Empirical Analysis of Security Vulnerabilities in Python Packages

Mahmoud Alfadel, Diego Elias Costa, Emad Shihab
Data-driven Analysis of Software (DAS) Lab
Concordia University
Montreal, Canada
{mahmoud.alfadel, diego.costa, emad.shihab}@concordia.ca

***Abstract*—Software ecosystems play an important role in modern software development, providing an open platform of reusable packages that speed up and facilitate development tasks. However, this level of code reusability supported by software ecosystems also makes the discovery of security vulnerabilities much more difficult, as software systems depend on an increasingly high number of packages. Recently, security vulnerabilities in the *npm* ecosystem, the ecosystem of Node.js packages, have been studied in the literature. As different software ecosystems embodied different programming languages and particularities, we argue that it is also important to study other popular programming languages to build stronger empirical evidence about vulnerabilities in software ecosystems.**

In this paper, we present an empirical study of 550 vulnerability reports affecting 252 Python packages in the Python ecosystem (*PyPi*). In particular, we study the propagation and life span of security vulnerabilities, accounting for how long they take to be discovered and fixed. Our findings show that the discovered vulnerabilities in Python packages are increasing over time, and they take more than 3 years to be discovered. The majority of these vulnerabilities (50.55%) are only fixed after being publicly announced, giving ample time for attackers exploitation. We find similarities in some characteristics of vulnerabilities in *PyPi* and *npm* and divergences that can be attributed to specific *PyPi* policies. By leveraging our findings, we provide a series of implications that can help the security of software ecosystems by improving the process of discovering, fixing and managing package vulnerabilities.

***Index Terms*—python, pypi, packages, vulnerabilities, empirical studies.**

I. Introduction

Modern software systems increasingly depend on external reusable code. This reusable code takes the form of packages (e.g., libraries) and is available from online repositories and often delivered by package management systems, such as *npm* for Node.js packages and *PyPi* for Python packages. The collection of packages that are reused by a community, together with their users and contributors is denoted as a *software ecosystem*. While software ecosystems have many benefits, providing an open platform with a large number of reusable packages that speed up and facilitate development tasks, such openness and large scale leads to the spread of vulnerabilities through package network, making the vulnerability discovery much more difficult, given the heavy dependence on such packages and their potential security problems [1].

Many software applications depend on vulnerable packages [2]. The two most critical aspects in dealing with package vulnerabilities are how fast developers can discover and fix the vulnerability, and how fast the applications update their packages to accommodate the fixed versions. The delay between discovering a package vulnerability and releasing its fix may expose the applications to threats and increase the likelihood of an exploit being developed. Heartbleed, a security vulnerability in OpenSSL package, is perhaps the most infamous example. The vulnerability was introduced in 2012 and remained uncovered until April 2014. After its disclosure, researchers found more than 692 different sources of attacks attempting to exploit the vulnerability in applications that used the OpenSSL package [3].

Hence, studying how vulnerabilities propagate, get discovered and fixed is essential for the health of ecosystems. Recent studies [4]–[6] analysed the impact of vulnerabilities in the *npm* ecosystem. Decan et al. [5] found that it takes 24 months to discover 50% of *npm* package vulnerabilities, whilst 82% of the discovered vulnerabilities are fixed before being publicly announced, where they are less likely to be exploited.

While *npm* is one of the largest software ecosystems to date [7], the investigation of *npm* vulnerabilities provides an important but restricted view of the software development ecosystems. How much of the findings are particular to *npm*'s development culture and how much of it can be generalized to other ecosystems? We argue that it is important to study other software ecosystems to contrast with *npm* and draw more generalizable empirical evidence about vulnerabilities in software ecosystems. Our argument is supported by previous studies (e.g., [8]–[11]) that show differences across ecosystems. For instance, Decan et al. [11] found that *PyPi* ecosystem has a less complex and intertwined network than ecosystems such as *npm* and CRAN. This is partially due to Python's robust standard library, which discourage developers of using too many external packages in contrast to JavaScript and R ecosystems.

This motivated us to take a new look and provide a wider picture by studying security vulnerabilities in the *PyPi* ecosystem. Furthermore, Python is a major programming language in the current development landscape, used by 44.1% of professional developers according to the 2020 Stack-Overflow survey [12]. We conduct an exploratory research to study *security vulnerabilities prevalence and their respective discovery and fix timeline in the Python ecosystem*. Inspired by the study on the *npm* ecosystem [5], we aim to answer the following research questions (RQs):

- **RQ$_1$**: How are vulnerabilities distributed in the *PyPi* ecosystem?
- **RQ$_2$**: How long does it take to discover a vulnerability in the *PyPi* ecosystem?
- **RQ$_3$**: When are vulnerabilities fixed in the *PyPi* ecosystem?
- **RQ$_4$**: How long does it take to fix a vulnerability in the *PyPi* ecosystem?

Also, we compare our study, where applicable, to the *npm* ecosystem [5].

To answer our research questions, we analyzed 550 vulnerability reports that affect 252 Python packages of which 7,536 package versions are affected. We observed several interesting findings. In some aspects, our study yields similar findings to the ones observed in the *npm* study [5]. For example, vulnerabilities in both ecosystems take a significantly long time to be discovered, approximately 2 years in the *npm* and 3 years in the *PyPi* ecosystem.

However, in other aspects, our results show a drastic departure from *npm*'s reported findings. For example, unlike *npm*, the majority of *PyPi* vulnerabilities (50.55%) were only fixed after being publicly announced, which may increase the chances of having the vulnerability exploited by attackers. Our further investigation attributes such observation to the particularities of the *PyPi* ecosystem's protocol of disclosing and publishing vulnerabilities.

Based on our empirical findings, we offer several important implications to researchers and practitioners to help them provide a more secure environment for software ecosystems. To summarize, this paper makes the following contributions:

- We perform the first empirical study to analyse security vulnerabilities in the Python ecosystem. Our study covers 12 years of *PyPi* reported vulnerabilities, affecting 252 Python packages.
- We compare the findings of our study to a previous study conducted on the *npm* ecosystem. We also provide implications that aim at a more secure development environment for software ecosystems.
- We make our dataset of this study publicly available to facilitate reproducibility and future research [13].

