

**Analyzing the Direct and Transitive Impact of Vulnerabilities onto Diferent Artifact Repositories**

JOHANNES DÃĲSING, Technical University Dortmund, Germany BEN HERMANN, Technical University Dortmund, Germany

In modern-day software development, a vast amount of public software libraries enable the reuse of existing implementations for reoccurring tasks and common problems. While this practice does yield signiicant beneits in productivity, it also puts an increasing amount of responsibility on library maintainers. If a security law is contained in a library release, it may directly afect thousands of applications that are depending on it. Given the fact that libraries are often interconnected, meaning they are depending on other libraries for certain sub-tasks, the impact of a single vulnerability may be large, and is hard to quantify. Recent studies have shown that developers in fact struggle with upgrading vulnerable dependencies, despite ever-increasing support by automated tools, which are often publicly available. With our work, we aim to improve on this situation by providing an in-depth analysis on how developers handle vulnerability patches and dependency upgrades. In order to do so, we contribute a miner for artifact dependency graphs supporting diferent programming platforms, which annotates the graph with vulnerability information. We execute our application and generate a data set for the artifact repositories Maven Central, NuGet.org, and the NPM Registry, with the resulting graph being stored in a Neo4j graph database. Afterwards, we conduct an extensive analysis of our data, which is aimed at understanding the impact of vulnerabilities for the three diferent repositories. Finally, we summarize the resulting risks and derive possible mitigation strategies for library maintainers and software developers based on our indings. We found that NuGet.org, the smallest artifact repository in our sample, is subject to fewer security concerns than Maven Central or the NPM Registry. However, for all repositories, we found that vulnerabilities may inluence libraries via long transitive dependency chains and that a vulnerability in a single library may afect thousands of other libraries transitively.

CCS Concepts: · **Security and privacy** → **Vulnerability management**.

Additional Key Words and Phrases: Vulnerability, Impact Analysis, Repository Mining, Patch Detection

1 INTRODUCTION

Software reuse has proven to be an essential part of software development [5, 18]. A vast amount of public repositories exist, where developers can choose open-source software libraries for a variety of diferent tasks and programming environments. Various build systems like NPM [19], Nuget [25] or Maven [14] conveniently integrate the management of external components into development worklows, thus allowing developers to browse and add components by the click of a button.

As a result of the continuous improvement in tooling support, the amount of open-source library code has been shown to be more than 45% of the total project code for open-source Java projects, with only 10% of projects not reusing code at all [18]. Similarly, the 2021 Synposis Open Source Security Report inds that of 1,546 commercial codebases analyzed in 2020, 98% contained open-source library code, with the average application being comprised of 75% open source [42].

Authors’ addresses: [Johannes DÃĳsing,](https://orcid.org/0000-0002-9367-2206) Technical University Dortmund, Otto-Hahn-StraÃğe 14, Dortmund, Germany, johannes.duesing@tu-dortmund.de; [Ben Hermann,](https://orcid.org/0000-0001-9848-2017) Technical University Dortmund, Otto-Hahn-StraÃğe 14, Dortmund, Germany, ben.hermann@cs.tu-dortmund.de.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for proit or commercial advantage and that copies bear this notice and the full citation on the irst page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2022 Copyright held by the owner/author(s).

2576-5337/2022/1-ART1   
<https://doi.org/10.1145/3472811>

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| 1:2 | • | Johannes DÃĳsing and Ben Hermann |

It has been established by W.C. Lim and others, that software reuse has several positive efects on a software product and its development process, including increased productivity and shorter time-to-market [22]. By May 2021, almost 7 million artifacts are available on Maven Central, which is only one of many repositories for open-source Java libraries [28]. A similar trend in the availability of reusable software artifacts can be observed in other repositories as well, with examples like the NPM Registry and NuGet.org hosting over 16 million artifacts combined (cf. Section 4.3).

However, using external open-source libraries from repositories like NuGet.org also introduces some distinct disadvantages. Most prominently, those libraries impose a new level of complexity on a project. Many of them have dependencies to other libraries of the same repository, thus creating a potentially large dependency graph (often called dependency tree), containing both the direct and transitive dependencies of a project. Decan et al. have shown in 2019 that these graphs can exhibit depths larger than six, with roughly 25% of all artifacts having a dependency graph of depth at least three. The authors also observe that the ratio of transitive to direct dependencies roughly equals 15 for both the NPM Registry and NuGet.org [11]. Overseeing the diferent responsibilities, release dates and available versions of all libraries in that graph is a complex task that is likely to add additional overhead to the development process [32].

This fact becomes especially relevant when considering vulnerabilities in software libraries. A single vulnera-bility in a certain library may break the security of all libraries and projects using it, be it directly or transitively. While vulnerabilities being present in a dependency does not necessarily mean vulnerable code is being used [6], Lawrence and Frohof have shown that in Java vulnerabilities may be exploited simply because the corresponding code is on the classpath [21]. By May 2021, one single release of a popular library like jackson-databind is directly used by more than 18,000 other libraries on Maven Central alone [29], which underlines the impact a single vulnerability may have onto the whole repository.

In order to minimize the risk of developers not knowing about vulnerabilities in their library dependencies, those vulnerabilities are often disclosed using the CVE standard. The list of all known CVE vulnerabilities is publicly available [9] and frequently updated. The list has been started in 1999 and has, as of May 2021, more than 153,000 entries, averaging at more than 19 vulnerabilities per day. While those entries are not solely related to software libraries, but also applications, these numbers still indicate that ixing vulnerabilities in software libraries is not a rare case, but something that must be anticipated during the development process.

In general, the process of safely dealing with a newly published vulnerability is threefold:

(1) The maintainers of the afected library publish a new version that does not contain the vulnerability anymore. We call this new version a patch for the vulnerability.

(2) The maintainers of all libraries directly using the afected library need to publish new versions that upgrade their dependencies to use the patch instead of the vulnerable version of the library. This process needs to be iterated recursively to account for transitive dependencies.

(3) Similar to library maintainers, project developers need to upgrade their dependencies for any of the (transitively) afected libraries to not use vulnerable versions anymore.

While in theory, this process is adequate to deal with the problem at hand, in reality, there are several issues that negatively inluence its applicability. Most importantly, it requires the developers of all libraries to be actively maintaining their code. Moreover, they have to constantly monitor all published vulnerabilities and patches, and correlate that data with the dependency graph of their library. While loating version references (i.e. dependencies with a range of valid version numbers) may in some cases enable maintainers to automatically beneit from vulnerability patches, they still need to verify that the vulnerability does not apply anymore.

As the act of monitoring vulnerability publications for every artifact of a dependency graph is both cumbersome and repetitive, to this day several tools have been developed in an efort to automate this task. A prominent example is Snyk, which can be added to software repositories as a CI check that informs developers of vulnerabilities in

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| Analyzing Vulnerability Impact | • | 1:3 |

the project’s dependencies, as well as their severity and possible patches. Going one step further, Dependabot automatically upgrades vulnerable dependencies and creates a corresponding pull request.

Despite the availability of tools like Snyk and Dependabot addressing the problem of vulnerable dependencies, a 2017 study performed by Kula et al. found that 69% of developers claim to be unaware of being afected. As a result, more than 81% of the systems they analyzed contained vulnerable dependencies [20]. In addition to that, some library maintainers may never release a patched version, as they decided to stop maintaining their library. This information is not easy to obtain, as there is no standard way of publishing it. Pashchenko et al. [32] argue that such halted dependencies may be identiied by using a heuristic based on the average interval between diferent releases.

In their study, Kula et al. conclude that one reason for this is the developer’s perception of security patches and dependency upgrades as added responsibility yielding little to no beneits for the user [20]. While the act of upgrading a dependency does not consume much time, it requires careful consideration to account for compatibility, performance, and security. Therefore, a good understanding of the respective repository and its patching behavior is key to making informed decisions for dependency upgrades, thus ultimately improving application security.

However, the question of how and when developers release patches and upgrade dependencies in diferent artifact repositories has yet to be answered. This is due to a lack of empirical data on the subject, especially regarding the diferences between repositories. To this day, there is no vulnerability-enriched dependency graph available for any of the major artifact repositories, although that data is freely accessible online. As a result, no formal analysis has yet been conducted on patching behavior in the presence of multi-level dependency graphs.

The main goal of our work is to provide detailed insights into the problem of vulnerable software artifacts and their dependencies by conducting an in-depth analysis that investigates the patching behavior across diferent repositories. Based on that, we derive development strategies that increase application security. In summary, we contribute:

(1) A distributed application for mining the dependency graphs of diferent software repositories and annotating it with vulnerability information. Our implementation is presented in Section 4 and published on Zenodo [12].

(2) An analysis on the patching behavior of library developers in NuGet.org, the NPM Registry, and Maven Central. We highlight our analysis design, methodology, and results in Section 5.

(3) A set of resulting threats to application security and possible mitigation strategies, which are discussed in Section 6.

2 BACKGROUND

Build Systems like Maven or NPM resolve dependency speciications in so-called software artifact repositories. Popular examples for such repositories are Maven Central with almost 7 million artifacts by May 2021 [28], the NPM Registry (around 13.5 million artifacts, cf. Figure 4) and NuGet.org with around 2.8 million artifacts [26]. These repositories generally expose a web-based interface granting full read-access to any client that wants to consume the contained artifacts. For Maven Central this interface is ile-based1, while for other examples like NuGet.org2and the NPM Registry3it is a JSON-based artifact index.

When developers reference an artifact in their project, that artifact may itself be depending on other artifacts of the same repository, thus introducing transitive dependencies to the project. Consequently, each project has a dedicated dependency graph, which may span multiple levels and is resolved by the build system. Developers do not

1repo1.maven.org/maven2   
2api.nuget.org/v3/catalog0/index.json   
3skimdb.npmjs.com/registry/\_all\_docs

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| 1:4 | • | Johannes DÃĳsing and Ben Hermann |

necessarily know about the transitive dependencies in their project, as they are typically resolved transparently.

However, commands like mvn dependency:tree4may be used to explicitly list the full dependency graph.

There are two ways of referencing software artifacts in build systems for the context of this work. The irst one, which is used predominantly by Maven, is to specify the unique library identiier and the version number of the target artifact. We call this a ixed artifact reference, as it references exactly one target artifact. While Maven also supports the use of version ranges [30], we ind that these are rarely used in any of the artifacts online. On the other hand, loating artifact references, which are used in NPM and NuGet, consist of the unique library identiier and a range of valid version numbers. For this type of reference, the build system selects one of the valid target artifacts, which usually is the most recent release of the respective library.

Vulnerabilities in software artifacts can be discovered both unintentionally, e.g. by developers using the artifact, or intentionally, e.g. by users reviewing the source code of an artifact. Frei et al. illustrate the diferent options a discoverer has upon inding a vulnerability [15]. These include selling it on the black or white market, creating an exploit or a patch, or disclosing it to the public. For the context of this work, the last option is of special interest.

Launched by the MITRE Corporation in 1999, the CVE list [9] is an online database of such publicly disclosed vulnerabilities, and has since emerged as a de facto standard for identifying and referencing vulnerabilities [15]. For each vulnerability, it holds a unique identiier, a description, and a public reference. The NVD was created in 2005 and builds upon the CVE list by providing additional information on ixes, severity, and impact of each CVE entry [8].

Several tools incorporate those data sources in order to inform developers about vulnerabilities. Prominent examples include the Maven plugin Dependency-Check Maven [38] and Dependabot5. The services provided by Snyk6rely on a proprietary vulnerability database, which incorporates not only vulnerabilities published by the NVD but also indings from academic collaborations and proprietary research, as well as vulnerabilities reported by the community. According to Snyk, each inding is manually tested for accuracy and enriched with additional metadata, including an explicit link to the set of afected software artifacts [24]. Furthermore, for vulnerabilities not imported from public sources like the NVD, Snyk distinguishes the date of disclosure and date of publication. Upon disclosure, Snyk assess the severity and risks of vulnerabilities and, following an explicit Disclosure Policy, contacts the maintainers of the afected modules. After that, Snyk cooperates with the module maintainers and publishes the vulnerability information following a public disclosure timeline. Finally, as Snyk is oicially registered as CVE Central Naming Authority, it assigns a CVE identiier for the vulnerability [23].

3 APPROACH

Our approach incorporates three diferent phases. First, we aggregate data on software artifact dependencies for diferent repositories and annotate vulnerability information. We then deine research questions and answer them using the data produced in the previous phase. With our research questions, we investigate how and when library maintainers upgrade a directly or transitively vulnerable release, and how many artifacts are afected by vulnerabilities in the irst place. Finally, we discuss the results, identify potential security threats and derive mitigation strategies.

For data aggregation, we design and implement a distributed application called Miner. This application builds the artifact dependency graph for diferent repositories, which is done by querying the respective web interfaces. In a second step, the Miner utilizes a customized dump of the Snyk database to annotate the dependency graph with vulnerability information. The graph is then stored in a database for further processing. We present the

4maven.apache.org/plugins/maven-dependency-plugin/examples/resolving-conlicts-using-the-dependency-tree.html 5dependabot.com/   
6snyk.io

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| Analyzing Vulnerability Impact | • | 1:5 |

design of our Miner in Section 4.1, highlight implementation details in Section 4.2, and present insights into the resulting data set in Section 4.3.

In the second phase, we conduct our analysis, which is guided by four diferent research questions. We present those questions and deine their scope in Section 5.1. In order to answer those questions, we implement diferent analysis applications that extract metrics from the annotated dependency graph, and also implement tools for manual data exploration. We provide the results for each analysis in Section 5.3, which includes diagrams and plots of the previously generated metrics, as well as the outcomes of our manual analyses.

Finally, we discuss the results of our analysis in Section 6. We systematically derive potential threats to application security from our indings and analyze which repositories are afected by them. We then propose possible mitigation strategies for those threats, which are meant to help software developers improve the security of their development process.

4 DATA AGGREGATION

In order to aggregate the data set that we perform our analysis on, we developed a distributed application called Miner. It extracts the artifact dependency graph from diferent repositories, annotates it with vulnerability information and stores it in a persistent fashion for further processing. In this chapter, we present the design and implementation of said application.

4.1 Design

In this section, we present the core design decision upon which we based application development. We aimed to include all relevant properties that might increase analysis expressiveness in the inal data model.

