.googleblog.com/2021/02/know-prevent-fix-framework-for-shifting.html`.

Buskens, V. (1998). The social structure of trust. *Social networks*, 20(3):265–289.

Cadini, F., Zio, E., and Petrescu, C.-A. (2009). Using centrality measures to rank the importance of the components of a complex network infrastructure. In *Critical Information Infrastructure Security: Third International Workshop, CRITIS 2008, Rome, Italy, October13-15, 2008. Revised Papers 3*, pages 155–167. Springer.

Catabi-Kalman, B. (2020). Why do organizations trust snyk to win the open source security battle? `https://snyk.io/blog/why-snyk-wins-open-source-security-battle/`.

Ceolin, D. and Potenza, S. (2017). Social network analysis for trust prediction. In *IFIP International Conference on Trust Management*, pages 49–56. Springer.

Chinthanet, B., Kula, R. G., McIntosh, S., Ishio, T., Ihara, A., and Matsumoto, K. (2021). Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*, 26(3):1–28.

Christey, S. and Martin, B. (2013). Buying into the bias: Why vulnerability statistics suck. *BlackHat, Las Vegas, USA, Tech. Rep*, 1.

Cody, C., Mostafavi, B., and Barnes, T. (2018). Investigation of the influence of hint type on problem solving behavior in a logic proof tutor. In *International Conference on Artificial Intelligence in Education*, pages 58–62. Springer.

Cohen, I., Huang, Y., Chen, J., Benesty, J., Benesty, J., Chen, J., Huang, Y., and Cohen, I. (2009). Pearson correlation coefficient. *Noise reduction in speech processing*, pages 1–4.

Coker, J. (2020). Open source software vulnerabilities increased by 130% in 2019. `https://www.infosecurity-magazine.com/news/open-source-vulnerabilities`.

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

Cox, R. (2019). Surviving software dependencies. *Communications of the ACM*, 62(9):36–43.

Crain, S. P. (2017). Open source security assessment as a class project. *Journal of Computing Sciences in Colleges*, 32(6):41–53.

Das, K., Samanta, S., and Pal, M. (2018). Study on centrality measures in social networks: a survey. *Social network analysis and mining*, 8(1):1–11.

de Abajo, B. S. and Ballestero, A. L. (2012). Overview of the most important open source software: analysis of the benefits of openmrs, openemr, and vista. In *Telemedicine and e-health services, policies, and applications: Advancements and developments*, pages 315–346. IGI Global.

Decan, A. and Mens, T. (2019). What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 47(6):1226–1240.

Decan, A., Mens, T., and Claes, M. (2017). An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2–12. IEEE.

Decan, A., Mens, T., and Constantinou, E. (2018a). On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 404–414. IEEE.

Decan, A., Mens, T., and Constantinou, E. (2018b). On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 181–191.

Delaitre, A. M., Stivalet, B. C., Black, P. E., Okun, V., Cohen, T. S., and Ribeiro, A. (2018). SATE V Report: Ten years of static analysis tool expositions. Technical report, National Institute of Standards and Technology.

Deming, W. E. (2020). For good measure. *USENIX PATRONS*, page 83.

Duan, R., Alrawi, O., Kasturi, R. P., Elder, R., Saltaformaggio, B., and Lee, W. (2020). Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139*.

Edge, J. (2017). A new process for cve assignment. `https://lwn.net/Articles/716437/`.

Faraway, J. J. (2016). *Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models*. CRC press.

Fei, L., Mo, H., and Deng, Y. (2017). A new method to identify influential nodes based on combining of existing centrality measures. *Modern Physics Letters B*, 31(26):1750243.

Ferreira, G., Jia, L., Sunshine, J., and Kästner, C. (2021). Containing malicious package updates in npm with a lightweight permission system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1334–1346. IEEE.

Flowers, R. (2021). Supply chain attack: Npm library used by facebook and others was compromised. `https://hackaday.com/2021/10/22/supply-chain-attack-npm-library-used-by-facebook-and-others-was-compromised/`.

Foo, D., Yeo, J., Xiao, H., and Sharma, A. (2019). The dynamics of software composition analysis. *arXiv preprint arXiv:1909.00973*.

Freeman, L. (2004). The development of social network analysis. *A Study in the Sociology of Science*, 1(687):159–167.

Frei, S., May, M., Fiedler, U., and Plattner, B. (2006). Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 131–138.