Paper organization.
Section II describes the terminology and the process of collecting and curating our dataset. In Section III, we dive into our study by motivating and describing the methods used to investigate each research question, as well as presenting the findings obtained in our study. We discuss the results and implications of our study in Section IV. We state the threats to validity and limitations to our study in Section V. Related work is presented in Section VI. Finally, Section VII concludes our paper.

II. Methodology

In this section, we present an overview of software vulnerabilities and the terminology adopted throughout this study. We also explain how we collect and prepare the data used to investigate our research questions.

A. Terminology

A software vulnerability is a weakness that allows unauthorized actions and/or malformed access to a given software project [14]. The lifetime of a vulnerability typically goes through various stages, according to when a vulnerability was first introduced, discovered and publicly announced. To ground our study, we use the various stages and define dates specific to a package vulnerability:

- **package vulnerability introduction date** indicates when the vulnerability was first introduced in the affected package, i.e., the release date of the first affected version by the package vulnerability.
- **package vulnerability discovery date** indicates the date in which the package vulnerability was discovered and reported to the maintainer of the package.
- **package vulnerability publication date** marks the date when the vulnerability information was publicly announced.
- **package vulnerability fix date** indicates the release date of the first fixed version of the package vulnerability.

Next, we explain how we collect and process the data used to answer our RQs.

B. Data Collection and Processing

Data Collection. To conduct our study, we collect two datasets: (1) the Python (*PyPi*) packages and (2) the security vulnerabilities that affect those *PyPi* packages. We obtain the information of *PyPi* packages from Libraries.io [15], and the security vulnerabilities from the Snyk.io dataset [16].

To collect the *PyPi* packages, we use the service Libraries.io since it provides the *PyPi* packages along with their respective metadata. The metadata provides detailed information about each package such as, the existing versions and the creation timestamps of those versions. Such data is needed to map the affected versions given by our vulnerabilities dataset. Also, we need the versions timestamps to perform time-based analyses, such as the time it takes to discover and fix a vulnerability with respect to the first affected package version.

To collect the vulnerabilities for the *PyPi* packages, we resort to the dataset provided by Snyk.io [17]. Snyk.io is a platform that monitors security reports to provide a dataset for different package ecosystems, including *PyPi*, and publishes a series of information about vulnerabilities. We show in Table I an example of a security report extracted from Snyk.io dataset for the package *pillow*. For each affected package, the dataset specifies the type of vulnerability, the vulnerability constraint (this helps us to specify the affected versions) and the fixed versions (remediation range). Moreover, the report contains the dates when the vulnerability was discovered and the date when it was published on Snyk.io dataset. Severity level

TABLE I: Example of a security report extracted from Snyk.io for the `pillow` package.

Information	Example
Vulnerability type	Buffer Overflow
Affected package name	pillow
Platform type	*PyPi*
Vulnerable constraint (affected versions)	$< 6.2.2$
Vulnerability Discovery date	03 Jan, 2020
Vulnerability Published date	10 Jan, 2020
Severity level	High
Remediation	$\geq 6.2.2$

TABLE II: Descriptive statistics of the *PyPi* dataset.

Source	Stats	#
Libraries.io	*PyPi* packages	116,527
	Versions of *PyPi* packages	893,978
Snyk.io	Security reports on *PyPi*	550
	Corresponding vulnerable packages	252
	Versions of vulnerable packages	12,210
	Affected versions by vulnerability	7,536

has three possible values, high, medium, and low, which are assigned manually by the Snyk.io team based on the Common Vulnerability Scoring System (CVSS) [18].

Data Processing. As a pre-processing step, we need to determine all the vulnerable packages and their associated versions. First, we obtain the list of all versions of all vulnerable packages from the Libraries.io dataset. Then, we determine the affected versions of the vulnerable packages by cross-referencing the vulnerability constraint of the Snyk.io report (e.g., $< 6.2.2$) and resolving the versions by using the SemVer tool [19]. In the particular example of Table I, we resolve the constraint $< 6.2.2$ to a list of 68 versions of the `pillow` package affected by the Buffer Overflow vulnerability.

We want to analyze the time needed to discover a package vulnerability, hence, we need to identify the version that was *first affected* by a vulnerability. To that aim, once we identify the list of affected versions, we consider the first affected version as the oldest version of the vulnerable package. In the example of Table I, the first affected version was the package version 1.0.0.

We also aim to investigate the time it takes to fix a package vulnerability once the vulnerability is discovered. This requires that we identify the *first fixed version* of the package vulnerability. Similar to the identification of the first affected version, once we resolve the remediation range by using the SemVer tool, we collect a list of versions in which the vulnerability is considered fixed. We then assign the first fixed version as the oldest package version present in the list of fixed versions. In the example of Table I, the first fixed version is the package 6.2.2.

Our initial dataset contains 622 vulnerability reports on the *PyPi* packages. From this original set, 62 vulnerabilities do not match any packages in the Libraries.io database and were removed from our analysis. Following the filtration process applied by Decan *et al.* [5], we also removed 10 vulnerabilities of type “Malicious Package”, because they do not really introduce vulnerable code. These vulnerabilities are packages with names close to popular packages (a.k.a. typo-squatting) in an attempt to deceive users at installing their harmful packages. At the end of this filtering process, our dataset contains 550 vulnerability reports. Such reports affect 252 Python packages in *PyPi*. Note that these 252 Python packages have released a total of 12,210 versions, in which, according to the vulnerable constraint of reports, 7,536 versions (61.7%) contain at least one reported vulnerability. Table II shows the descriptive statistics of our dataset.