Selecting Repositories. While we designed our application with extensibility in mind, we selected an initial set of artifact repositories in order to identify a superset of relevant artifact properties. We decided to include repositories for three programming platforms: the JVM, .NET, and JavaScript. To ensure the relevance of our work, we consider only the most popular repository for each platform, yielding Maven Central (JVM), NuGet.org (.NET) and the NPM Registry (JavaScript).

Defining a Suitable Data Model. The creation of a suitable data model for the resulting data set involves deining properties for artifacts, vulnerabilities, and the relation of those two entities.

We deine our data model for software artifacts by inspecting the data provided by the three selected repositories.

Based on that, we identify common properties and evaluate their usefulness in the context of our work.

While named diferently, all three repositories incorporate the concept of a unique library identiier, which is always accompanied by a corresponding version number. In Maven Central, the library identiier is composed of two attributes called groupId and artifactId, for NuGet.org and the NPM Registry it is a simple attribute called id and name, respectively. The concatenation of library identiier and version number yields a unique artifact identiier. Also, all three repositories expose the publication date of an artifact.

Naturally, a dependency graph cannot be constructed without information about an artifact’s dependencies. We represent each dependency as a tuple, containing both a library identiier and a domain-speciic expression encoding a range of valid version numbers. As a result, the NPM dependency "lodash":"^4.17.4" references all artifacts of library lodash with a version number greater than or equal to 4.17.4 up until the next major version 5.0.0, excluding. For our data model, we introduce an entity that holds three properties: the target’s repository name and library identiier, as well as a repository-speciic version range speciier. For the sake of traceability, we keep the original deinition of the range speciier in our datamodel, and only normalize the data when resolving references in a later stage (cf. Section 4.2).

Digit. Threat. Res. Pract.

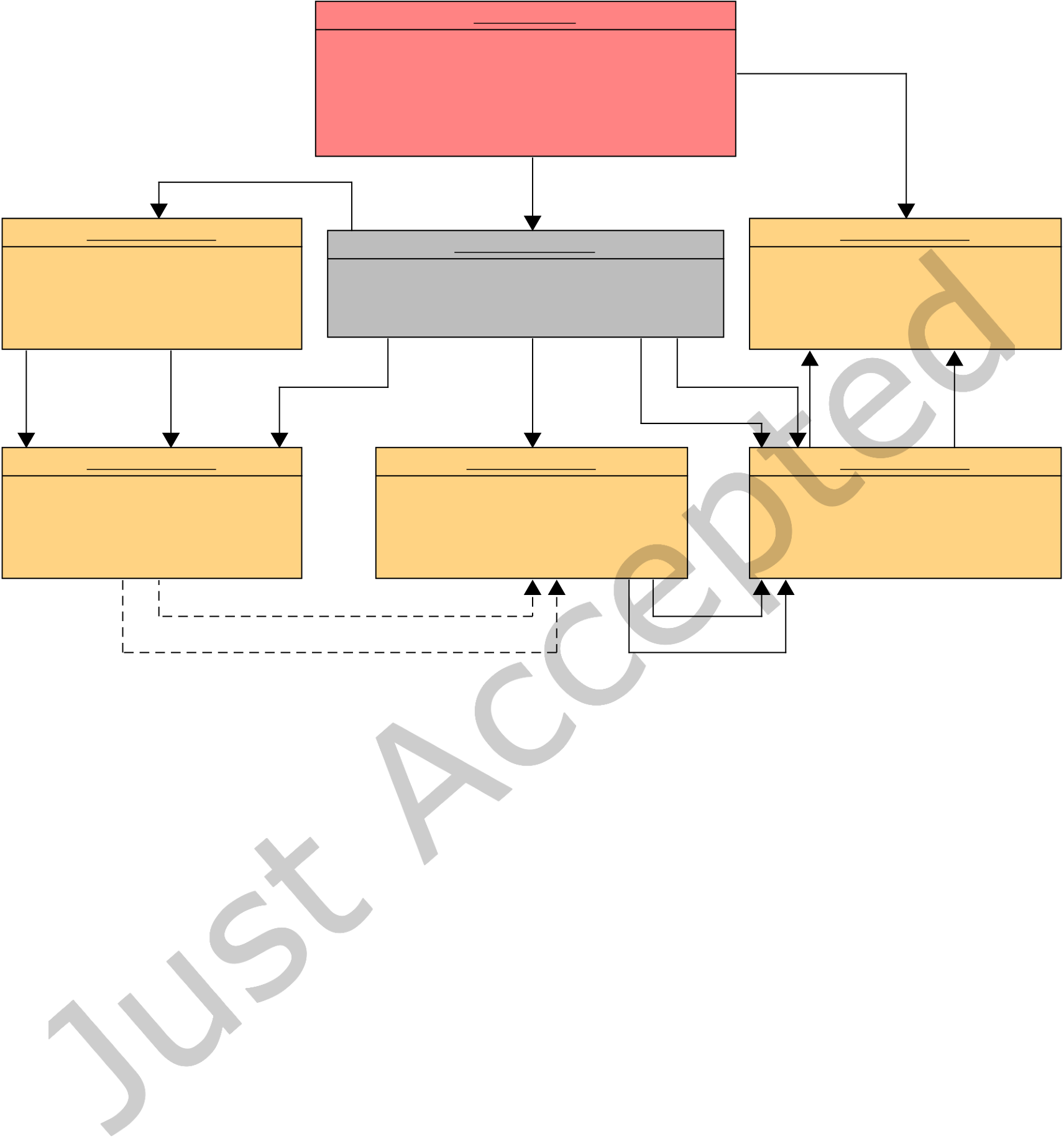
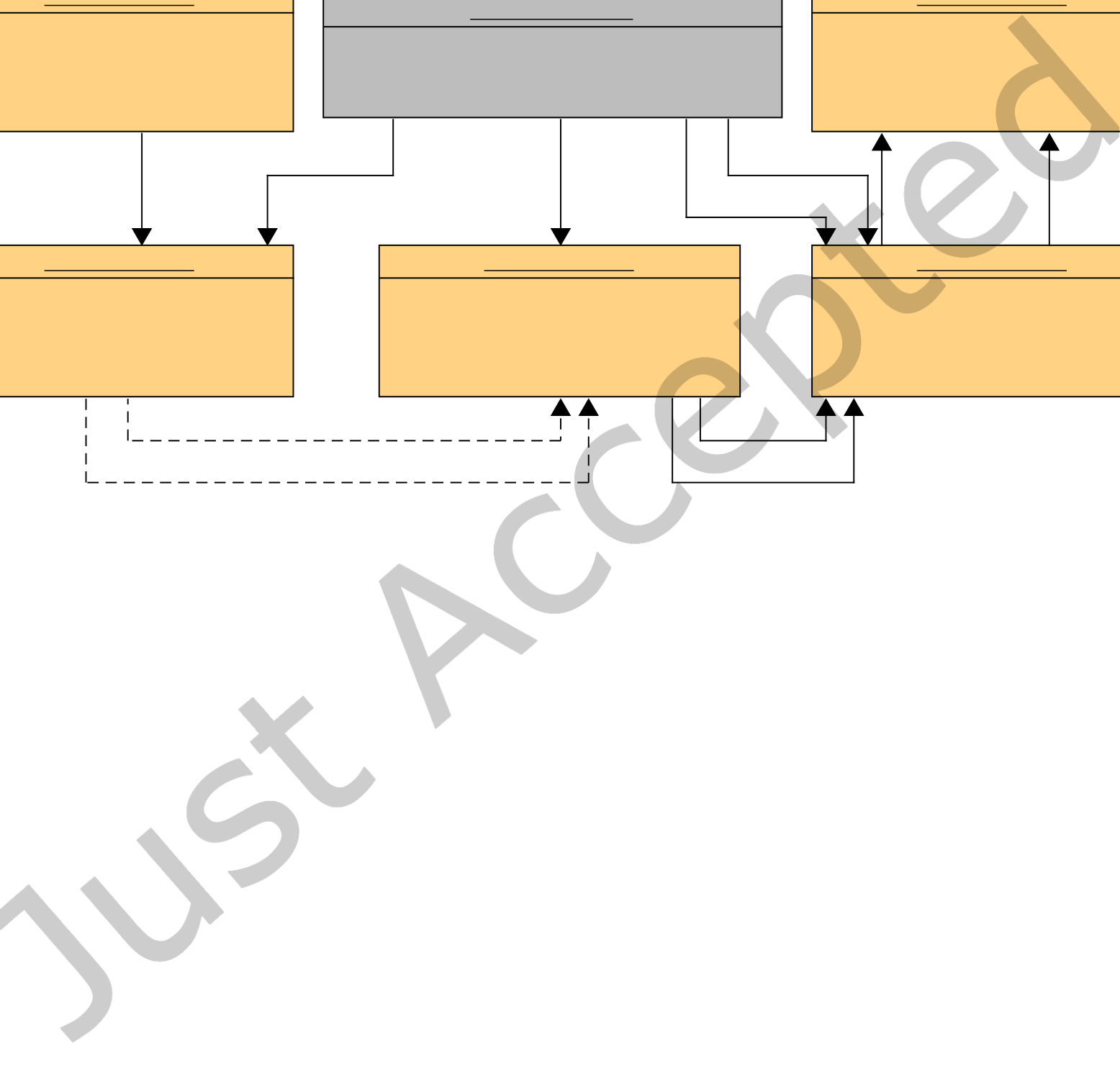
|  |  |  |
| --- | --- | --- |
| 1:6 | • | Johannes DÃĳsing and Ben Hermann |

So in summary, the set of properties for software artifacts in our data model contains a unique library identiier, a version number, the release date, and a list of artifact dependencies.

Our source of information on software vulnerabilities is a customized data dump of the Snyk Vulnerability DB, which was provided to us by Snyk Ltd. in an efort to support our research, and is up-to-date as of 16thJuly 2020. Based on this data source, we deine our data model for vulnerabilities, which includes the vulnerability’s internal Snyk identiier, its severity (Critical, High, Medium or Low) and a reference to an associated CVE identiier, if available. Furthermore, we incorporate the date of disclosure and date of publication. Lastly, vulnerabilities must be linked to the software artifacts that they afect. In Snyk, this is done by providing a unique library identiier and domain-speciic version range speciier, for which they use a syntax that is diferent in all three repositories. Similar to artifact dependencies, we keep the domain-speciic range speciier in our data model, and normalize the data when it is processed.

|  |  |
| --- | --- |
| is patched by > | next release > |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | referenc | es > |  |  | 0..m | | 0..1 | 0..1 |
| 0..n |  |  | 0..n |  |  |  |
|  |  |  |  |  |  | *SoftwareArtifact* | | | | |
| Vulnerability | |  |  |  |  |  |
|  | ArtifactReference | |  |  | + Repository: String + UniqueId: String  + LibraryId: String  + Version: String  + ReleaseDate: Date | | | | |
| + SnykId: String  + AssociatedCVE: String + Severity: String  + PublishedAt: Date  + DisclosedAt: Date | |  |  |  |
|  | + TargetRepository: String + LibraryIdReference: String + VersionRange: String | |  |  |
|  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |



|  |  |  |
| --- | --- | --- |
| Analyzing Vulnerability Impact | • | 1:7 |

: Vulnerability

SnykId = "SNYK-DOTNET-JQUERY-565440"

AssociatedCVE = �"CVE-2020-11023"

Severity = "medium"

PublishedAt = "2020-04-13T15:33:49"

DisclosedAt = "2020-04-10T00:00:00"

patched by

references

affects

: SoftwareArtifact : SoftwareArtifact

: ArtifactReference

Repository = "nuget" Repository = "nuget"

UniqueId = "jQuery:1.4.1" TargetRepository = "nuget" UniqueId = "jQuery:3.5.0"

LibraryId = "jQuery" LibraryIdReference = "jQuery" LibraryId = "jQuery"

Version = "1.4.1" VersionRange = "[1.0.3 ,3.5.0)" Version = "3.5.0"

ReleaseDate = "2011-01-07T07:54:41" ReleaseDate = "2020-04-146T21:42:55"

references current target

next release next version references next release next version

references

: SoftwareArtifact : SoftwareArtifact : SoftwareArtifact

Repository = "nuget" Repository = "nuget" Repository = "nuget"

UniqueId = "jQuery:1.4.2" UniqueId = "jQuery:3.4.0" UniqueId = "jQuery:3.4.1"

LibraryId = "jQuery" LibraryId = "jQuery" LibraryId = "jQuery"

Version = "1.4.2" Version = "3.4.0" Version = "3.4.1"

ReleaseDate = "2011-01-07T07:54:42" ReleaseDate = "2019-05-026T14:16:36" ReleaseDate = "2019-05-026T18:16:50"

next version (transitive x54) next version

next release (transitive x54) next release

Fig. 2. Example instance of the data model shown in Figure 1

the data model deined in Figure 1. As the interfaces of all three repositories are diferent from each other, there

is one dedicated miner implementation per repository.

The task of parsing the list of vulnerabilities and transforming them to the aforementioned data model is

handled by the Vulnerability Processor. Finally, the Reference Resolver enumerates all ArtifactReference entities

in the database, which implicitly reference a set of artifacts, and converts them to explicit entity-to-entity relations.

This is done so that the subsequent analysis does not need to re-resolve those implicit relations. Using those

explicit relations, the Patch Detector then enumerates all vulnerabilities in the data set and inds those artifacts

that patch the vulnerability, i.e. the irst artifacts of the targeted library that are not afected by the vulnerability

anymore. The vulnerability and its patching artifact are connected via an entity-to-entity relation in the database.

4.2 Implementation

As previously mentioned, we design a distributed application for the task of aggregating our data set. We

separately implement and package each component, and manage their deployment using Docker and Portainer.

For our implementations we use Python 3.8, and our Docker images are based on the python-alpine image in

version 3.8.2-alpine3.11.

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| 1:8 | • | Johannes DÃĳsing and Ben Hermann |

We publish our miners and the respective components for postprocessing as open-source Docker images via GitHub8. Due to our data on vulnerabilities being the intellectual property of Snyk Ltd., we cannot publish any components handling this data. However, they can be obtained directly from Snyk Ltd.

In the following, we highlight the most important implementation details, the challenges we faced, and the tools we utilized.

Database. As our data set is essentially a large dependency graph with annotated vulnerability nodes, we select the graph database Neo4j as our storage backend. We deploy it using the oicial Docker image9in version 4.0.5.

Miners. For each repository, we develop and implement a dedicated miner component. For Maven Central we initially relied on an existing implementation by Benellalam et al. [4], which also stores a full dependency graph using Neo4j. However, due to multiple problems we faced when executing this implementation (cf. Section 4.3), we had to re-implement the Maven miner from scratch.