Fruhlinger, J. (2020). Equifax data breach faq: What happened, who was affected, what was the impact? `https://www.csoonline.com/article/3444488/equifax-data-breach-faq-what-happened-who-\was-affected-what-was-the-impact.html`.

Giovanini, L., Oliveira, D., Sanchez, H., and Shands, D. (2021). Leveraging team dynamics to predict open-source software projects' susceptibility to social engineering attacks. *arXiv preprint arXiv:2106.16067*.

Gonzalez, D., Zimmermann, T., Godefroid, P., and Schäfer, M. (2021). Anomalicious: Automated detection of anomalous and potentially malicious commits on github. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 258–267. IEEE.

Goswami, P., Gupta, S., Li, Z., Meng, N., and Yao, D. (2020). Investigating the reproducibility of npm packages. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 677–681. IEEE.

Gousios, G., Vasilescu, B., Serebrenik, A., and Zaidman, A. (2014). Lean ghtorrent: Github data on demand. In *Proceedings of the 11th working conference on mining software repositories*, pages 384–387.

Goyal, R., Ferreira, G., Kästner, C., and Herbsleb, J. (2018). Identifying unusual commits on github. *Journal of Software: Evolution and Process*, 30(1):e1893.

Haddad, I. (2020). An open guide to evaluating software composition analysis tools.

Hagberg, A. and Conway, D. (2020). Networkx: Network analysis with python. *URL: https://networkx. github. io*.

Hancock, B., Ockleford, E., and Windridge, K. (2001). *An introduction to qualitative research*. Trent focus group.

Hat, R. (2015). The hidden costs of embargoes. `https://access.redhat.com/blogs/766093/posts/1976653`.

Hawthorne, M. J. and Perry, D. E. (2005). Software engineering education in the era of outsourcing, distributed development, and open source software: challenges and opportunities. In *International Conference on Software Engineering*, pages 166–185. Springer.

Hejderup, J. (2015). In dependencies we trust: How vulnerable are dependencies in software modules? `https://repository.tudelft.nl/islandora/object/uuid:3a15293b-16f6-4e9d-b6a2-f02cd52f1a9e`.

Hejderup, J., Beller, M., and Gousios, G. (2018). Prazi: From package-based to precise call-based dependency network analyses. *Delft University of Technology*.

Herbold, S., Amirfallah, A., Trautsch, F., and Grabowski, J. (2021). A systematic mapping study of developer social network research. *Journal of Systems and Software*, 171:110802.

iba (2019). 9 reasons for keeping software dependencies up to date. `https://nullbeans.com/9-reasons-for-keeping-software-dependencies-up-to-date/`.

Imtiaz, N., Khanom, A., and Williams, L. (2022). Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Transactions on Software Engineering*.

Imtiaz, N., Murphy, B., and Williams, L. (2019). How do developers act on static analysis alerts? an empirical study of coverity usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 323–333. IEEE.

Imtiaz, N., Thorn, S., and Williams, L. (2021). A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11.

Imtiaz, N. and Williams, L. (2022). Phantom artifacts & code review coverage in dependency updates. *arXiv preprint arXiv:2206.09422*.

Kaczorowski, M. (2020). Secure at every step: What is software supply chain security and why does it matter? `https://github.blog/2020-09-02-secure-your-software-supply-chain-and-protect-against-supply-chain-threats-github-blog/`.

Kaplan, B. and Qian, J. (2021). A survey on common threats in npm and pypi registries. In *International Workshop on Deployable Machine Learning for Security Defense*, pages 132–156. Springer.

Kerzazi, N. and El Asri, I. (2016). Who can help to review this piece of code? In *Working Conference on Virtual Enterprises*, pages 289–301. Springer.

Khandkar, S. H. (2009). Open coding. `http://pages.cpsc.ucalgary.ca/~saul/wiki/uploads/CPSC681/opencoding.pdf`.

Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE.

Kinzer, S. (2015). Using cpes for open-source vulnerabilities? think again. `https://www.veracode.com/blog/managing-appsec/using-cpes-open-source-vulnerabilities-think-again`.

Koishybayev, I. and Kapravelos, A. (2020). Mininode: Reducing the attack surface of node. js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 121–134.

Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., and Godfrey, M. W. (2015). Investigating code review quality: Do people and participation matter? In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 111–120. IEEE.

Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. (2018). Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417.

Lacan, O. (2021). keep a changelog. `https://keepachangelog.com/en/1.0.0/`.