As part of our study goal is to compare our results to the *npm* study, we verify how our dataset compares with the one used by Decan *et al.* [5]. The *npm* dataset contains 399 vulnerabilities which affect 269 *npm* packages with a similar number of versions (14,931) and similar number of affected versions (6,752). Both datasets are comparable in terms of the number of vulnerability reports and the number of affected packages and versions. Finally, note that we collect our dataset in the similar timeline as the *npm* study in order to make our study comparable and to perform a relatively fair comparison between our findings and the ones reported from *npm* [5], i.e., we collect all vulnerability reports that were published before Jan. 2018.

III. Results

In this section, we present the findings of our empirical study. For each RQ, we present a motivation, describe the approach used to tackle the research question and discuss the results of our analysis.

A. ***RQ***$_1$*: How are vulnerabilities distributed in the PyPi ecosystem?*

Motivation. Prior work reported a steady growth of packages in software ecosystems [11], [20]. This growth may have serious repercussions for package vulnerabilities, facilitating their spread to high number of packages and applications, and magnifying their potential for exploitation. Therefore, in this RQ we investigate how software package vulnerabilities are distributed in the *PyPi* ecosystem. We examine the distribution from three perspectives: a) the trend of discovered vulnerabilities over time; b) how many versions of packages are affected by vulnerabilities; and c) what are the most commonly identified types of vulnerabilities in *PyPi*.

Approach. To shed light on the distribution of software vulnerabilities in the *PyPi* software ecosystem, we leverage the following approaches:

In the first analysis, we focus on investigating the trend of discovered vulnerabilities over time in the *PyPi* ecosystem. In essence, we want to investigate how the number of discovered vulnerabilities change and how many packages are affected as the ecosystem grows? To do that, we group the discovered vulnerabilities by the time they were reported, and present

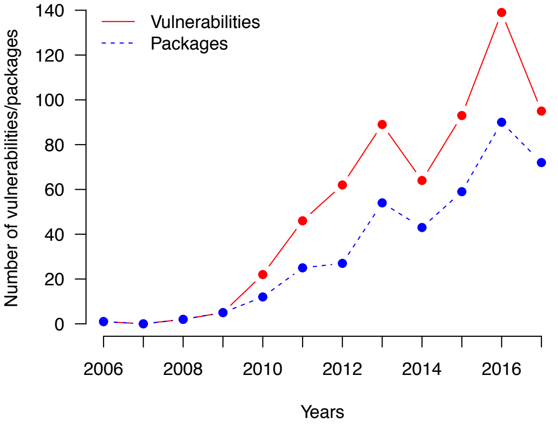


Fig. 1: Introduction of vulnerabilities and packages being affected per year.

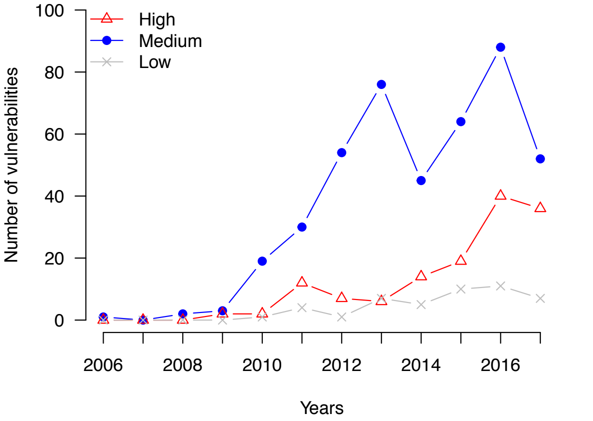


Fig. 2: Introduction of vulnerabilities per year by the severity levels: high, medium, and low.

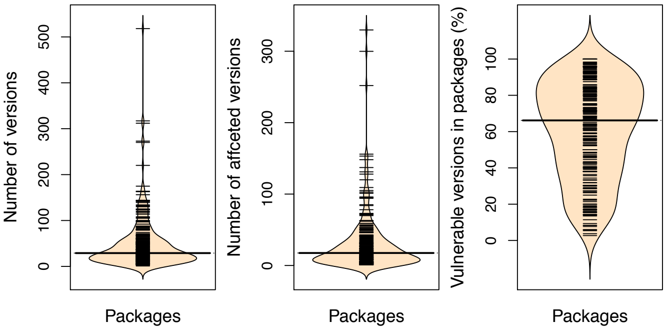
the evolution of the number of vulnerabilities and packages affected per year. We also break the analysis per severity level, provided by Snyk.io, to help us quantify the level of threat of newly discovered vulnerabilities in the ecosystem.

In the second analysis, we investigate the vulnerabilities distribution over package versions. A single vulnerability can impact many versions of a package, making it harder for dependents to select a version unaffected by this vulnerability. To that aim, we utilize the vulnerability constraint provided by the Snyk.io dataset (mentioned in Table I, Section II-B) to identify the list of affected versions by a vulnerability.

The third analysis has the goal of reporting the most commonly identified vulnerability types in the *PyPi* ecosystem. The Snyk.io dataset associates each vulnerability report with a Common Weakness Enumeration (CWE) [21], aiming at categorizing vulnerabilities based on the explored software weaknesses (e.g. Buffer Overflow). Currently, CWE contains a community-developed list of 700 common software weaknesses. We examine the frequency of vulnerability types to establish a profile of the vulnerabilities in the *PyPi* ecosystem. In adittion, we also break our analysis by severity level to investigate how the threat levels are distributed in each vulnerability type.

Findings. Figure 1 shows the number of discovered vulnerabilities as well as the number of packages being affected over the years. **We observe a steady increase in the number of vulnerable packages, accompanying the *PyPi* ecosystem growth**. In 2012, in the middle of this ecosystem lifetime, 27 packages were discovered to be vulnerable, in 2016 this number increased three-fold, i.e., 90 vulnerable packages were newly discovered.

Figure 2 presents the introduction of vulnerabilities over time by the severity level, showing that **the majority of newly discovered vulnerabilities are of medium and high severity.** Overall, the vulnerabilities classified with medium severity make the bulk of 71.64% of all vulnerabilities, followed by high severity vulnerabilities representing 20.73% of our dataset. These findings are worrisome to the *PyPi* community,



(a) Number of versions.