All our implementations follow an asynchronous Producer-Consumer-Architecture: a producer enumerates artifacts, which are grouped into batches and piped to a pool of consumer threads. Those consumers store all artifacts in the database. By decoupling those two processes, periods of low storage throughput do not slow down the artifact enumeration process.

For NuGet.org, artifacts are enumerated by consuming an append-only catalog that keeps track of all types of artifact-related events. For each publication event, the miner retrieves corresponding artifact properties by issuing a single HTTP request to the NuGet.org API. The catalog consists of multiple pages, for which the index is located at api.nuget.org/v3/catalog0/index.json.

In contrast to that, the NPM Registry provides a single index ile at replicate.npmjs.com/\_all\_docs, which contains identiiers for all libraries ever created. Our NPM Miner utilizes this identiier to retrieve a list of all artifacts belonging to that library at the NPM Registry API.

Finally, for Maven Central we leverage an Apache Lucene index of all artifact identiiers, which is located at <https://repo.maven.apache.org/maven2/.index/> and updated weekly. For each identiier, the miner downloads the corresponding pom.xml ile from Maven Central and extracts information about the artifact and its dependencies. However, in contrast to other repositories, Maven’s dependencies cannot always be extracted by looking at a single ile. They may be speciied in one of the artifact’s parent POM iles, or use property values that need to be resolved in the POM ile hierarchy. These problems can be addressed by using the Maven command-line interfaces to resolve dependencies, which does, however, take a very long time and uses a signiicant amount of disk space for POM ile caching. Instead, we developed our own dependency resolver using the standard Java XML libraries, which signiicantly increases the performance of our application. To guarantee correctness, we use a secondary dependency resolver as a fallback: Whenever our XML resolver fails to resolve a property value (which may happen in cases of so-called import-scope dependencies), we re-resolve dependencies using the Eclipse Aether library10. As this library is solely available for the JVM platform, we implement the Maven miner in Java and deploy it using the oicial maven Docker image in version 3.6.0-jdk-8-alpine.

Vulnerability Processor. The vulnerability processor consumes our snapshot of the Snyk Vulnerability Database, which has been provided in the JSON ile format and creates corresponding entities of type Vulnerability and ArtifactReference in the database.

Postprocessing. After the Miners and Vulnerability Processor have aggregated all data necessary, two distinct components perform postprocessing steps in order to prepare the data for analysis.

8The repository is located at [https://github.com/sse-labs/dependency-graph-miner.](https://github.com/sse-labs/dependency-graph-miner) An archived version is available via Zenodo at [https: //doi.org/10.5281/zenodo.5040439](https://doi.org/10.5281/zenodo.5040439)   
[9hub.docker.com/\_/neo4j](https://doi.org/10.5281/zenodo.5040439)   
10wiki.eclipse.org/Aether

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| Analyzing Vulnerability Impact | • | 1:9 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 1.0.0 | 1.8.0 | 1.8.11 | 1.8.12 | 1.8.13 | 1.9.0 | 1.9.9 | 2.0.0 | 2.0.3 | 3.0.0 | |

Fig. 3. Detecting patching library releases

The Reference Resolver converts a number of implicit relations between database entities into explicit relations, which enables us to greatly improve the performance of our subsequent analyses and simpliies transitive queries to the database. It irst processes all artifacts for each unique library identiier and creates relations of type NEXT and NEXT\_RELEASE. These represent the order of versions (according to the Semantic Versioning 2.0.0 speciication11) and the order of release dates for all artifacts of the given library, as illustrated in our data model in Figure 1. Afterwards, the Reference Resolver connects all database entities of type ArtifactReference with their target artifacts by creating direct relations of type REFERENCES. This is done by parsing the speciic version range qualiier and then iltering all artifacts of the target library for matching versions. The Reference Resolver also creates a special relation called CURRENT\_TAR’ET between each ArtifactReference and its target artifact with the highest version number. The result of a successful run based on a reference to the Nuget.org library jQuery can be seen in Figure 2.

Finally, the Patch Detector builds on top of the transformations performed by the Reference Resolver and detects the set of patches for each vulnerability. As the range of version numbers afected by a vulnerability may be discontinuous, it is possible that there are multiple patches for a single vulnerability. The component incorporates two diferent approaches: First it checks whether we have explicit data on the patching software artifact, which is true for most of the vulnerabilities. If no such data exists, the patching artifacts are calculated by inding the irst non-afected library releases for which the predecessor is afected by the vulnerability. This process is illustrated in Figure 3, where version numbers highlighted red are afected by a vulnerability, and green version numbers indicate the detected patches. In the database, a vulnerability is connected to each of its patches with a relation of type PATCHED\_BY.

4.3 Execution Results

We initially executed our miner throughout the course of 70 days. The irst phase was dedicated to running the Miners for all three repositories, which took a total of 43 days. The second step of executing the Vulnerability Processor took less than one minute, whereas the postprocessing phase required another 27 days to complete.

We identiied two main factors with signiicant impact on the application performance:

**Maven Miner** For our Maven Miner component we initially reused an existing implementation published by Benelallam et al. [4] in 2019. During its execution, we observed a large variation regarding the artifact mining speed. Furthermore, some internal components of the application required constant manual restarts, which lead to additional performance penalties. Due to those problems, we aborted the program execution and instead developed our own Maven Miner implementation from scratch (cf. Section 4.2).

**Reference Resolver** The Reference Resolver identiies potential targets of a reference, which involves a lookup via their unique library identiier. For all three repositories, these identiiers are generally case-insensitive. However, in NuGet.org references often use a style of capitalization that difers from the original deinition. Therefore, the Reference Resolver is required to do a case-insensitive property lookup in the Neo4j database, which turned out to be signiicantly slower than an exact-match lookup. For future runs of the NuGet.org Miner, this problem has been averted by converting all package names to lowercase.

Running our own Maven Miner implementation took additional 14 days, including the second phase of postprocessing. Our inal data for NuGet.org and the NPM Registry is up-to-date as of June 2020, while the data

11semver.org/

Digit. Threat. Res. Pract.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1:10 | • | Johannes DÃĳsing and Ben Hermann | | | |
| 15  10  5  0 | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | | | | | | |  | | | | | | |  | | | | | | |  | | | | |  | |  |  |  |  |  | | --- | --- | --- | --- | --- | | |  | | --- | |  | | |  | | --- | |  | | Mi | | |  | | --- | |  | | |  | | --- | |  | | Mi | | | | | | |  |  |  |  |  | 1 | 3 | 56 | | | | | | |  | | | | | l  ll | i  lio | n n | Libraries Artifacts |  |  |  |  |  |  |  |  |  |  | |  | | | | | Th | | |  |  | | --- | --- | | 22 | 4.04 | | | | | | |  |  |  | | | | | |  | | | | |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | | |
|  |  |  |  |



|  |  |  |
| --- | --- | --- |
| Analyzing Vulnerability Impact | • | 1:11 |

the duration between vulnerability disclosure and patch release is of interest, as well as a comparison of the two values.

**RQ2: What is the transitive impact of vulnerabilities?** We determine the number of artifacts that are afected by a given vulnerability in the irst place. While this is rather trivial to calculate for artifacts directly afected by a vulnerability, it is interesting to also identify artifacts that are transitively afected. It is especially relevant to compare these values across diferent repositories since the inluence of vulnerabilities is expected to vary. The results are likely to help us understand the net impact a vulnerability may have, depending on the respective repository.

**RQ3: How long does it take developers to upgrade vulnerable direct dependencies?** We analyze the arti-facts directly depending on a vulnerable artifact. Our overall goal is to understand when vulnerable dependencies are ixed by library developers, as compared to the vulnerability’s publication. It is also interesting to see whether or not these upgrade durations depend on the respective library, and how they difer across multiple repositories. The results may help library developers to understand the consequences of vulnerable dependencies, and other developers to carefully select dependencies for their project.

**RQ4: Do developers adhere to published security advisories?** We investigate whether or not developers use vulnerable dependencies, for which the corresponding security advisory has already been made public. The results indicate whether or not vulnerability publications are being monitored and considered during development.

5.2 Methodology

We extract diferent metrics from our data set in order to answer the RQs deined above. Besides those automated analyses, in some cases, we also require utilities for manual data exploration, which we implement using Python, Plotly12and Dash13.

We implement two diferent modes of operation regarding the detection of dependencies for all metrics that involve transitive relations. In the data set, a dependency is modeled using a node of type ArtifactReference, which is connected to all valid releases of the target library (as seen in Figure 2). Considering all of those targets as possible dependencies vastly increases the search space of the respective analysis, and does not correspond to reality, where a package manager picks exactly one of the valid target releases. Therefore, this mode of operation results in data that is an overapproximation (M>) of reality.

Another option for dependency detection is to select only one of the possible targets for any ArtifactReference. The most reasonable choice is to select the most recent release of the target library, as this is what any package manager does in the absence of version conlicts. Our analyses leverage the CURRENT\_TAR’ET relation annotated by the Reference Resolver to do so. This mode of operation yields an underapproximation (M<) of reality.

All of our analyses that consider transitive dependencies can be executed in both the M> and M< mode. However, since package managers always select one target per dependency, and only deviate from the most recent release in the rare case of version conlicts, we consider M< to yield data that is intuitively closer to reality. Therefore, all results presented in the following Section 5.3 have been obtained using this mode of operation.

5.3 Results

In this section, we present our analysis results for the aforementioned research questions. For each of them, we briely present the metrics involved and illustrate the results per repository using diferent types of diagrams.

12plotly.com/python/   
13plotly.com/dash/

Digit. Threat. Res. Pract.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1:12 | • | Johannes DÃĳsing and Ben Hermann | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  | | --- | --- | |  |  | | Low | |  |  | | --- | --- | |  |  | | Medium | |  |  | | --- | --- | |  |  | | High | |  |  | | --- | --- | |  |  | | Critical | | |
| |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  |  |  | | --- | --- | --- | --- | --- | | |  | | --- | |  | | |  | | --- | |  | | Unpatched Vulnerabilities [%] | | |  | | --- | |  | | |  | | --- | |  | | Wildcard Version References [%] | | | | | |
| 40% | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  | 4 | 9. | 5 |  | | | | |  | | | | |  | | | | 4 | 2. | 7 |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  | 5. | 5 | 20 | | | | |  | | | |  |  |  | 2 | 8.4 | 4 |  |  | | 2 | 2.1 | 1 |  |  |  |  |  | |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 17 |  |  |  |  | |  |  |  |  | 4.6 |  | |  |  |  |  |  |  | | | | | 100%  40%  20%   |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  |  |  |  | |  | | |  | | |  | | |  |  | |  |  | |  |  | |  | |  | |  | | |  |  | |  |  | |  |  | |  | |  |  | |  |  | |  |  | |  | |  | | |  |  | |  |  | |  |  | |  | |  | | |  |  |   80%  60%  0%  Maven NPM NuGet |
| 20% | |
| 0% | |
| Maven | NPM | NPM WSH | NuGet |

Fig. 5. Percentage of unpatched vulnerabilities and wildcard version range specifiers per repository

Fig. 6. Severity of unpatched vulnerabilities

**RQ1: How long does it take developers to release patches?** For this irst RQ, we analyze the time from a vulnerability’s publication or disclosure to the release of an appropriate patch by the developers of the target library. A previous study on the NPM Registry from 2018 [10] suggests that patches are often published before the date of publication, and even before the date of disclosure. We, therefore, expect that most patching durations are negative.

|  |  |
| --- | --- |
| 1 | MATCH (v:Vulnerability) -[: PATCHED\_BY]->(artifact) |
| 2 | WITH v.SnykId AS id, v.DisclosedAt AS disclosure , v.PublishedAt AS publication , artifact. |

|  |  |
| --- | --- |
|  | UniqueId AS artifact , artifact.releaseDate AS release |
| 3 | RETURN id, disclosure , publication , artifact , release; |

Listing 1. Cypher query to extract patching durations

For each vulnerability and patching artifact, our analysis extracts the duration from vulnerability publication to patch release (δP) and from vulnerability disclosure to patch release (δD) in seconds. This is done by issuing the Cypher query illustrated in Listing 1, and calculating the diferences between the datetime values release and dis / pub for each patching artifact. As this process is limited to vulnerabilities that have patches associated to them, a separate database query irst retrieves the number of unpatched vulnerabilities for each repository.

The bar chart in Figure 5 illustrates the number of unpatched vulnerabilities per repository. In addition to that, the percentage of vulnerabilities with wildcard version range speciiers is shown. These vulnerabilities by deinition cannot be patched by any artifact, as they reference the entirety of their target library, for example with a version range qualiier of "\*" or "[0,]".

While on average 35.8% of all vulnerabilities are not patched, NPM has by far the highest amount with a percentage of almost 50%. Upon further investigation, we found that this is partially due to so-called Security Holders, which we found to be referenced by 14% of all NPM vulnerabilities. Security Holders are symbolic artifacts that act as a placeholder for library identiiers that have previously been in use and are not usable anymore for security reasons [43]. Figure 5 shows the values for NPM when excluding security holders as NPM WSH (without security holders).

Digit. Threat. Res. Pract.



|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | |  | **Maven** | **NPM** | **Nuget** | | 100% | Analyzing Vulnerability Impact | | | | • | 1:13 |
| |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  | | --- | --- | |  |  | | δD ≤ 0 | |  |  | | --- | --- | |  |  | | δD > 0 > δP | |  |  | | --- | --- | |  |  | | δP ≥ 0 | | | | | | |
| |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  |  | |  |  | |  |  | |  |  | |  | |  | | |  |  | |  |  | |  |  | |  |  | |  | |  | | |  | |  | | |  |  | |  |  | |  |  | |  |  | |  | |  | | |  | | |  |  | |  |  | |  |  | |  |  | |  | |  | | |  |  | |  |  | |  |  | |  |  | |  | | | | | |
| 80% |
| |  |  |  |  | | --- | --- | --- | --- | | **Mean** δD | 73.1 | 7.9 | −3.7 | | **Median** δD | −0.6 | 0.8 | 0.7 | | **IQR** δD | 184.0 | 25.4 | 3.8 | |
| 60% |
| 40% |
| |  |  |  |  | | --- | --- | --- | --- | | **Mean** δP | −188.2 | −159.8 | −209.8 | | **Median** δP | −28.2 | −7.9 | −30.3 | | **IQR** δP | 311.8 | 143.5 | 395.8 | |
| 20% |
| 0% |
| Table 1. Key measures for δP and δD in days | Maven | NPM | Nuget | Total | |

Fig. 7. Distribution of δP and δD values per repository

In Figure 6, we illustrate the distribution of severities for those unpatched vulnerabilities. While we initially expected those vulnerabilities to be of minor severity, our data indicates that in fact more than 50% are either of severity high or critical across all three repositories. This is mainly inluenced by the NPM repository, in which more than 70% of all unpatched vulnerabilities have a high or critical severity. For Nuget and Maven, we observed a behavior that is more in line with our expectations, as for both of them around 66% of the unpatched vulnerabilities are either of severity low or medium.