Ladisa, P., Plate, H., Martinez, M., and Barais, O. (2022). Taxonomy of attacks on open-source software supply chains. *arXiv preprint arXiv:2204.04008*.

Lam, P., Dietrich, J., and Pearce, D. J. (2020). Putting the semantics into semantic versioning. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 157–179.

Lamp, J., Rubio-Medrano, C. E., Zhao, Z., and Ahn, G.-J. (2018). The danger of missing instructions: a systematic analysis of security requirements for mcps. In *2018 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pages 94–99. IEEE.

Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., and Kirda, E. (2018). Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918*.

Levien, R. (2009). Attack-resistant trust metrics. In *Computing with Social Trust*, pages 121–132. Springer.

Li, F. and Paxson, V. (2017). A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215.

Li, F., Rogers, L., Mathur, A., Malkin, N., and Chetty, M. (2019). Keepers of the machines: examining how system administrators manage software updates. In *Proceedings of the Fifteenth USENIX Conference on Usable Privacy and Security*, pages 273–288. USENIX Association.

Liu, C., Chen, S., Fan, L., Chen, B., Liu, Y., and Peng, X. (2022). Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. *arXiv preprint arXiv:2201.03981*.

Maddox, S. (2010). Writing a security advisory. `https://ffeathers.wordpress.com/2010/08/08/writing-a-security-advisory/`.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60.

Martius, F. and Tiefenau, C. (2020). What does this update do to my systems?–an analysis of the importance of update-related information to system administrators. In *Workshop on Security Information Workers, WSIW*, volume 20, pages 1–12.

Massacci, F. (2022). Pseudo ground-truth generators and large-scale studies. *IEEE Security & Privacy*, 20(02):4–7.

Mcbride, L. (2021). Software supply chains: an introductory guide. `https://blog.sonatype.com/software-supply-chain-a-definition-and-introductory-guide`.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189.

Meneely, A., Tejeda, A. C. R., Spates, B., Trudeau, S., Neuberger, D., Whitlock, K., Ketant, C., and Davis, K. (2014). An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, pages 37–44.

Meneely, A. and Williams, L. (2011). Socio-technical developer networks: should we trust our measurements? In *Proceedings of the 33rd International Conference on Software Engineering*, pages 281–290.

Meneely, A., Williams, L., Snipes, W., and Osborne, J. (2008). Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23.

Meo, P. D., Musial-Gabrys, K., Rosaci, D., Sarne, G. M., and Aroyo, L. (2017). Using centrality measures to predict helpfulness-based reputation in trust networks. *ACM Transactions on Internet Technology (TOIT)*, 17(1):1–20.

Mirhosseini, S. and Parnin, C. (2017). Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 84–94. IEEE.

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A., and Canfora, G. (2016). Arena: an approach for the automated generation of release notes. *IEEE Transactions on Software Engineering*, 43(2):106–127.

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

Nakajima, A., Watanabe, T., Shioji, E., Akiyama, M., and Woo, M. (2019). A pilot study on consumer iot device vulnerability disclosure and patch release in japan and the united states. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 485–492.

Nguyen, V. H., Dashevskyi, S., and Massacci, F. (2016). An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering*, 21(6):2268–2297.

Nia, R., Bird, C., Devanbu, P., and Filkov, V. (2010). Validity of network analyses in open source projects. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 201–209. IEEE.

of Standards, N. I. and (NIST), T. (September 2012). Guide for conducting risk assessments, nist special publication 800-30. `https://csrc.nist.gov/publications/detail/sp/800-30/rev-1/final`. [Online; accessed 7-Oct-2020].

Ohm, M., Plate, H., Sykosch, A., and Meier, M. (2020). Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer.

Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., and Massacci, F. (2020a). Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*.

Pashchenko, I., Vu, D.-L., and Massacci, F. (2020b). A qualitative study of dependency management and its security implications. *Proc. of CCS*, 20.

Pashchenko, I., Vu, D.-L., and Massacci, F. (2020c). A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1513–1531.

Plackett, R. L. (1983). Karl pearson and the chi-squared test. *International Statistical Review/Revue Internationale de Statistique*, pages 59–72.

Plate, H., Ponta, S. E., and Sabetta, A. (2015). Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–420. IEEE.

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

Ponta, S. E., Plate, H., and Sabetta, A. (2020). Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, pages 1–41.

Preston-Werner, T. (2021). Semantic versioning 2.0.0. `https://semver.org`.