(b) Number of affected versions.

(c) Percentage of affected versions.

Fig. 3: Distribution of versions and affected versions of the 252 vulnerable packages of our dataset. In median, packages have 29 versions and 18 affected versions once a vulnerability is discovered.

as such critical vulnerabilities have a higher chance of being exploited, i.e., allow an attacker to execute malicious code and damage the software.

Figure 3 shows bean plots of three distributions: the number of versions of the 252 vulnerable *PyPi* packages in our dataset (Figure 3a), the number of affected versions in such vulnerable packages (Figure 3b), and the percentages of vulnerable versions in the packages (Figure 3c). We observe that most packages have dozens of versions (median number of versions is 29), and tend to have, on median, 18 vulnerable versions. The affected versions represent an alarmingly high proportion of all versions in a package, considering the package versions available at discovery time of the vulnerability. Figure 3c shows that **half of the packages have at least 68% of their versions affected by a vulnerability, when a vulnerability is first discovered**. In 15% of the packages, the share of vulnerable versions can represent 90% of all released versions at the time the vulnerability was discovered. The result indicates that vulnerabilities are not limited to a few versions of a package, making it difficult for dependents to rollback to an unaffected

TABLE III: Ranking of the 5 most commonly found vulnerability types (CWE) in *PyPi*.

Rank	Vulnerability type (CWE)	Freq.	Frequency by severity		
			High	Medium	Low
1	Cross-Site-Scripting (XSS)	130	4	118	8
2	Denial of Service (DoS)	72	11	59	2
3	Arbitrary Code Execution	66	39	26	1
4	Information Exposure	60	8	44	8
5	Access Restriction Bypass	34	10	23	1

version if a fix is not available at the time of the vulnerability discovery.

Since vulnerabilities can have different types (e.g., Buffer Overflow and SQL injection), we examine the different vulnerability types given by the Common Weakness Enumeration (CWE) that *PyPi* packages have. While we found that packages in the *PyPi* ecosystem are affected by 90 distinct CWEs, **the majority of the discovered vulnerabilities (65.82%) are concentrated on 5 main types**. Table III shows the distribution of the vulnerabilities over the 5 most commonly found CWEs. As we can see, **XSS is the most common CWE with 130 vulnerabilities**. Also, we observe that most of the XSS vulnerabilities are of medium severity. For the remaining CWEs, the proportion in each type varies from 72 vulnerabilities of type Denial of Service (DoS) to 34 of type Access Restriction Bypass CWE. Breaking down the proportions of vulnerabilities by severity shows that the majority of vulnerabilities from these types are of medium and high severity, indicating that they represent a serious threat to affected applications. This is particularly severe for the vulnerabilities of Arbitrary Code Execution type, where we found a higher frequency of high severity vulnerabilities than of medium and low severity levels combined.

Comparison to the *npm* ecosystem. The vulnerabilities found in *npm* [5] followed a similar distribution to our findings in the *PyPi* ecosystem. In *npm*, a) the new discovered vulnerabilities are increasing over the time, and the majority of those vulnerabilities are also of medium and high severity; b) such *npm* vulnerabilities are not limited to a few versions, i.e., 75% of vulnerable packages have more than 90% of their versions being affected by a vulnerability at the discovery time; c) XSS was also found to be the most common vulnerability among *npm* vulnerabilities (i.e., 105 occurrences out of 399 vulnerabilities overall).

> The number of vulnerabilities is increasing over time in the *PyPi* ecosystem accompanying the growth of the ecosystem. Newly reported vulnerabilities tend to be of medium and high severity and affect the majority of versions of a software package. The majority of vulnerabilities are concentrated on five vulnerability types, with Cross-Site-Scripting (XSS) being the most common.

*B. **RQ$_2$**: How long does it take to discover a vulnerability in the PyPi ecosystem?*

Motivation. This question aims to investigate how long it takes to discover package vulnerabilities in the *PyPi* ecosystem. Answering this question is relevant since the longer a vulnerability remains undiscovered, the higher the chances it will be exploited by attackers. Also, since security maintainers need to discover vulnerabilities as soon as possible to mitigate the harmful impact, providing them with information regarding the life cycle of a vulnerability discovery is vital. Therefore, in this question, we study how long does it take to discover a vulnerability since it was first introduced in the package's source-code?

Approach. Our goal is to calculate the time required to discover a vulnerability in the *PyPi* ecosystem. To do so, we collect the discovery dates of all the vulnerabilities from the Snyk.io dataset. Then, we obtain the timestamps of the vulnerabilities introduction date from Libraries.io (as described in Section II-B). Note that the vulnerability introduction date is the release date of the first affected version by the package vulnerability. We then calculate the time difference between the vulnerabilities discovery date and the vulnerabilities introduction date.

To gain more insight about the time it takes to discover the vulnerabilities, we conduct a survival analysis method (a.k.a. event history analysis) [22]. The survival analysis is a non-parametric statistic used to measure the survival function from lifetime data where the outcome variable is the "time until the occurrence of an event of interest". For example, it may be used to measure the time duration an employee remains unemployed after a job loss. In the context of our study, we are interested in the time it takes to discover a vulnerability. We use the non-parametric Kaplan-Meier estimator [23] to conduct the survival analysis, as used in previous studies [5], [24].

Findings. Figure 4 presents the survival probability for the vulnerability before it gets discovered. The Left-side plot of Figure 4 reveals that **the probability that a *PyPi* package vulnerability takes 37 months to be discovered is 50%**. In practice, this shows that vulnerabilities are not discovered early in the project development. Also, this long process for discovering vulnerabilities explays why a single vulnerability tends to affect dozens of package versions once it is first discovered (RQ1).