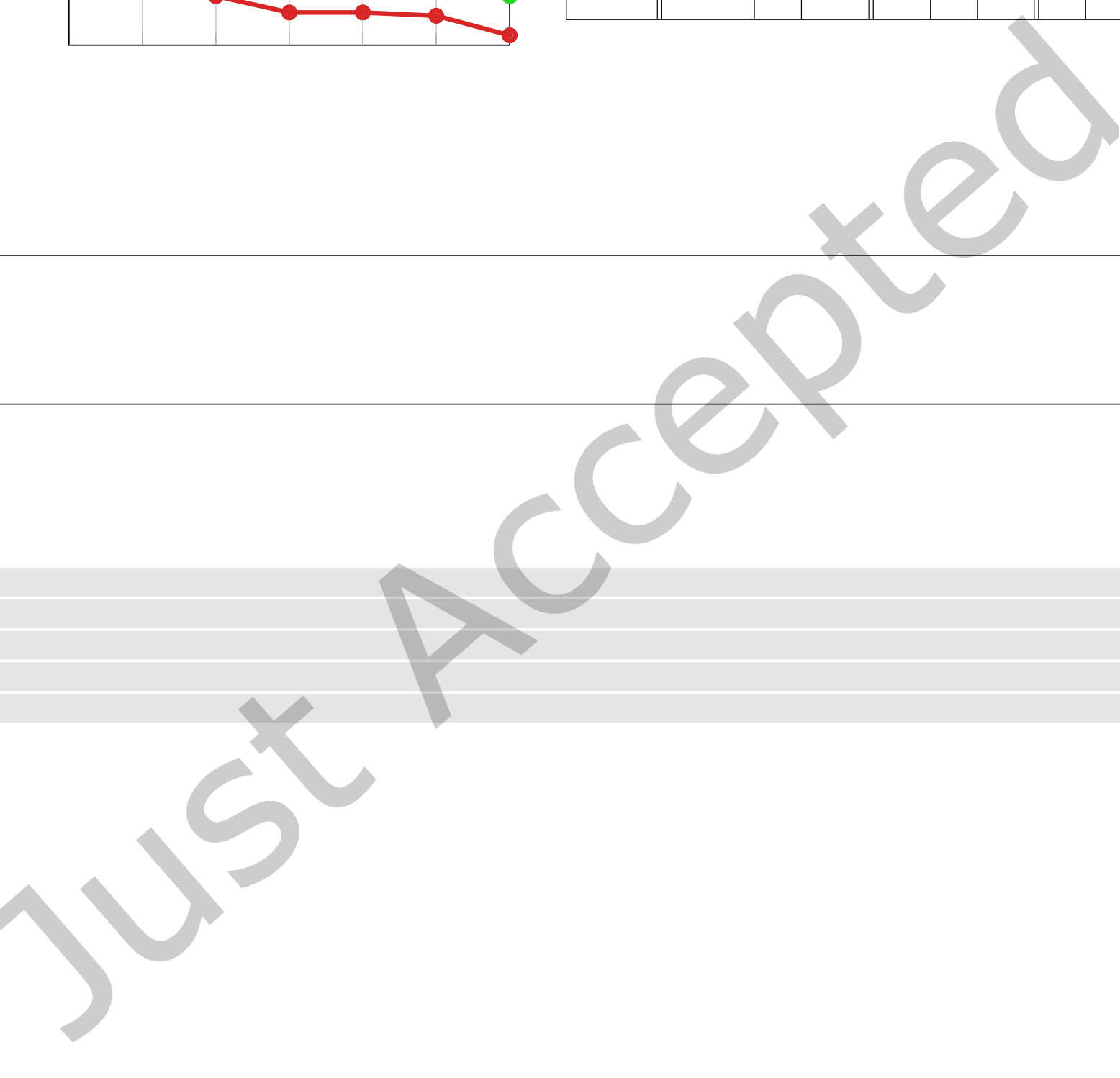
For the remainder of this section, we only consider vulnerabilities with existing patches for our analysis. In Table 1, we present the mean and median values for δD and δP, as well as the respective Interquartile Range (IQR) in days. Regarding δP, we observe that vulnerabilities in NuGet.org seem to be patched the earliest, followed by Maven Central and then the NPM Registry. For δD we observe very similar median values for all three repositories, however, there are large diferences in the IQR values, indicating a much more skewed distribution for Maven Central (184.0) as compared to the NPM Registry (25.4) or NuGet.org (3.8).

In general, in Table 1 we observe rather large IQR values, especially for δP. In order to gain more meaningful insights, we plot a stacked histogram of both the δD and δP values in Figure 7. For each repository, the chart indicates the amount of patches released before vulnerability disclosure (δD ≤ 0), between disclosure and publication (δD > 0 > δP) and after publication (δP ≥ 0).

We observed that, in total, almost 50% of all vulnerabilities are patched before their date of disclosure. This suggests that library developers employ vulnerability detection processes, and manage to signiicantly reduce the attack surface of their library before any external actor informs them about a particular threat. Furthermore, around 73% of all vulnerabilities are patched before their publication, meaning that without any insider knowledge, an attacker may only be able to exploit the remaining 27% for an attack. In addition to that, our data suggests that an additional 7% of vulnerabilities are patched within the irst month after publication.

While this average data does match our general expectation of the patch release date distribution, there are some notable diferences between repositories. While for Maven Central over 50% of all vulnerabilities are patched before their disclosure, the corresponding value for NuGet.org is only 33%. However, with 86% being patched before publication, NuGet.org outperforms Maven Central, which is below 75% for this value. This suggests that maintainers of the NuGet.org repository rely on vulnerability disclosure to a larger degree than their colleagues at Maven Central, but are still able to patch a larger amount of vulnerabilities before they are made public.

Digit. Threat. Res. Pract.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1:14 | • | Johannes DÃĳsing and Ben Hermann | | | | | 4 | Maven | 6 | **Maven Central** | | | | | **NPM Registry** | | | | **NuGet.org** | | |
| Amount of vulnerabilities | 100% | | | | |
| 80% | | | | | NPM |
| Nuget | X | | ˜X | | IQR | X | | ˜X | IQR | X | ˜X | IQR |
| 60% | | | | |
| **Level 0** | 0.5 | 0.2 | | 0.5 | 0.3 | | 0.1 | 0.3 | 0.8 | 0.7 | 0.9 |
| **Level 1** | 88.5 | 1.2 | | 18.4 | 26.9 | | 0.0 | 2.2 | 6.9 | 0.0 | 0.0 |
| 40% | | | | | 5 | **Level 2** | 587.2 | 1.4 | | 75.8 | 35.9 | | 0.0 | 1.0 | 8.6 | 0.0 | 0.0 |
| **Level 3** | 1,878.0 | 1.0 | | 190.7 | | 31.9 | 0.0 | 0.2 | 8.3 | 0.0 | 0.0 |
| **Level 4** | 4,200.0 | 0.5 | | 353.8 | | 25.8 | 0.0 | 0.0 | 9.3 | 0.0 | 0.0 |
| 20% | | | | |
| **Level 5** | 7,649.3 | 0.2 | | 444.6 | | 20.2 | 0.0 | 0.0 | 4.1 | 0.0 | 0.0 |
| **Level 6** | 11,645.9 | | 0.0 | 634.3 | | 15.5 | 0.0 | 0.0 | 2.0 | 0.0 | 0.0 |
| 0% | 0 | 1 | 2 | 3 | Table 2. Mean, median and IQC values for IArt,n per 100,000 | | | | | | | | | | | |
| total artifacts for each level and repository | | | | | | | | | | | |

Level

Fig. 8. Amount of vulnerabilities with impact per level

**Over 25% of all vulnerabilities are unpatched, with the majority being of high or critical severity. The main driver for both trends is the NPM Registry repository. Almost 75% of all patches are released be-fore vulnerability publication, on average this happens 184 days prior. NuGet.org patches are released the earliest compared to vulnerability publication.**

**RQ2: What is the transitive impact of vulnerabilities?** We inspect the net impact of vulnerabilities by measuring the number of artifacts that are afected, both directly and transitively. As part of this, we also analyze the length of transitive dependency chains that propagate such vulnerabilities.

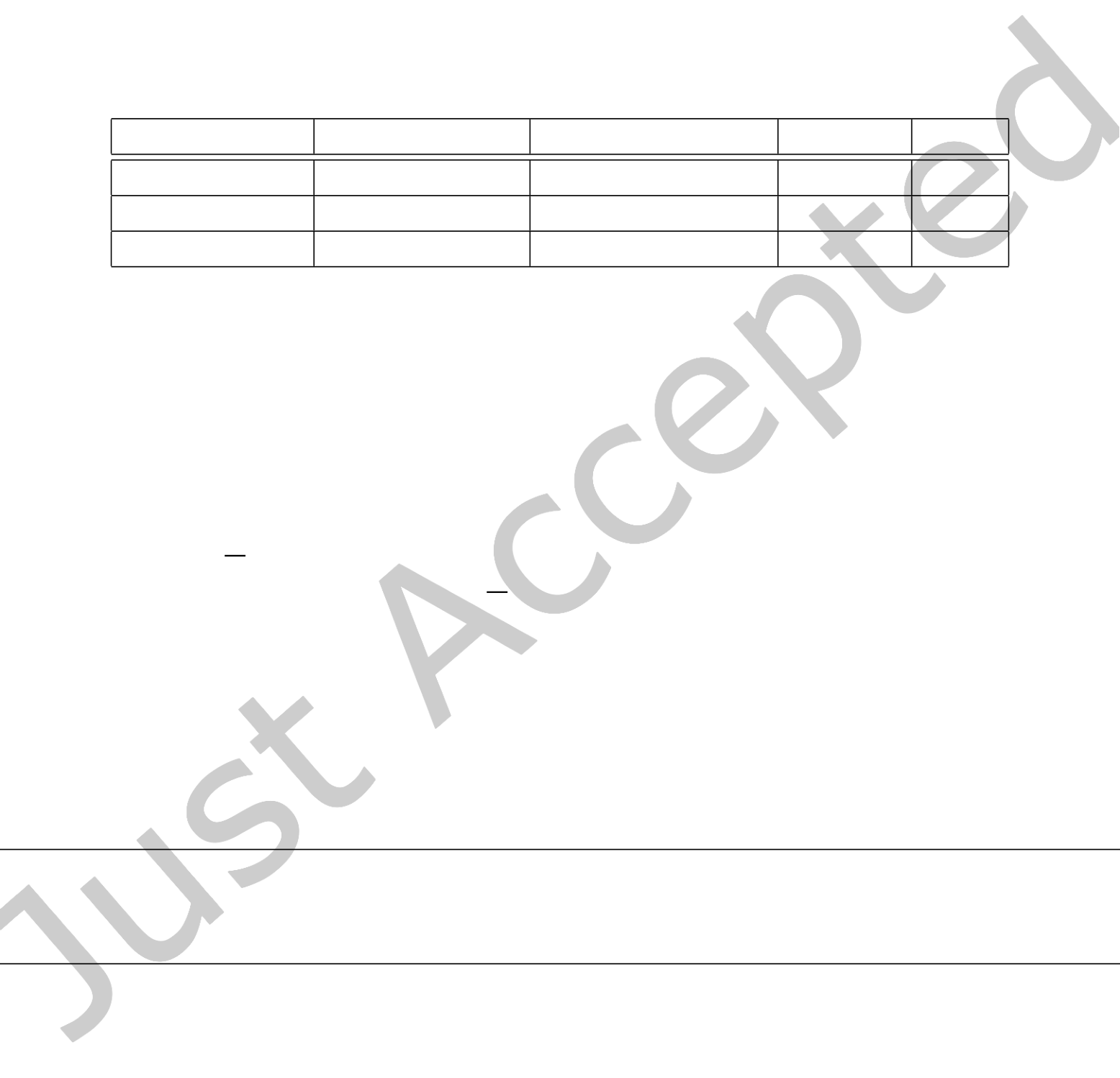
1 MATCH (v:Vulnerability {snyk\_id:'DUMMY -ID'}) -[:AFFECTS|REFERENCES \*2]->(l0:Artifact) 2 MATCH p = (l0) <-[:DEPENDS\_ON \*0..6] -(a:Artifact)   
3 WITH length(p) AS depth , a.LibraryId AS lib , a.UniqueId AS uid   
4 WITH depth , count(DISTINCT lib) AS lib\_count , count(DISTINCT uid) AS art\_count 5 RETURN depth , lib\_count , art\_count

Listing 2. Cypher query used to obtain values IArt,n and ILib,n for a single vulnerability

In order to do so, we query our data set and for each vulnerability extract the number of unique afected libraries (ILib,n) and unique afected artifacts (IArt,n) per level n, where the level captures the length of the transitive chain. IArt,0(v) therefore corresponds to the number of artifacts directly afected by vulnerability v, whereas IArt,1(v) represents the number of artifacts directly depending on a level zero artifact for the same vulnerability. As described in Section 5.2, the detection of dependencies is executed in an underapproximation mode (M<). Due to performance reasons, we cut of our analysis at a depth of six. Listing 2 presents an example query that is used to obtain values for a single vulnerability in the Maven Central dependency graph, with the cutof being speciied as the recursive multiplicity of the DEPENDS\_ON relation in line 2.

In Figure 8 we illustrate the transitive impact of vulnerabilities. We say a vulnerability v has impact on level n if it afects at least one library on that level, ie. ILib,n(v) > 0. We observe that vulnerabilities in the Maven Central Repository have the most impact on the irst six levels. Here, more than 70% of all vulnerabilities have an impact on the irst level, and roughly 50% have an impact on the sixth level. For the NPM Registry these values are signiicantly lower. The former one being not more than 60% and the latter just over 15%. Finally, vulnerabilities in NuGet.org exhibit the least impact of all three repositories, merely 3% have an impact on level six.