Que, X., Checconi, F., Petrini, F., and Gunnels, J. A. (2015). Scalable community detection with the louvain algorithm. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 28–37. IEEE.

Radichel, T. (2017). Why patching software is hard: Technical challenges. `https://www.darkreading.com/vulnerabilities-and-threats/why-patching-software-is-hard-technical-\challenges-/a/d-id/1330181`.

Raemaekers, S., Van Deursen, A., and Visser, J. (2014). Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224. IEEE.

Ramsauer, R., Bulwahn, L., Lohmann, D., and Mauerer, W. (2020). The sound of silence: Mining security vulnerabilities from secret integration channels in open-source projects. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 147–157.

Riquelme, F., Gonzalez-Cantergiani, P., Molinero, X., and Serna, M. (2018). Centrality measure in social networks based on linear threshold model. *Knowledge-Based Systems*, 140:92–102.

Rizvi, S. Z., Fong, P. W., Crampton, J., and Sellwood, J. (2015). Relationship-based access control for openMRS. *arXiv preprint arXiv:1503.06154*.

Ruohonen, J., Hyrynsalmi, S., and Leppänen, V. (2020). A mixed methods probe into the direct disclosure of software vulnerabilities. *Computers in Human Behavior*, 103:161–173.

Sabah, L. and Şimşek, M. (2023). A new fast entropy-based method to generate composite centrality measures in complex networks. *Concurrency and Computation: Practice and Experience*, 35(10):e7657.

Sadowski, C., Söderberg, E., Church, L., Sipko, M., and Bacchelli, A. (2018). Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190.

Schreiber, R. R. and Zylka, M. P. (2020). Social network analysis in software development projects: A systematic literature review. *International Journal of Software Engineering and Knowledge Engineering*, 30(03):321–362.

Schueller, W. and Wachs, J. (2022). Modeling interconnected social and technical risks in open source software ecosystems. *arXiv preprint arXiv:2205.04268*.

Schueller, W., Wachs, J., Servedio, V. D., Thurner, S., and Loreto, V. (2022). Evolving collaboration, dependencies, and use in the rust open source software ecosystem. *Scientific Data*, 9(1):703.

Scott, J. (2012). *What is social network analysis?* Bloomsbury Academic.

Sejfia, A. and Schäfer, M. (2022). Practical automated detection of malicious npm packages. *arXiv preprint arXiv:2202.13953*.

Shabrina, P., Mostafavi, B., Abdelshiheed, M., Chi, M., and Barnes, T. (2022). Investigating the impact of backward strategy learning in a logic tutor: Aiding subgoal learning towards improved problem solving. *arXiv preprint arXiv:2208.04696*.

Shahzad, M., Shafiq, M. Z., and Liu, A. X. (2012). A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781. IEEE.

Shahzad, M., Shafiq, M. Z., and Liu, A. X. (2019). Large scale characterization of software vulnerability life cycles. *IEEE Transactions on Dependable and Secure Computing*, 17(4):730–744.

Sherchan, W., Nepal, S., and Paris, C. (2013). A survey of trust in social networks. *ACM Computing Surveys (CSUR)*, 45(4):1–33.

Şimşek, M. and Meyerhenke, H. (2020). Combined centrality measures for an improved characterization of influence spread in social networks. *Journal of Complex Networks*, 8(1):cnz048.

Singh, S. and Bawa, S. (2007). A privacy, trust and policy based authorization framework for services in distributed environments. *International Journal of Computer Science*, 2(2):85–92.

Smith, E., Loftin, R., Murphy-Hill, E., Bird, C., and Zimmermann, T. (2013). Improving developer participation rates in surveys. In *2013 6th International workshop on cooperative and human aspects of software engineering (CHASE)*, pages 89–92. IEEE.

Snyk (2023). How is a vulnerability's severity determined? `https://support.snyk.io/hc/en-us/articles/360001040078-How-is-a-vulnerability-s-severity-determined-`.

Speicher, R. (2021). `https://about.gitlab.com/blog/2021/01/04/how-we-prevented-security-fixes-leaking-into-our-public-repositories/`.

Sui, L., Dietrich, J., Tahir, A., and Fourtounis, G. (2020). On the recall of static call graph construction in practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1049–1060.

Synopsys (2021). 2021 open source security and risk analysis report. `https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html`.

Tofallis, C. (2014). Add or multiply? a tutorial on ranking and choosing with multiple criteria. *INFORMS Transactions on education*, 14(3):109–119.