Since vulnerabilities impact packages at different severity levels, we break down the analysis of discovered vulnerabilities by their severity. The right-side plot of Figure 4 presents the survival probability for the event "vulnerability is discovered" by their severity and depicts no significant differences among the severity levels. We confirm this result by using the log-rank statistical method [25] to investigate the statistical significance of the results with a confidence level 95% (p-value = 0.94). ***PyPi* vulnerabilities take a substantial long time to be discovered and reported,**

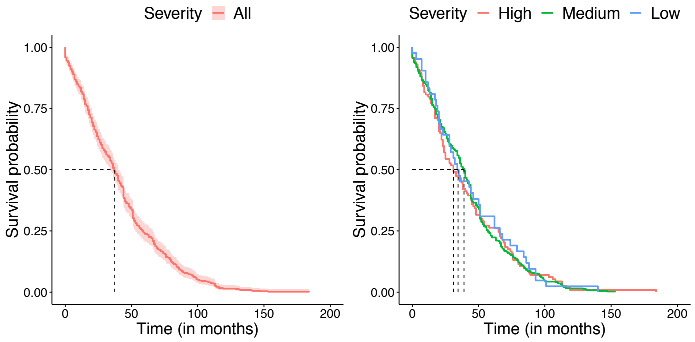


Fig. 4: Kaplan-Meier survival probability for package vulnerabilities to get discovered for all vulnerabilities (left-side plot) and for vulnerabilities broken by severity (right-side plot).

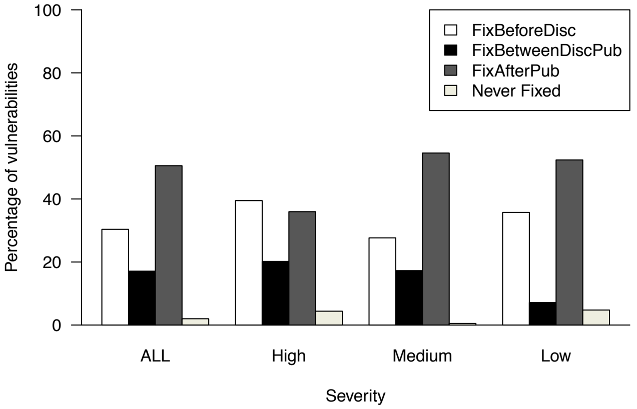


Fig. 5: Percentages of vulnerabilities according to the release time of the first fixed version by severity.

independently of their severity.

Comparison to the *npm* ecosystem. We found a significant difference on the time it takes to discover a vulnerability between the *PyPi* and *npm* packages. Vulnerabilities are discovered with a median of 24 months in the *npm* ecosystem, considerably sooner than the 37 months required for *PyPi* package vulnerabilities. Given the popularity of Javascript programs, *npm* became a prime target for attackers [6], which may have contributed to a faster identification of vulnerabilities. Overall, *npm* and *PyPi* vulnerabilities still take considerably long time to discover vulnerabilities, indicating an issue in the process of testing and detecting vulnerabilities in open source packages.

> Package vulnerabilities in the *PyPi* ecosystem take, on median, more than 3 years to get discovered, regardless of their severity.

*C. **RQ**$_3$: When are vulnerabilities fixed in the PyPi ecosystem?*

Motivation. Vulnerable packages remain affected even after they are discovered [5], [26]. In fact, in many cases, a method of exploitation is reported when the vulnerability is made public, which increases the chances of the vulnerability being exploited by attackers [27]. Therefore, it is of paramount importance that developers release a fix of the package vulnerability quickly. In open-source ecosystems, a quick fix is the only weapon at developers disposal for minimising the risk of exploitation. Hence, in this question we provide package maintainers and users with information about at which stage a vulnerability fix is released in the *PyPi* ecosystem with respect to its discovery and publication date, i.e., we investigate whether a vulnerability fixed version is released before or after the vulnerability becomes publicly announced to better understand the threat level of *PyPi* package vulnerability.

Approach. Our goal is to study when vulnerabilities are fixed. To that aim, we categorise a vulnerability fix based on the stages of a vulnerability lifecycle. In other words, we analyse if the fix version was released before the vulnerability discovery time, in between the discovery time and publication time, or after the vulnerability is made public.

To achieve our goal, we obtain, for each vulnerability, the date of the first fixed version and compare it to the discovery and publication dates. The fix can then be categorized as: “before the vulnerability has been discovered” or short FixBeforeDisc, “between discovery and publication date” or short FixBetweenDiscPub, “after the vulnerability has been made public” or short FixAfterPub, and “Never Fixed”. We then report the frequencies of fixes in each category.

Findings. Figure 5 shows the distribution of vulnerabilities according to the four stages in which the first fixed version was released. **We can observe that 50.55% of vulnerabilities were fixed after the vulnerability has been made public**, with the observation being more noticeable for vulnerabilities of medium and low severity (H = 35.96%, M = 54.57%, and L = 52.38%). Such results indicate that the majority of the *PyPi* package vulnerabilities become public before having any patch addressed to fix them.

For the remaining vulnerabilities, 30.36% of all vulnerabilities were already fixed even before their discovery. One possible explanation is that the maintainers of such affected packages prefer to disclose the vulnerability and report its information while working in silence on a fix to mitigate its impact and reduce the chances of being exploited by potential attackers. Finally, 17.09% of the vulnerabilities were fixed between the vulnerability discovery date and the vulnerability publication date.

Comparison to the npm ecosystem. Unlike *npm*, our findings show that *PyPi* package vulnerabilities tend to be fixed only after publication. In *npm*, 82% of vulnerabilities are fixed after the vulnerability discovery time and before its publication time. Our findings for *PyPi* show a different picture, with the close majority of vulnerabilities (50.55%) being fixed after their publication. Such differences can be attributed to community practices and policies in each ecosystem for reporting and disclosing vulnerabilities. We discuss these policies, their limitations, and how to better control them in section IV.

The majority of vulnerabilities (50.55%) are only fixed after the vulnerability is made public, while 30.36% are fixed before the vulnerability is first discovered, and 17.09% are fixed between the discovery and publication dates.