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| Analyzing Vulnerability Impact | • | 1:15 |

As we cut of our automated analysis at level six, it does not reveal any information about the total length of impact chains in the data set. Therefore, we conducted a structured manual analysis on the most impactful vulnerabilities, which are likely to exhibit the longest impact chains. However, it must be noted that there may be longer chains for less impactful vulnerabilities, therefore our results may only be seen as an indication for the general trend in terms of impact chain length. Table 3 reports the maximum chain length LMAX that we observed during manual inspection for each repository. The longest chain is comprised of 38 artifacts and is part of the Maven Central repository. The impact chains for the NPM Registry and NuGet.org proved to be shorter, here we observed maximum lengths of 15 and 13, respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Repository** | **Target Library** | **CVE Identiier** | **Severity** | L**MAX** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Maven Central | log4j:log4j | CVE-2019-17571 | Critical | 38 |
| NPM Registry | lodash | CVE-2019-1010266 | Medium | 15 |
| NuGet.org | log4net | CVE-2018-1285 | High | 13 |

Table 3. Vulnerability with maximum impact chain length seen during manual analysis

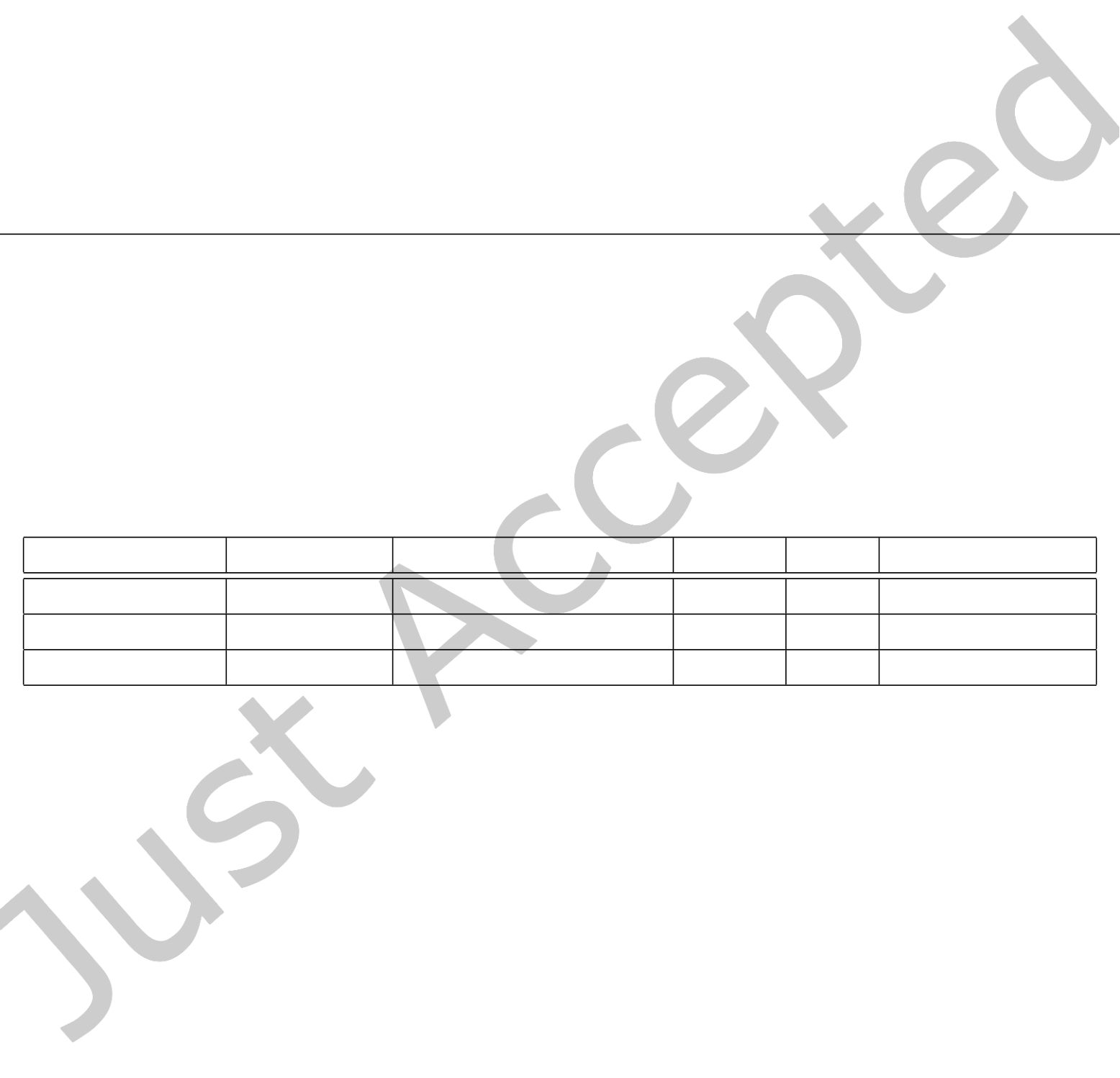
Apart from the length of impact chains, we also investigate the IArt,n values for each level n. In order to enable a comparison between repositories, we normalize those values per 100,000 total artifacts for each repository. Table 2 presents the mean (X) and median (˜X) number of afected artifacts per level, as well as the corresponding IQR. Interestingly, we observe turning points for X at levels two and four for the NPM Registry and NuGet.org, while the values for Maven Central exhibit continuous growth until level six. This matches both our observation of very long impact chains in Maven Central (cf. Table 3) and the large number of Maven vulnerabilities with impact on level six or higher (cf. Figure 8).

However, the direct comparison between Maven Central and NuGet.org reveals a stark contrast in vulnerability impact: While in NuGet.org on average only 2 artifacts per 100,000 are afected by vulnerabilities on level 6, the corresponding values for Maven Central is almost 6,000 times larger.

|  |
| --- |
| **Vulnerabilities in the Maven Central repository afect the most artifacts and do so via the longest tran-sitive dependency chains. On the other hand, NuGet.org vulnerabilities have the shortest transitive impact chains, and afect less than 0.02% the number of artifacts compared to Maven.** |

**RQ3: How long does it take developers to upgrade vulnerable direct dependencies?** We analyze how long it takes library maintainers to upgrade a vulnerable direct dependency, as compared to the vulnerability’s publication. We are especially interested in inding out whether the publication of vulnerabilities triggers an immediate response from library maintainers. Since maintainers are generally not involved in the disclosure process, we do not compare the time of upgrades to vulnerability disclosure dates here. Furthermore, we present information on the relative number of afected libraries per repository and investigate the upgrade ratio for such libraries.

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| 1:16 | • | Johannes DÃĳsing and Ben Hermann |

**Algorithm 1** UpgradeDurationAnalysis

1: **for** v ← VulnerabilitiesWithPatch **do**

|  |  |
| --- | --- |
| 2:  3:  4:  5:  6:  7:  8:  9:  10:  11:  12:  13: | AfectedLibs ← getLibrariesUsingArtifactAfectedBy(v) **for** lib ← AfectedLibs **do**   Report ← LibraryReports.get(lib)   Report.nV D += 1   FirstNotAfected ← getFirstReleaseNotAfectedBy(lib, v)  **if** FirstNotAfected != None **then**   δPU ← FirstNotAfected.ReleaseDate - v.PublishedAt Report.updateMeanDuration(δPU )  **else**   Report.nU D += 1  **end if**  **end for** |

14: **end for**

For each unique library identiier, our analysis collects the number of total vulnerable dependencies (nVD), as well as the number of vulnerable dependencies that have not been upgraded for this library, which we call unpatched dependencies (nUD). Based on that, we deine the patching ratio r P for each library as r P = 1 −nUD nVD. Furthermore, for every library that has at least one patched vulnerable dependency, we extract the mean duration between vulnerability publication and dependency upgrade, δPU, in seconds. This methodology is illustrated in Algorithm 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Repository** | **# Libraries** | **Relative # Libraries** | E[n**VD**] | E[r **P**] | E[δ**PU**] **in Days** |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Maven Central | 99,162 | | 28,97% | 8.7 | 38% | -192 |
| |  |  | | --- | --- | | NuGet.org | 3,144 | | 1.16% | 2.8 | 40% | -212 |
| |  |  | | --- | --- | | NPM Registry | 102,666 | | 7.96% | 3.4 | 32% | -503 |

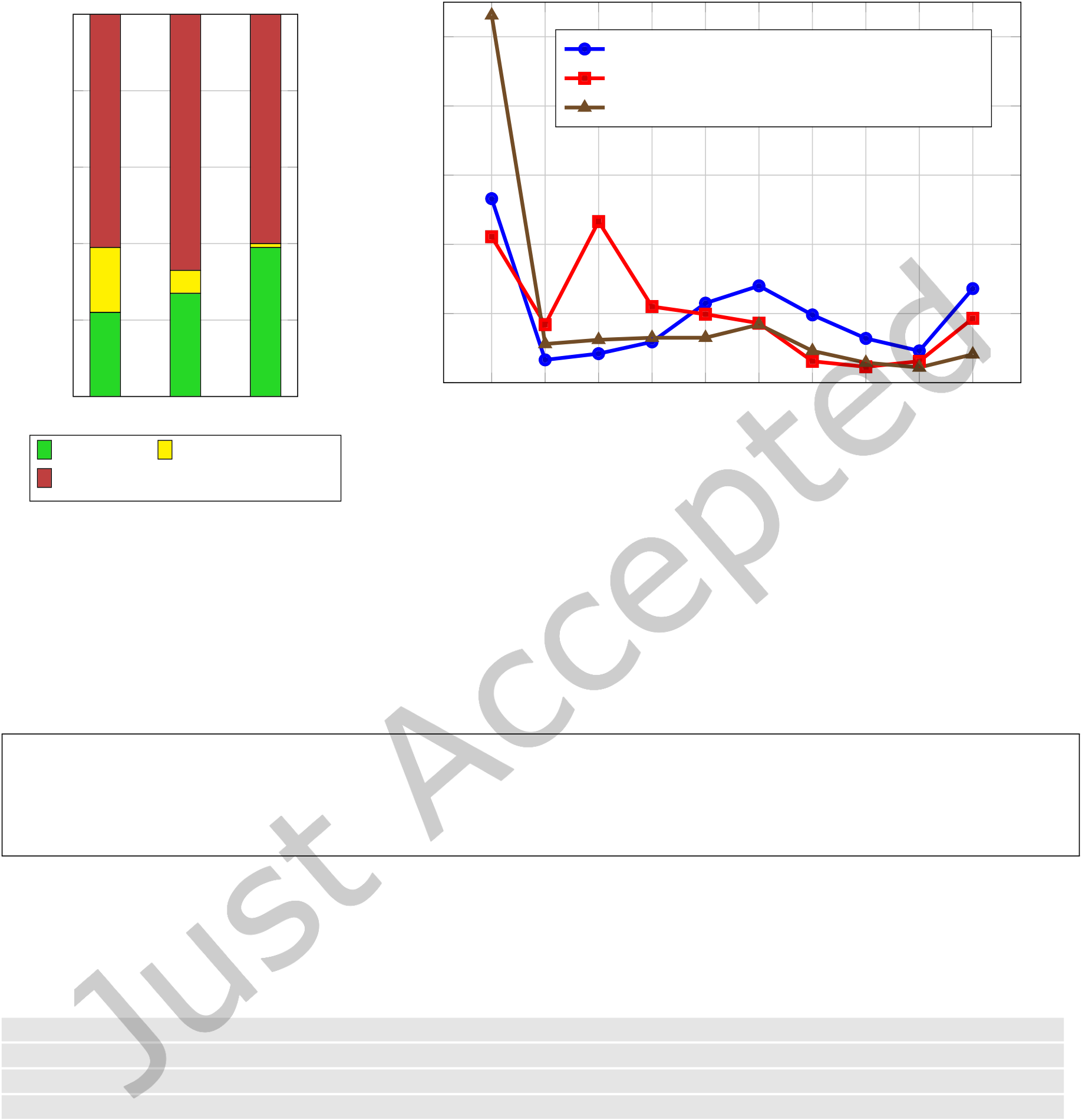
Table 4. Key characteristics of the analysis results for RQ3 per repository

Table 4 summarizes the main characteristics of our results for this RQ. It contains not only the total but also the relative amount of libraries that have ever been afected by direct vulnerable dependencies. Here, we observe signiicant diferences between repositories: While in Maven Central more than a fourth of all libraries are afected, in NuGet.org this value is just above 1%. In addition to that, Maven’s average nV D value of 8.7 is notably larger than the values for the NPM Registry (3.4) and NuGet.org (2.8). Despite these diferences, the average values for r P are rather similar for all three repositories, with values ranging from 32% to 40%. On average, vulnerable dependencies are patched up to 503 days before vulnerability publication (NPM Registry).

Upon investigating the distribution of patching rates, which is illustrated in Figure 9, we made an interesting observation: Library maintainers tend to either upgrade all vulnerable dependencies (rP = 100%), or none of them (rP = 0%). We observed partial upgrades for only 17% (Maven Central) to 1% (NuGet.org) of all libraries.

The distribution of all δPU values is plotted in Figure 10. As we expected, most values for the NPM Registry are below -400 days, which matches the low average reported in Table 4. However, in Maven Central and NuGet.org between 10% and 13% of all values are larger than 400 days, which indicates that a substantial amount of vulnerable dependencies is upgraded more than a year after the corresponding security advisory was published.

Digit. Threat. Res. Pract.



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 100% | | | Amount of total values in interval | Analyzing Vulnerability Impact | | • | 1:17 |
| 80% | | | 50% | δPU Distribution for Maven Central |
| δPU Distribution for NuGet.org | |
| 40% | δPU Distribution for the NPM Registry |
| 60% | | | 30% | |
| 40% | | | 20% | |
| 20% | | | 10% | |
| 0% | Maven | NPM NuGet.org | 0% | |
| r P = 100% r P = 0% | | 100% > r P > 0% | Fig. 10. Distribution of δPF values for each repository in days | |
| Fig. 9. Distribution of rP values | | |

Interestingly, both Maven Central and NPM Registry exhibit a slight peak in the [0; 100) interval, which may indicate that the publication of a vulnerability prompts some developers to immediately upgrade their dependencies. Another reason for this observation may be the use of automated tools for vulnerability detection, as mentioned in Section 2.