Tøndel, I. A., Jaatun, M. G., Cruzes, D. S., and Williams, L. (2019). Collaborative security risk estimation in agile software development. *Information & Computer Security*.

Torres-Arias, S., Afzali, H., Kuppusamy, T. K., Curtmola, R., and Cappos, J. (2019). in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1393–1410.

Vu, D.-L., Massacci, F., Pashchenko, I., Plate, H., and Sabetta, A. (2021). Lastpymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 780–792.

Walsh, T. et al. (1999). Search in a small world. In *Ijcai*, volume 99, pages 1172–1177. Citeseer.

Washburn, J. (2017). What is foss, and why should i be worried about it? `https://www.stoelprivacyblog.com/2017/10/articles/software/what-is-foss-and-why-should-i-be-worried-about-it/`.

Wasserman, S. and Faust, K. (1994). Social network analysis: Methods and applications. `https://www.google.com/books/edition/Social_Network_Analysis/CAm2DpIqRUIC?hl=en&gbpv=0`.

Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442.

Weisstein, E. W. (2004). Bonferroni correction. *https://mathworld. wolfram. com/.*

Wermke, D., Wöhler, N., Klemmer, J. H., Fourné, M., Acar, Y., and Fahl, S. (2022). Committed to trust: A qualitative study on security & trust in open source software projects. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*. IEEE Computer Society.

Wicks, D. (2017). The coding manual for qualitative researchers. *Qualitative research in organizations and management: an international journal.*

Willis, T. (2021). Project zero. `https://googleprojectzero.blogspot.com/2021/04/policy-and-disclosure-2021-edition.html`.

Wood, G. (2016). Polkadot: Vision for a heterogeneous multi-chain framework. *White paper*, 21(2327):4662.

Xu, Z., Chen, B., Chandramohan, M., Liu, Y., and Song, F. (2017). Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472. IEEE.

Yakovenko, A. (2018). Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper.*

Yang, J., Lee, Y., and McDonald, A. P. (2021). Solarwinds software supply chain security: Better protection with enforced policies and technologies. In *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 43–58. Springer.

Zahan, N., Williams, L., Zimmermann, T., Godefroid, P., Murphy, B., and Maddila, C. (2021). What are weak links in the npm supply chain? *arXiv preprint arXiv:2112.10165.*

Zahan, N., Zimmermann, T., Godefroid, P., Murphy, B., Maddila, C., and Williams, L. (2022). What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 331–340.

Zahi, A. H. and Hasson, S. T. (2020). Improved trust model based on centrality measures and recommendation in social network. In *2020 4th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, pages 837–842. IEEE.

Zapata, R. E., Kula, R. G., Chinthanet, B., Ishio, T., Matsumoto, K., and Ihara, A. (2018). Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 559–563. IEEE.

Zerouali, A., Constantinou, E., Mens, T., Robles, G., and González-Barahona, J. (2018). An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*, pages 95–110. Springer.

Zerouali, A., Mens, T., Decan, A., and De Roover, C. (2021). On the impact of security vulnerabilities in the npm and rubygems dependency networks. *arXiv preprint arXiv:2106.06747*.

Zhou, J., Pacheco, M., Wan, Z., Xia, X., Lo, D., Wang, Y., and Hassan, A. E. (2021a). Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 705–716. IEEE.

Zhou, Y. and Sharma, A. (2017). Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 914–919.

Zhou, Y., Siow, J. K., Wang, C., Liu, S., and Liu, Y. (2021b). Spi: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–27.

Zimmermann, M., Staicu, C.-A., Tenny, C., and Pradel, M. (2019). Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 995–1010.

Zoric, N. (2018). A tutorial for tagging releases in git. `https://dev.to/neshaz/a-tutorial-for-tagging-releases-in-git-147e`.

## APPENDIX

APPENDIX

# A ACRONYMS

A summary of all acronyms is documented in Table A.1.

Table A.1: A summary of acronyms used in alphabetical order.

| Acronym | Abbreviation |
|---|---|
| Software Composition Analysis | SCA |
| National vulnerability Database | NVD |
| Common Weakness Enumeration | CWE |
| Common Vulnerabilities & Exposure | CVE |
| Lines of Code | LOC |
| Code Review Coverage | CRC |
| Supply chain Levels for Software Artifacts | SLSA (salsa) |
| Developer Social Network | DSN |
| Social Network Analysis | SNA |