*D. **RQ**$_4$: How long does it take to fix a vulnerability in the PyPi ecosystem?*

Motivation. So far, we have observed that the majority of vulnerabilities are fixed after the vulnerability is reported to be discovered, either in between discovery and publication (17.09%) or after the vulnerability publication (50.55%). In this question, we focus on those vulnerabilities and investigate how long it takes for a fix patch to be released after a vulnerability is reported to be discovered. Vulnerabilities that remain un-patched for a long time after being reported and discovered can leave an open channel for successful attacks. Also, a healthy open source package should have a quick response to most vulnerability reports. Therefore, answering this question will give us important insights about the prioritization of fixing vulnerabilities of a package.

Approach. To achieve our goal, we focus now on only those vulnerabilities that get fixed after being discovered, i.e., we omit vulnerabilities that have their fixed versions before the discovery date (30.36%). For the remaining vulnerabilities, we conduct the survival analysis method to provide information about how long it takes to fix a vulnerability after being discovered. We calculate the time difference between the release date of the first fixed version and the vulnerability discovery date. Similarly to the analysis conducted in Section III-B, we use the Kaplan-Meier estimator [23] for the survival analysis. Furthermore, to understand if the severity level of a vulnerability has any impact on the time required to fix a vulnerability, we also conduct the previous analysis per severity level.

Findings. Figure 6 presents the survival probability for the vulnerabilities to be fixed after being discovered. **As we can observe from the left-side plot, the probability that a vulnerability is fixed 4 months following its discovery is 50%.** Also, we can observe that there is a small share (8.37%) of those vulnerabilities that still take more than a year to get fixed after being discovered.

The right-side plot of Figure 6 presents the previous analysis per severity level. Using the log-rank statistical method [25], we found no statistically significant difference in the time to fix vulnerabilities of different severities with a confidence level 95% (p-value = 0.41).

We further analyse the vulnerable *PyPi* packages that took more than a year for their vulnerabilities to be fixed after the discovery date, to gain insights as to why they take such a long time to address potentially impactful vulnerabilities. Upon close manual inspection, we found that 64.7% of these packages are not popular (i.e., have less than 1000 downloads) and are not actively maintained, with the latest version been released two years ago, in 2018. We expect the developers of those vulnerable packages to be unresponsive to security reports.

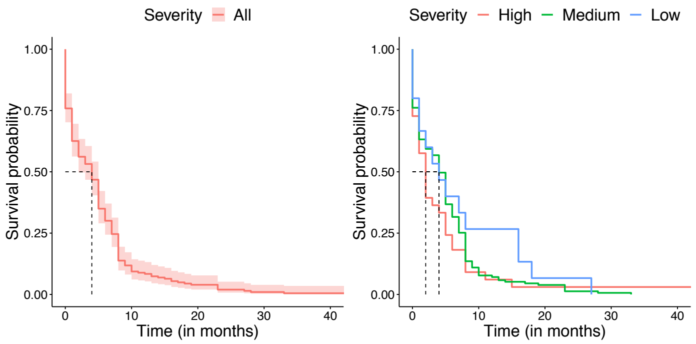


Fig. 6: Kaplan-Meier survival probability for vulnerable packages to get fixed after being discovered.

Comparison to the npm ecosystem. Our findings show that *PyPi* package vulnerabilities overall take longer to be fixed than those found in *npm*. In *npm*, it takes a median of one month to fix vulnerabilities, regardless of their severity. In *PyPi*, we found that *PyPi* vulnerabilities take a median of 4 months to release a fix after the vulnerability has been discovered.

Vulnerabilities in *PyPi* take, on median, 4 months to be fixed. The severity level of a *PyPi* vulnerability does not make a statistically significant difference for the time needed to fix the reported vulnerabilities.

IV. Discussion and Implications

In this section, we discuss more details about our results with comparison to the *npm* ecosystem (Section IV-A). Then, we highlight the implications of our study to researchers and practitioners (Section IV-B).

A. Comparison to the npm ecosystem

As shown in our comparison to the npm, some of our findings generalized also to the npm ecosystem, while others did not. Therefore, in this section, we delve into some of the reasons both ecosystems exhibit some similar characteristics as well as explanations about the divergent findings.

Vulnerability distribution. Both studies found that the number of newly discovered vulnerabilities are growing over time. We attribute the reason for this increase to the increasing popularity of open source components combined with the awareness of vulnerabilities in such components [2]. At first sight, this is a healthy sign of both ecosystems. The increase in the number of reported vulnerabilities is a result of coordinated efforts in increasing awareness and continuous process testing packages to identify the vulnerabilities before they can be exploited. However, the growth of the ecosystem calls also for the continuous and comprehensive effort for analysing package vulnerabilities to mitigate their negative impact.

We observed that the vast majority of the vulnerabilities identified in the *npm* and *PyPi* ecosystems are of medium severity.We believe that this observation is due to the fact that many of the tools used by security package maintainers to discover vulnerabilities in open source packages are not qualified to find more complex and critical issues although they are good at discovering new vulnerabilities. Robust tools that combine exhaustive techniques like program analysis, testing, and verification are required to find high-hanging vulnerabilities [28].

We observed that Cross-Site-Scripting (XSS type or CWE-79), is the most common vulnerability found in both ecosystems. The dominance of the XSS CWE vulnerability can be justified by 1) its effectiveness in granting unauthorized access into a system and the ease in which the attack method can be applied on a web application [29], [30]; and 2) the community efforts in taming this well-known vulnerability, as identifying XSS has been a top concern by OWASP [31] for more than 15 years. We conjecture that other types of vulnerabilities might be not as easily detectable, or easy to exploit, taking away the incentive of attackers in searching such vulnerabilities in the *PyPi* and *npm* ecosystems.

Vulnerabilities discovery, publication, and fix. In *npm*, the majority of reported vulnerabilities (82%) were fixed after they were discovered and before the publication date [5]. Contrasting to these findings, we found 50.55% of the *PyPi* package vulnerabilities to be fixed after the vulnerability has been made public. A possible reason for this is that 3 out of 4 vulnerabilities in *PyPi* get published right after their discovery, which reduces the time window for a vulnerability to be fixed.