**Vulnerable dependencies afect only 1% of all libraries in NuGet.org, but up to 29% in Maven Cen-tral. Upgrades of vulnerable dependencies usually happen more than 200 days prior to vulnerability publication. For some libraries, vulnerability publication seems to trigger an immediate dependency upgrade, suggesting the use of automation.**

**RQ4: Do developers adhere to published security advisories?** Once published, a vulnerability and all artifacts afected by it are publicly accessible by any software developer. We inspect whether or not developers adhere to those public security advisories. In order to do so, we calculate the number of artifacts that use a vulnerable dependency but were released after the vulnerability’s publication.

1 MATCH (v:Vulnerability) -[:AFFECTS|REFERENCES \*2]->(: Artifact) <-[:DEPENDS\_ON ]-(l1:Artifact)

2 WITH v.PublishedAt AS publication , l1.UniqueId AS artifact , l1.releaseDate AS release

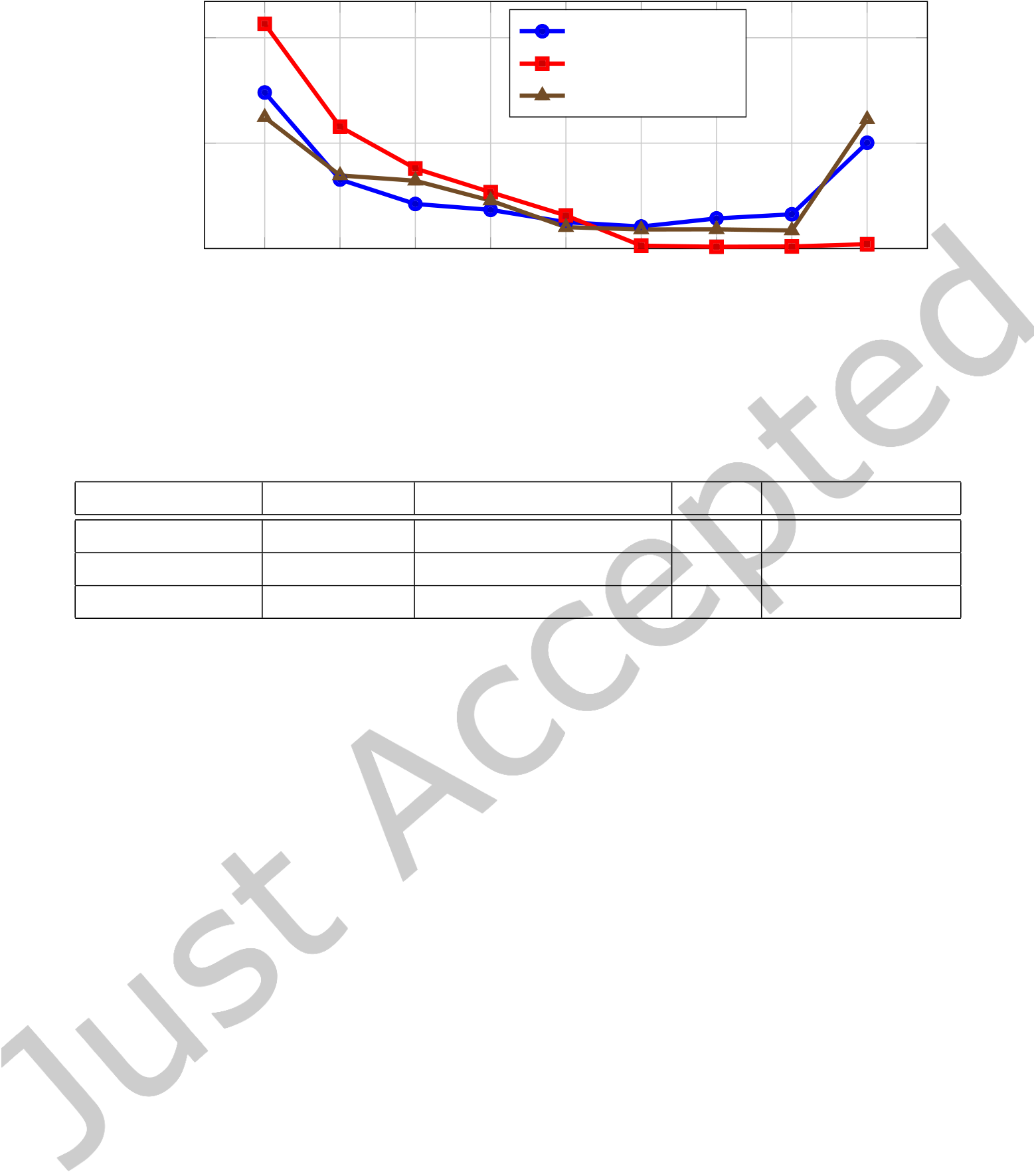
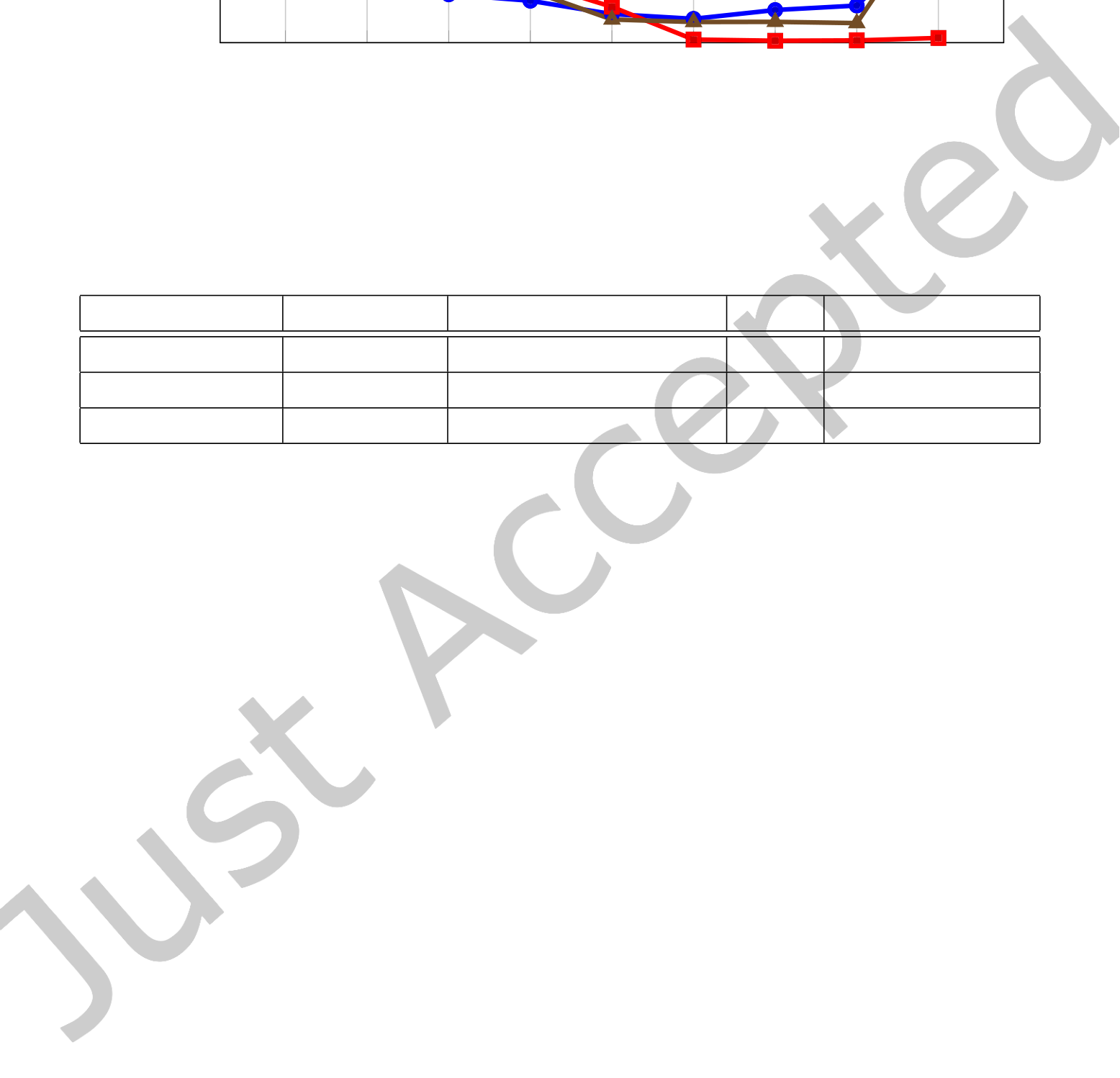
3 WHERE release > publication

4 RETURN DISTINCT artifact , publication , release;

Listing 3. Cypher query used to obtain data for RQ4

For each such artifact, we compute the number of vulnerable dependencies, for which the respective vulnera-bility had already been published at the time of artifact release (nV). Furthermore, for each artifact we calculate the minimum duration for which those vulnerabilities have been publicly accessible at that time in seconds, which we call δPR. We collect the raw data for this analysis by executing the Cypher query presented in Listing 3.

Digit. Threat. Res. Pract.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1:18 | • | Johannes DÃĳsing and Ben Hermann | | Maven Central |
| Amount of total values in interval | 40%  20%  0% |
| NuGet.org |
| NPM Registry |

Fig. 11. Distribution of durations δPR in months

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Repository** | **# Artifacts** | **Relative # Artifacts** | E[n**V**] | E[δ**PR**] **in Days** |
| Maven Central | 807,200 | 15,46% |
| 3.3 | 470 |
| NuGet.org | 15,119 | 0.50% | 1.3 | 155 |
| NPM Registry | 2,203,639 | 16.25% | 2.8 | 431 |

Table 5. Results of RQ4 per repository

Similar to RQ3, Table 5 presents the total and the relative number of afected artifacts per repository. Fur-thermore, the table holds the average values for nV and δPR. Here, we observe signiicant diferences between repositories. While in NuGet.org only 0.5% of all artifacts are afected, in the NPM Registry this value is more than 32 times higher. In addition to that, afected artifacts in NuGet.org on average have 1.3 vulnerable dependencies with published advisories, compared to 3.3 and 2.8 vulnerable dependencies per artifact in Maven Central and the NPM Registry, respectively.

The average duration between vulnerability publication and artifact release ranges from 155 to 470 days, which implies that some maintainers use dependencies known to be vulnerable for well over a year. We further investigate the distribution of those δPR values in Figure 11.

The distribution of values in the NuGet.org repository most closely matches our general expectations. Here, more than 90% of all artifacts have been released no longer than one year after vulnerability publication, and only 0.8% after more than two years. In contrast to that, over 20% of all δPR values for both Maven Central and the NPM Registry are greater than two years. Compared to NuGet.org, the values for both repositories are distributed more evenly. This may indicate that in NuGet.org, compared to the other two repositories, security advisories have a more direct inluence on dependency upgrades.

|  |
| --- |
| **In the NPM Registry, 16.25% of all artifacts are released with dependencies for which public security advisories are available. This phenomenon also afects Maven Central (15.46%) but has a lesser im-pact on NuGet.org (0.5%). On average, afected artifacts have between 1.3 (NuGet.org) and 3.3 (Maven Central) such dependencies, which advisories being published for up to 470 days (Maven Central).** |

Digit. Threat. Res. Pract.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | DISCUSSION | Analyzing Vulnerability Impact | • | 1:19 |

In this section, we discuss our analysis results and their implications. We start by pointing out security risks that result from our indings and go on to derive possible mitigation strategies that may improve the current situation.

6.1 Security Risks

Based on our analysis results presented in Section 5.3, we identify a set of potential threats for software security. We present this set, which we name T, in Table 6. In the following paragraphs, we highlight the core issues of each member of the set T, and present empirical evidence found in the analysis results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Id** | **Name** | **Maven Central** | **NPM Registry** | **NuGet.org** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| T1 | Patch Release after Vulnerability Publication | Yes | Yes | Yes |
| T2 | Excessive Transitive Vulnerability Impact | Yes | No | No |
| T3 | Libraries Keep Vulnerable Dependencies | Yes | Yes | Yes |
| T4 | Publication has no Impact on Upgrade Duration | No | No | Yes |
| T5 | New Releases have Vulnerable Dependencies | Yes | Yes | No |

Table 6. Set of all threats T with repositories they apply to

As part of our analysis for RQ1, we found that while most patches are released before vulnerability publication, a non-negligible amount of over 25% is in fact released afterwards (T1). This implies that for a certain period of time, software libraries were publicly known to be vulnerable, making them a prime target for exploitation. This threat is most pronounced for Maven Central (29%) and the NPM Registry (23%), but also applies to NuGet.org (13%).

The propagation of vulnerabilities along transitive dependency chains leads to an exponentially growing number of artifacts potentially afected and makes it increasingly hard to decide whether vulnerabilities apply to a speciic software artifact (T2). In our results for RQ2, we found that vulnerability impact chains in Maven Central are substantially longer than in the other two repositories. This is true both on average (cf. Figure 8) and at maximum (cf. Table 3). Furthermore, the amount of Maven artifacts transitively afected by a vulnerability is over 750 times higher than for the NPM Registry, and almost 6,000 times higher than for NuGet.org.

As part of our results for RQ3, we found that between 1.16% (NuGet.org) and 28.97% (Maven Central) of all repository libraries have at one point been afected by vulnerable direct dependencies. A further classiication discovered that, out of all afected libraries, between 60% and 66% did not upgrade any of those vulnerable dependencies at all (cf. Figure 9). As a result, from 0.7% (NuGet.org) up to 17.4% (Maven Central) of all repository libraries have potentially multiple vulnerable dependencies that have not been upgraded. This fact poses a major security threat (T3), as consumers of those libraries may not be aware of the vulnerable dependencies, and do not expect the latest release to be vulnerable.

When analyzing the distribution of upgrade durations for RQ3, we found that the publication of a vulnerability seems to trigger an immediate dependency upgrade for some libraries in Maven Central and the NPM Registry (cf. Figure 10). This indicates that some library maintainers may be actively monitoring vulnerability publications, and respond if needed. However, we found no evidence for the same kind of behavior in the NuGet.org repository, meaning that vulnerable dependencies may afect artifacts longer than necessary (T4).

During the previous analysis for RQ4 we found that some software artifacts use vulnerable dependencies, for which the respective vulnerability has been published before the artifact release. This implies that developers did either not consult or deliberately ignore public security advisories like the CVE list. As a result, consumers of the corresponding repositories can not even rely on new releases of a library to be vulnerability-free, which may

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| 1:20 | • | Johannes DÃĳsing and Ben Hermann |

lead to a false sense of security when upgrading dependencies to the latest release (T5). According to Table 5, this issue afects artifacts in all three repositories, but is most pronounced for the NPM Registry (16.25% of all artifacts) and Maven Central (15.46%), while only afecting 0.5% of all artifacts in NuGet.org.