To gain more insights, we investigate the protocol and policies in place for reporting and publishing vulnerabilities of both *npm* and *PyPi* ecosystems. We find that *npm* ecosystem has a protocol for reporting and publishing vulnerabilities, which enforces a 45 days waiting time before the publication of a vulnerability [32], aiming to give package developers a time window to fix the vulnerability. In *PyPi*, however, If a vulnerability is assessed to have low risk of being exploited or causing damage, the *PyPi* security team prefers to publish the vulnerability right after its discovery [33]. We noticed that most vulnerabilities (74.55%) are published as soon as they are discovered, effectively reducing the time window for a vulnerability to be fixed before publication.

An example of that, is a security issue that was found in *elementtree* package [34]. In this issue, the vulnerability could cause serious problems (high-severity level) through a Use-After-Free (UAF) [35] vulnerability related to incorrect use of dynamic memory, where the attacker causes the program to crash by accessing the memory after it has been freed. Yet, the *PyPi* security team stated that in this specific case, an attacker could not exploit this vulnerability because it requires a privileged position that is not often possible from the attacker side. Such specificities and polices is a supportive reason behind having majority of vulnerabilities being fixed after the public disclosure. Note that the risk assessment conducted by the *PyPi* security team is different from the CVSS severity level assigned to a vulnerability in the Snyk.io dataset [36].

B. Implications

In the following, we highlight the most important implications driven by our findings. We provide implications to both researchers and practitioners by discussing the aspects that the development community needs to address in order to provide a more secure development environment for package ecosystems.

There is a dire need for more effective process to detect vulnerabilities in open source packages. Our findings show that vulnerabilities in Python packages are hidden, on median, 3 years before being first discovered (RQ$_2$). These findings point to inadequacy process of testing open source packages against vulnerabilities. In fact, both *npm* and *PyPi* allows to publish a package release to the registry with no security checks exist before publishing the package. An open avenue for future research is the development of a process that ensures some basic security checks (code vetting) before publishing a release of a package. Inspired by other ecosystems, such as mobile application stores [37], [38], *npm* and *PyPi* could enforce some testing before publishing a new release of a package. Recently, there have been several research attempts to improve the security of the packages uploaded and distributed via the ecosystems, e.g., [39], [40] for *PyPi*, and Synode [41], NoRegrets [42] for *npm*. The vetting process can start with the most popular packages and move gradually, given the growth of the software ecosystems. Also, the code vetting process can focus on specific categories of security issues, e.g., malicious code or code that steals sensitive information from users, which is triggered by performing XSS attacks, the most common vulnerability found in *npm* and *PyPi* (RQ$_1$).

***PyPi* needs to employ a better protocol of publishing package vulnerabilities.** The current process of disclosing and publishing a package vulnerability in Python seems to remain ad-hoc. Our findings show that over 50% of *PyPi* package vulnerabilities were unfixed when they were first publicly announced (RQ$_3$), and took a couple of months to be fixed and released (RQ$_4$). To better control the process of reporting and disclosing package vulnerability information and limit its leakage, practitioners should refine the process to balance the advantages from an early and public-disclosure process of a vulnerability versus private-disclosure process. A possible improvement could be by forestalling the vulnerability publication until valued package users and vendors are privately notified about the vulnerability to give them a little some time to prepare properly before the vulnerability is publicly disclosed. Such controlled process is adopted by various internet networking software packages like BIND 9 and DHCP [43]. The *npm* ecosystem defines to some extent a strict timeline for reporting a vulnerability providing only 45 days for package maintainers to fix their vulnerabilities before publishing them. Yet, its efficacy is not known.

***PyPi* should deprecate packages that suffer continuously from vulnerabilities**. In our study, we observed that the vast majority of packages that take longer to fix vulnerabilities are due to project inactivity (RQ_4). A relatively new idea introduced by Pashchenko *et al.* [44] is the concept of "halted package", which is a package where the time to release the latest version surpasses by a large margin the time maintainers took to release previous versions of the package. This concept can be used to identify packages that are becoming less maintained over time, and therefore, should be replaced by a better maintained alternative in the software ecosystem.

***PyPi* and *npm* should provide package users with vulnerability information to support them with the selection process of packages.** Previous work [45] has studied several factors that influence the adoption of packages by developers. Researchers report that the occurrence of vulnerabilities and the number of vulnerabilities not quickly fixed in the packages are two important security-related factors. Currently, both *npm* [46] and *PyPi* [47] package managers provide basic quality metrics on package popularity for each package such as, list of versions, downloads count, stars count, and number of open issues. However, they lack any information on security issues. A methodology, similar to the one used in our study, could be employed to define a lightweight security metrics, to support developers when selecting their packages. An example of such metrics is to calculate the average time to patch a package vulnerability after been reported to be discovered (RQ_4). This metric will give package users insights about the prioritization of fixing vulnerabilities of the package.

***PyPi* should employ tools to audit vulnerabilities when installing the packages.** Our findings show that package vulnerabilities remain unfixed for a few months even after being publicly announced. Hence, Python applications that make use of such vulnerable packages could be exposed to vulnerabilities through their dependencies. Therefore, developers should be aware of vulnerabilities in their packages before installing them. Similar to the security audit tool provided by *npm* (i.e., `npm audit`) [48], which warn developers when installing a known vulnerable package, *PyPi* community could employ a similar tool that instantly warns developers about vulnerable packages once the vulnerable package version is installed. Recently, GitHub acquired DependaBot tool [49], a tool that tracks vulnerabilities in several ecosystems. Researchers should work on evaluating such tools to understand their effectiveness and uncover their limitations.

V. Threats to Validity

In this section, we state some threats to validity that our study is subject to, as well as the actions we took to mitigate these threats.

A. Internal Validity

The internal validity is related to the validity of the vulnerabilities dataset used in our analysis. Our dataset is restricted to a limited number of vulnerabilities (i.e., 550 security reports). We believe that many vulnerable packages may have been discovered and fixed but not yet reported. However, since Snyk team monitors more widely used packages [50], we expect our results to be representative of high-quality Python packages.