6.2 Mitigation Strategies

As a inal contribution of our work, in this section, we present a set of mitigation strategies for maintainers and consumers of all three repositories, which aim to mitigate the threats presented in the previous section. Table 7 presents the set of strategies, which we call S, alongside the corresponding threats and the target audience. Most of the strategies presented here are well-known in software engineering research. However, here we provide justiication for their usefulness, as we link them to threats that we observed based on empirical indings in our dataset.

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Threats** | **Audience** |

|  |  |  |  |
| --- | --- | --- | --- |
| S1 | Closely Monitor Vulnerability Publications | T1, T4 | Maintainers |
| S2 | Employ Bug Bounty Programs | T1 | Maintainers |
| S3 | Use Tools for Dependency Monitoring | T2, T3, T5 | Maintainers & Consumers |
| S4 | Check for Vulnerable Dependencies on Release | T5 | Maintainers |
| S5 | Minimize the Number of Dependencies | T2 | Maintainer & Consumers |

Table 7. Set S containing mitigation strategies derived from T

Threat T1 refers to a substantial amount of library maintainers releasing patches a long time after the cor-responding vulnerability has been published. Furthermore, T4 points out that maintainers of libraries with vulnerable dependencies do not act upon the vulnerability’s publication. As a result, there are artifacts that are subject to publicly known vulnerabilities without a patch being available in all three repositories. An easy way of mitigating this risk is to closely and actively monitor the publication of vulnerabilities (S1), so that maintainers can immediately upgrade the respective dependency or start implementing a patch, thus reducing the time for which an artifact is vulnerable. There are multiple ways to automatically monitor new vulnerability publications. The NVD provides an RSS feed on the latest vulnerabilities that have been analyzed14, and the MITRE corporation does the same on their Twitter feed15. Developers may also be informed about vulnerabilities in dependencies by the use of tools for dependency monitoring, which are regularly updated with the latest set of vulnerabilities. Examples include the services provided by Snyk, as well as the npm audit command.

While adhering to strategy S1 reduces the time of exposure to published vulnerabilities signiicantly, it is still likely to take library maintainers a couple of days or weeks to develop a patch and release it. In an efort to further reduce this time, and therefore minimize the risk imposed by threat T1, maintainers may encourage members of the open-source community to ind and report security vulnerabilities on their own, thus increasing the chances of discovering vulnerabilities before their oicial publication (S2). Such initiatives are often called Bug Bounty Programs and are already ofered by a variety of corporations and institutions, including Microsoft [27] and the U.S. Pentagon [17]. While for large corporations these programs often include incentives like monetary rewards, smaller groups of library maintainers may still be able to ofer some sort of reward, which may even be an honorable mention in the library description. It must be noted that both S1 and S2 can only be applied for libraries that are still being actively maintained, as opposed to libraries that have been discontinued.

14nvd.nist.gov/vuln/data-feeds#RSS   
15twitter.com/CVEnew

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| Analyzing Vulnerability Impact | • | 1:21 |

Strategy S1 already suggested the usage of automated tools for dependency monitoring, which helps library maintainers monitor vulnerability publications. In fact, this strategy also applies to regular repository consumers and has the potential to mitigate several risks identiied in the previous section (S3). As those tools can often be integrated into a CI worklow, they inform library maintainers about vulnerable dependencies on every commit, making it efectively impossible to not be aware of their existence. Therefore, the risks imposed by T2, T3 and T5 are reduced substantially. Furthermore, the usage of automated tools for detecting vulnerable dependencies may result in repository consumers preferring more secure libraries for their projects. The more consumers show an obvious interest in security, the more likely library developers are to shift their development focus on security-related aspects.

Due to the larger amount of artifacts being released despite their dependencies being afected by published vulnerabilities, as implied by T5, strategy S4 encourages library maintainers to inspect all their library’s dependen-cies on every release. By establishing a deined process of doing so, the risk of unintentionally using vulnerable dependencies is efectively eliminated. Ideally, this process is automated by making use of similar tools to the ones discussed for S3. While maintainers may still intentionally decide to keep vulnerabilities for a given release, this guideline assures that at least one developer has evaluated the resulting risk and decided that it does not have a critical impact on library consumers.

The impact of threat T2 is directly correlated to the density of the corresponding dependency graph, and therefore to the number of dependencies per artifact. Consequently, the impact may be reduced by reducing the average number of dependencies for each software library (S5). A similar argument can be made for repository consumers, where the number of potential security threats is directly linked to the number of project dependencies. While it is a rather simple task to remove redundant dependencies from an application or library, replacing those that are actually used can only be achieved by implementing the desired functionality from scratch. However, this would be in stark contrast to the concept of software reuse, which has been proven to be efective. Nevertheless, especially in the NPM Registry, some of the libraries are small enough to be replaced by a single function deinition. Examples include libraries like array-first, which provides a single function for returning the irst n elements of an array, or to-capital-case, which capitalizes a string value. These examples illustrate that in some cases a trade-of between software reuse and potential security risks must be considered. This has also been observed in a 2020 study on the use of trivial packages by Abdalkareem et al. [1], where the authors observe that trivial packages make up between 10.5% and 16% of PyPI and the Npm Registry, despite the fact that up to 72% of such packages to not incorporate any tests. Similarly, Soto-Valero et al. ind that many dependencies in public software artifacts are actually unused, and present a tool called DepClean to tackle this issue [41].

7 THREATS TO VALIDITY

In this section, we present possible threats to the validity of our work, and how we chose to mitigate them as much as possible.

Construct Validity. In our analyses for RQ2, RQ3, and RQ4, we investigate metrics that involve vulnerabilities afecting (transitive) dependencies of a software artifact. It must be noted, that the mere presence of such a vulnerable dependency does not imply that the vulnerability applies to the artifact. In fact, researchers associated with the FASTEN Project propose to use method-level call graph analysis to decide whether or not vulnerabilities in dependencies actually apply to a project [31]. They argue that analyzing vulnerabilities on the package-level leads to false positives, and therefore wastes development resources, which is also supported by the indings of an empirical study conducted by Zapata et al. in 2018 [13]. However, the indings of this project are still preliminary and have yet to be implemented for entire software ecosystems [6]. On the other hand, sometimes vulnerabilities in dependencies can be exploited simply because the corresponding code is on the classpath, as shown by Lawrence and Frohof [21]. As our analysis handles vulnerabilities on the package level, we note that

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| 1:22 | • | Johannes DÃĳsing and Ben Hermann |

our indings generally have to be considered an overapproximation, which may be reined by incorporating precise call graph information for each artifact in the future.

Internal Validity. As mentioned in Section 5.2, our algorithm for dependency resolving supports two modes of execution, yielding either all valid targets (overapproximation) or the most recent valid target (underapproxi-mation). As package managers only refrain from using the most recent target in case of version conlicts, our analysis results have been obtained using the latter. Therefore, there might be some dependency paths that have not been counted towards our results for RQ2, RQ3, and RQ4.

For the threats presented in Section 7 we often assume that not being aware of vulnerabilities in dependencies is a major reason for not performing an upgrade or not releasing a patch. However, there may in fact be other reasons involved in this decision, for example, libraries may not prioritize ixes, or maintainers may decide that vulnerabilities do not apply to their speciic usage scenario. Further investigations need to be performed in order to identify those reasons and their relevance in this context.

We rely on the correctness of the Synk dataset we use throughout the analysis. As the dataset is manually veriied in a structured process and includes oicial data from the NVD, we think we can safely assume the data to be correct.

External Validity. We used vulnerability data that is the intellectual property of Snyk Ltd. and can, consequently, not be published. Therefore, others can not reproduce our analysis, as we are only able to make our tools for generating dependency graphs publicly available. We chose this approach as the data provided by Snyk both contains more vulnerabilities than sources like the NVD, and is manually veriied to be correct. We strive to include open vulnerability sources and repeat our analysis in the future. In the meantime, interested researchs may of course obtain the vulnerability data through the same oicial channels we used.

Conclusion Validity. As many of our analysis metrics imply, artifacts from the NuGet.org repository seem to be somehow less afected by vulnerabilities compared to the NPM Registry and Maven Central. While this may be an indication of better processes and awareness by the respective developers, it may also be a result of the size and popularity of the repository. Out of the three repositories observed in this work, NuGet.org has both the least amount of artifacts and the fewest downloads per week, which may lead to fewer vulnerabilities being introduced, and fewer vulnerabilities being discovered. While we normalized our measures to include the size of repositories where applicable, further analysis is necessary to identify the efects of repository size onto the behavior we observed.

8 RELATED WORK

Dependencies between software artifacts for certain software systems have been the topic of a variety of publications. Furthermore, many researchers have analyzed the impact and lifecycle of software vulnerabilities, as well as possible mitigation strategies. In this section, we present the most prominent examples of related work.

In [4], Benelallam et al. present a snapshot of the Maven Central dependency graph as of September 6, 2018. The authors implemented a tool called Maven Miner, which processes artifacts from the Maven Central repository, calculates their dependencies and creates a graph representation that is stored in a Neo4J graph database. For our work, we initially used the Maven Miner to gather an up-to-date version of the dependency graph, but failed to do so in time (cf. Section 4.3).

In 2018, Decan et al. performed an empirical study on the impact of software vulnerabilities in the NPM Registry [10]. Based on 700 vulnerabilities, which the authors manually gathered from Snyk.io, they computed a set of afected package releases using the online service at libraries.io. Among other things, the authors observed an ever-growing number of vulnerabilities and afected package releases over time, with more pronounced rates of growth for medium and high severity vulnerabilities. A similar observation is made for commercial

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| Analyzing Vulnerability Impact | • | 1:23 |

applications by Mike Pittenger in [35]. Decan et al. also analyzed the time it takes package maintainers to discover vulnerabilities, and report that the majority was found in packages older than 28 months. Similar to this work, the authors found that most vulnerabilities are patched before their publication, but there "is still a non-negligible proportion of vulnerabilities that take a long time to be ixed." [10].

Bavota et al. performed an in-depth study on 147 Java projects of the Apache ecosystem in 2015 [3], in which they analyze how and why the dependencies between projects are upgraded, left unchanged, or dropped. The authors evaluate the inluence of the client project size, the number of LOC changed in the library project, the number of bugs ixed, and other factors on the upgrade behavior. They ind that substantial changes to the library project, especially bug ixes, are adopted earliest, which matched the authors’ observations in developer communications (eg. via mailing lists). Similarly, a 2021 empirical study conducted by Chinthanet et al. analyzes the adoption of vulnerability-related ixes in NPM projects on GitHub [7]. By inspection of 231 package-side ixing releases, the authors ind that such releases consist of up to 86% code unrelated to the actual ix, and that quick availability of ixes does not ensure a fast adoption process by clients. Nevertheless, a 2019 study by Gkortzis et al. inds evidence for an inverse correlation between code reuse and the number of vulnerabilities, which seems to indicate that ł[...] a high reuse ratio is associated with a lower vulnerability densitył [16].

A 2017 study by Kula et al. analyzes the dependency upgrade behavior of software developers in detail [20]. In an empirical study, the authors collected 4,600 Java projects that are using Maven dependencies from GitHub. They found that while developers heavily rely on third-party libraries, they do not often upgrade their dependencies and tend to stick to popular old releases. An additional case study revealed that Awareness Mechanisms like Release Announcements and Security Advisories have a mixed inluence on developers. The likelihood of upgrading a library dependency is observed to be decreasing with increasing migration efort. A subsequent interviewing process revealed that almost 70% of all developers were unaware of a vulnerable dependency in their project. The other 30% named mostly project-speciic reasons for which a dependency upgrade was not performed or not prioritized, eg. because the developers found that it does not actually apply to the project. This intuition is conirmed in a 2018 study conducted on 60 NPM projects by Zapata et al., which inds that as much as 73.3% of such vulnerable dependencies may not actually apply to a project, concluding that package-level vulnerability analysis leads to signiicant overapproximation [13].

In 2018, Jukka Ruohonen published an analysis on vulnerabilities in Python packages that target web devel-opment [39]. The base data is collected from the Python Package Index (PyPI) and the Safety DB16, a curated list of vulnerabilities in Python packages. The author argues that, while often used in scientiic work, the CVE list of vulnerabilities does often not include smaller libraries or lesser-known exploits. A subsequent analysis of packages related to web development showed that most vulnerabilities in the data set are of mild severity, with Input Validation and XSS being the most common attack vectors. A inal conclusion states that a more meaningful analysis would have to take the dependencies between artifacts into account.

Ponta et al. [36] present a novel approach to detect whether or not a software artifact is afected by a vulnerability. Instead of consuming metadata iles (eg. pom.xml), their tool Vulas combines static and dynamic analyses to decide whether or not the actual vulnerable code construct (eg. a method) is reachable from the analyzed artifact. This way, the number of false positives is reduced and the intuition of vulnerabilities not applying in certain contexts, as reported by Kula et al. in [20], is formalized. In their latest publication [37], Ponta et al. report on the current state of Vulas, and perform a comparative study on 300 enterprise projects. Compared to OWASP Dependency Check (OWASP DC), their tool identiied about 1800 additional true positive indings, whereas almost 89% of 17000 vulnerabilities only reported by OWASP DC turned out to be false positives.

In their 2016 publication, Alqahtani et al. use semantic web technologies to establish links between sources on software vulnerabilities and source code repositories [2]. They argue that traditionally, information sources on

16github.com/pyupio/safety-db

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| 1:24 | • | Johannes DÃĳsing and Ben Hermann |

vulnerabilities and project metadata are heterogeneous, which hinders the estimation of vulnerability impact and artifact security.

Shahzad et al. published their study on the life cycles of software vulnerabilities in 2012 [40]. They obtained a sample of 46,310 vulnerabilities from diferent sources, including the NVD, and present an analysis regarding multiple data dimensions. Based on that, they extract association rules regarding the patching and exploitation behavior, which are then used for detecting patterns in the data set.

In their 2018 publication, Pashchenko et al. argue that existing tools for vulnerable dependency detection often produce false positives, which in turn lead to ineicient development processes, especially in commercial software development [32]. The authors state that dependency scopes are often not evaluated in existing approaches, as well as halted dependencies. The paper presents an analysis approach that is able to detect vulnerable dependencies for Maven artifacts while improving on the deiciencies mentioned above. In particular, the authors use heuristics based on the average release time of a library in order to identify halted dependencies. Instead of automated vulnerability aggregation, the authors rely on manual identiication of vulnerable code for each vulnerability. Finally, Pashchenko et al. conduct a study of their approach on 200 open source Maven libraries. They ind that 14% of all dependencies in their sample are halted, with 1% of them being afected by known vulnerabilities. They conclude that employing their approach helped to reduce the time developers have to spend on false positives in vulnerable dependency detection.

Building on their previous work, in 2020 Pashchenko et al. present Vuln4Real, a methodology for overcoming the inaccuracies of traditional approaches for vulnerable dependency detection [33]. They implement the methodology for the Maven build system and perform an empirical study on 500 libraries. The authors ind that 80% of all vulnerable dependencies can potentially be ixed by a direct dependency upgrade. Furthermore, Vuln4Real decreases the number of false positives by 27%, thus efectively saving developers the time they would otherwise have to spend on analyzing an inaccurate alert.

In 2020, Pashchenko et al. published a qualitative study on the security implications of developer decisions re-garding dependency management [34]. The authors interviewed a total of 25 developers from diferent enterprises to investigate the trade-of between functional and security-related concerns. Their results imply that selecting an appropriate dependency is a complex task, and security concerns are often neglected in these scenarios. Furthermore, the authors observe that patches are more likely to be adopted by developers if they are not bundled with functional changes.

9 CONCLUSION

Our work presented three core contributions, that mitigate the resulting threats to application security and foster a general understanding of how vulnerability patches and dependency upgrades are performed in Maven Central, NuGet.org, and the NPM Registry.

At irst, we designed and implemented a distributed application that aggregates the dependency graphs for artifacts from all three repositories. An additional post-processing step leverages a Snyk data dump to annotate information about CVE software vulnerabilities to the graph. By executing the application we obtained a vulnerability-enriched dependency graph for Maven Central (as of September 2018), NuGet.org, and the NPM Registry (both up-to-date as of June 2020). It contains a total of 19 million software artifact nodes, which belong to 1.8 million libraries and have 4.9 million unique dependency speciications, with 5378 vulnerabilities being annotated.

Our second contribution is a detailed analysis of the data that we previously generated. It was conducted based on four diferent research questions and comprised the creation of applications for data processing as well as interactive data visualizations for manual data exploration.

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| Analyzing Vulnerability Impact | • | 1:25 |

We found that a substantial amount of vulnerabilities is still unpatched (over 25%), but existing patches are often released long before vulnerability publication. We also observed that vulnerabilities may transitively afect an artifact with a transitive depth of up to 25, making it hard to manually discover a potential security risk. In addition to that, our analysis revealed that up to 20% of all libraries (Maven Central) in a repository have been afected by direct vulnerable dependencies, and 60% of those libraries did not upgrade any of them. Interestingly, while most dependency upgrades happen long before a vulnerability is published, the act of publishing does seem to trigger an immediate dependency upgrade for some maintainers in Maven Central and the NPM Registry, thus indicating that they may be actively responding to security advisories.

Our third and inal contribution is a set of observed security threats and mitigation strategies, which we derived from the analysis results described above. These strategies are meant to mitigate the security threats that result from our indings, thus potentially increasing application security.

With each of our contributions, we aim to improve the current state-of-the-art regarding the development of patches and the upgrades of vulnerable dependencies, thus ultimately reducing the number of exploitable security laws in software applications. Our indings provide the foundation for a common understanding of the topic and indicate how to tackle some of the problems that we pointed out. Our results may be extended by incorporating additional repositories or data sources.

ACKNOWLEDGMENTS

Special thanks to Snyk Ltd. for sharing their data on vulnerabilities for the context of this research project.

REFERENCES

[1] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: an empirical case study on npm and PyPI. Empirical Software Engineering 25, 2 (01 Mar 2020), 1168ś1204. <https://doi.org/10.1007/s10664-019-09792-9>[2] Sultan S. Alqahtani, Ellis E. Eghan, and Juergen Rilling. 2016. Tracing known security vulnerabilities in software repositories âĂŞ A Semantic Web enabled modeling approach. Science of Computer Programming 121 (2016), 153ś175. [https://doi.org/10.1016/j.scico.2016.](https://doi.org/10.1016/j.scico.2016.01.005)  [01.005](https://doi.org/10.1016/j.scico.2016.01.005) Special Issue on Knowledge-based Software Engineering.

[3] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: an evolutionary study. Empirical Software Engineering 20, 5 (01 Oct 2015), 1275ś1317. [https://doi.org/10.1007/](https://doi.org/10.1007/s10664-014-9325-9)  [s10664-014-9325-9](https://doi.org/10.1007/s10664-014-9325-9)   
[4] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. 2019. The Maven Dependency Graph: A Temporal Graph-Based Representation of Maven Central. In Proceedings of the 16th International Conference on Mining Software Repositories (Montreal, Quebec, Canada) (MSR ’19). IEEE Press, 344âĂŞ348. <https://doi.org/10.1109/MSR.2019.00060>  
[5] B. Boehm. 1999. Managing software productivity and reuse. Computer 32, 9 (1999), 111ś113. <https://doi.org/10.1109/2.789755>[6] Paolo Boldi and Georgios Gousios. 2021. Fine-Grained Network Analysis for Modern Software Ecosystems. ACM Transactions on Internet Technology 21, 1 (Feb 2021), 1âĂŞ14. <https://doi.org/10.1145/3418209>  
[7] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability ixes. Empirical Software Engineering 26, 3 (Mar 2021). [https://doi.org/10.1007/s10664-](https://doi.org/10.1007/s10664-021-09951-x) [021-09951-x](https://doi.org/10.1007/s10664-021-09951-x)   
[8] MITRE Corporation. 2019. CVE and NVD Relationship. [https://cve.mitre.org/about/cve\_and\_nvd\_relationship.html.](https://cve.mitre.org/about/cve_and_nvd_relationship.html)  Accessed: 2020-06-04.

[9] MITRE Corporation. 2021. CVE List Home. [https://cve.mitre.org/cve/.](https://cve.mitre.org/cve/) Accessed: 2021-05-17.

[10] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the npm Package Dependency Network. In Proceedings of the 15th International Conference on Mining Software Repositories (Gothenburg, Sweden) (MSR âĂŹ18).

Association for Computing Machinery, New York, NY, USA, 181âĂŞ191. <https://doi.org/10.1145/3196398.3196401>  
[11] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering 24 (Feb 2019), 381ś416. <https://doi.org/10.1007/s10664-017-9589-y>[12] Johannes DÃĳsing and Ben Hermann. 2021. sse-labs/dependency-graph-miner: Full Release for Final Paper. (Jun 2021). [https:](https://doi.org/10.5281/zenodo.5040439)  [//doi.org/10.5281/zenodo.5040439](https://doi.org/10.5281/zenodo.5040439)   
[13] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. 2018. Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages. In 2018

Digit. Threat. Res. Pract.



|  |  |  |
| --- | --- | --- |
| 1:26 | • | Johannes DÃĳsing and Ben Hermann |

IEEE International Conference on Software Maintenance and Evolution (ICSME). 559ś563. <https://doi.org/10.1109/ICSME.2018.00067>[14] Apache Software Foundation. 2020. Maven - Introduction. [https://maven.apache.org/what-is-maven.html.](https://maven.apache.org/what-is-maven.html) Accessed: 2020-11-19. [15] Stefan Frei, Dominik Schatzmann, Bernhard Plattner, and Brian Trammell. 2010. Modeling the Security Ecosystem - The Dynamics of (In)Security. 79ś106. <https://doi.org/10.1007/978-1-4419-6967-5_6>  
[16] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2019. A Double-Edged Sword? Software Reuse and Potential Security Vulnerabilities. In Reuse in the Big Data Era, Xin Peng, Apostolos Ampatzoglou, and Tanmay Bhowmik (Eds.). Springer International Publishing, Cham, 187ś203.

[17] Hackerone. 2020. Hack the Pentagon. [https://www.hackerone.com/hack-the-pentagon.](https://www.hackerone.com/hack-the-pentagon) Accessed: 2020-08-20.

[18] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. 2011. On the Extent and Nature of Software Reuse in Open Source Java Projects. Top Productivity through Software Reuse: 12th International Conference on Software Reuse, ICSR, Pohang, South Korea 6727, 207ś222. <https://doi.org/10.1007/978-3-642-21347-2_16>  
[19] NPM Inc. 2020. About NPM. [https://docs.npmjs.com/about-npm.](https://docs.npmjs.com/about-npm) Accessed: 2020-11-19.

[20] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? Empirical Software Engineering 23, 1 (01 Feb 2018), 384ś417. <https://doi.org/10.1007/s10664-017-9521-5>  
[21] Gabriel Lawrence and Chris Frohof. [n.d.]. Marshalling Pickles - How Deserializing Objects Can Ruin Your Day. [https://www.slideshare.](https://www.slideshare.net/frohoff1/appseccali-2015-marshalling-pickles)

[net/frohof1/appseccali-2015-marshalling-pickles](https://www.slideshare.net/frohoff1/appseccali-2015-marshalling-pickles)   
[22] W. C. Lim. 1994. Efects of reuse on quality, productivity, and economics. IEEE Software 11, 5 (1994), 23ś30. [https://doi.org/10.1109/52.](https://doi.org/10.1109/52.311048)

[311048](https://doi.org/10.1109/52.311048)   
[23] Snyk Limited. 2020. Snyk - Disclosing vulnerabilities. [https://support.snyk.io/hc/en-us/articles/360005933037-Snyk-open-source-](https://support.snyk.io/hc/en-us/articles/360005933037-Snyk-open-source-packages-disclosure-policy) [packages-disclosure-policy.](https://support.snyk.io/hc/en-us/articles/360005933037-Snyk-open-source-packages-disclosure-policy) Accessed: 2021-04-14.

[24] Snyk Limited. 2021. Snyk Intel Vulnerability Database. [https://snyk.io/product/vulnerability-database/.](https://snyk.io/product/vulnerability-database/) Accessed: 2021-04-14.

[25] Microsoft. 2019. An introduction to NuGet. [https://docs.microsoft.com/en-us/nuget/what-is-nuget.](https://docs.microsoft.com/en-us/nuget/what-is-nuget) Accessed: 2020-11-19. [26] Microsoft. 2020. Create .NET apps faster with NuGet. [https://www.nuget.org/.](https://www.nuget.org/) Accessed: 2020-11-20.

[27] Microsoft. 2020. Microsoft Online Services Bounty Program. [https://www.microsoft.com/en-us/msrc/bounty-microsoft-cloud?rtc=1.](https://www.microsoft.com/en-us/msrc/bounty-microsoft-cloud?rtc=1)  Accessed: 2020-08-20.

[28] MvnRepository. 2021. Central Repository. [https://mvnrepository.com/repos/central.](https://mvnrepository.com/repos/central) Accessed: 2021-05-17.

[29] MvnRepository. 2021. Jackson Databind. mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind. Accessed: 2021-05- 17.

[30] Oracle. 2015. Understanding Maven Version Numbers. [https://docs.oracle.com/middleware/1212/core/MAVEN/maven\_version.htm#](https://docs.oracle.com/middleware/1212/core/MAVEN/maven_version.htm#MAVEN402)  [MAVEN402.](https://docs.oracle.com/middleware/1212/core/MAVEN/maven_version.htm#MAVEN402) Accessed: 2021-05-14.

[31] OW2. 2021. FASTEN Project. [https://www.fasten-project.eu/view/Main/.](https://www.fasten-project.eu/view/Main/) Accessed: 2021-04-30.

[32] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable Open Source Dependencies: Counting Those That Matter. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (Oulu, Finland) (ESEM 18). Association for Computing Machinery, New York, NY, USA, Article 42, 10 pages. [https:](https://doi.org/10.1145/3239235.3268920)  [//doi.org/10.1145/3239235.3268920](https://doi.org/10.1145/3239235.3268920)   
[33] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci. 2020. Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies. IEEE Transactions on Software Engineering (2020), 1ś1. <https://doi.org/10.1109/TSE.2020.3025443>  
[34] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A Qualitative Study of Dependency Management and Its Security Implications.

In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (Virtual Event, USA) (CCS ’20). Association for Computing Machinery, New York, NY, USA, 1513âĂŞ1531. <https://doi.org/10.1145/3372297.3417232>  
[35] Mike Pittenger. 2016. The State of Open Source Security in Commercial Applications. Black Duck Open Source Security Analysis (2016). [36] S. E. Ponta, H. Plate, and A. Sabetta. 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). 449ś460. [https:](https://doi.org/10.1109/ICSME.2018.00054)  [//doi.org/10.1109/ICSME.2018.00054](https://doi.org/10.1109/ICSME.2018.00054)   
[37] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. Empirical Software Engineering 25, 5 (01 Sep 2020), 3175ś3215. <https://doi.org/10.1007/s10664-020-09830-x>[38] Open Web Application Security Project. 2020. Dependency-Check Maven. [https://jeremylong.github.io/DependencyCheck/dependency-](https://jeremylong.github.io/DependencyCheck/dependency-check-maven/index.html) [check-maven/index.html.](https://jeremylong.github.io/DependencyCheck/dependency-check-maven/index.html) Accessed: 2020-06-05.

[39] J. Ruohonen. 2018. An Empirical Analysis of Vulnerabilities in Python Packages for Web Applications. In 2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP). 25ś30. <https://doi.org/10.1109/IWESEP.2018.00013>  
[40] M. Shahzad, M. Z. Shaiq, and A. X. Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. In 2012 34th International Conference on Software Engineering (ICSE). 771ś781. <https://doi.org/10.1109/ICSE.2012.6227141>  
[41] CÃľsar Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. Empirical Software Engineering 26, 3 (Mar 2021). <https://doi.org/10.1007/s10664-020-09914-8>

Digit. Threat. Res. Pract.

|  |  |  |
| --- | --- | --- |
| Analyzing Vulnerability Impact | • | 1:27 |

[42] Synopsys, Inc. 2021. Open Source Security and Risk Analysis Report. [https://www.synopsys.com/software-integrity/resources/analyst-](https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html) [reports/open-source-security-risk-analysis.html.](https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html) Accessed: 2021-05-17.

[43] NPM Security Team. 2016. Security Holding Package. [https://www.github.com/npm/security-holder#readme.](https://www.github.com/npm/security-holder#readme) Accessed: 2020-09-07.



Digit. Threat. Res. Pract.