Also, we collect the *PyPi* vulnerabilities reports that were published before Jan. 2018. Results might differ if we consider vulnerability reports published after Jan. 2018. However, since a key point of our study is to compare our findings to the ones reported from npm, we collect our dataset in the same timeline as the npm study in order to make our study comparable. Furthermore, our vulnerability dataset contains more than 500 reports that cover 12 years of *PyPi* reported vulnerabilities, and many of these reports are related to a popular and most used Python packages (e.g., Django, Flask, requests).

Finally, in our analysis, we used the vulnerability severity level provided by Snyk.io to quantify their impact. However, the severity level published by Snyk.io is not necessarily uncontested, as discussed in Section IV-A, *PyPi* security advisories might have had different assessments on the severity of some vulnerabilities. Unfortunately, the severity analysis data provided by the *PyPi* ecosystem is not publicly available, therefore, we had to rely on the Snyk.io dataset as the only source of information for the severity of vulnerabilities. Also, vulnerability sources other than Snyk could be used, however, our choice of Snyk is influenced by several previous studies [5], [51], [52]

B. External Validity

External validity concerns the generalization of our results to other software ecosystems and programming languages. As shown in our comparison to the *npm*, some of our findings generalized also to the *npm* ecosystem, while other findings did not. Although our methodology and approach could be applied to other software ecosystems, results might be (and unsurprisingly so) quite different from *PyPi*, due to characteristics such as policies, community practices, programming language features and other factors belonging to software ecosystems [10], [53]. Therefore, a replication of our work using packages written in programming languages other than *PyPi* and *npm* is required to establish a more complete view of vulnerabilities in software ecosystems.

VI. Related Work

In this section, we discuss the related literature by focusing on studies that investigate software ecosystems in general, and studies that approach the link of security-related issues and software ecosystems.

Software Ecosystems. Several aspects of *Software Ecosystems* have been subject of great interest in the related literature. For example, some works analysed the ecosystem's growth [54], [55]. Fard *et al.* [54] showed that the number of dependencies in *npm* projects is 6 on average and the number is always in a growing trend.

Other works qualitatively studied the fragility and breaking changes in software ecosystems. Bogart *et al.* [53] compared Eclipse, CRAN, and *npm* in terms of practices that are used by developers to decide about causes of API breaking changes. They found that all three ecosystems are significantly different in terms of practices towards breaking changes, due to some particular community values in each ecosystem.

A few studies conducted a comparison across software ecosystems. Decan et al. [9], [11] empirically compared the dependency network evolution in 7 ecosystems (including *npm*). They discovered some differences across ecosystems that can be attributed to ecosystems' policies. For instance, the CRAN ecosystem has a policy called "rolling release", where packages should always be compatible with the latest release of their dependencies since CRAN can only install the latest release automatically. Hence, developers could face issues when updating because a change in one package can affect many others.

While the aforementioned work served as a motivation to our investigation, the focus of our study is fundamentally different. Our work can be used as to complement previous work by providing a view on another important quality metric of software ecosystems: security vulnerabilities.

Security Vulnerabilities in Ecosystems. The potential fragility of the ecosystems shown in previous studies (e.g., [8], [53]) has motivated researchers to examine security vulnerabilities, as vulnerabilities are one of the most problematic aspects of software ecosystems [1]. A study by Pham *et al.* [56] presented an empirical study to analyse vulnerabilities in the source code, and found that most vulnerabilities are recurring due to software code reuse and package adoption.

Other studies focused on analysing vulnerabilities in software ecosystems. Hejderup *et al.* [4] analysed 19 *npm* vulnerable packages, and found that the number of vulnerabilities is growing over time. Zimmermann *et al.* [6] studied the security risk of the *npm* ecosystem dependencies and showed that individual packages could impact large parts of the entire ecosystem. They also observed that a very small number of maintainers (20 accounts) could be used to inject malicious code into thousands of *npm* packages, a problem that has been increasing over time. Zerouali *et al.* [57] identified that vulnerabilities that affect *npm* packages are common in official Docker containers. A study by Zapata *et al.* [58] found that 73.3% of the 60 studied projects were actually safe because they did not make use of the vulnerable functionality.

The management of package vulnerabilities was also studied in other ecosystems like packages written in Java. Kula *et al.* [59] explored how developers respond to the available security awareness mechanisms such as library migration, and found that developers were unaware of most vulnerabilities in dependencies and prefer to use outdated versions to reduce the risks of breaking changes. Ruohomen [60] conducted a release-based time series analysis for vulnerabilities in Python web applications, and found the appearance probabilities of vulnerabilities in different versions of the applications followed the Markov model property. Also, Pashchenko *et al.* [61] conducted interviews with developers of C/C++, Java, JavaScript, and Python to understand how they manage their packages with respect to security vulnerabilities. The results indicated a high demand for high-level metrics to show how maintained and secure is a package. Our study methodology (as suggested in Section IV-B) can be employed to provide developers with such metrics for package selection process.

Ponta *et al.* [62], [63] proposed a code-centric approach to detect and mitigate open source vulnerabilities for Java and Python industry grade applications. Pashchenko *et al.* [64], [65] proposed an approach that addresses the over-estimation problem of approaches that report vulnerable dependencies in the Java ecosystem. The authors highlighted that many of the vulnerable dependencies were not actually deployed, and hence, their impact was neglected.

The work that is most close to our study is the *npm* study by Decan *et al.* [5]. Their work focused on analysing vulnerability in the *npm* package ecosystem. Inspired by their study, and supported by the fact that different ecosystems have different characteristics, we conducted our study to examine security vulnerabilities in the *PyPi* ecosystem. We studied several aspects related to vulnerability propagation, their discovery and fix timeliness. By comparing our findings with the ones reported by Decan *et al.* [5], we identified some particularities of the *PyPi* ecosystem and devise important recommendations to improve the safety of *PyPi*.

VII. Conclusion

In this paper, we conduct an empirical study of security vulnerabilities in the *PyPi* ecosystem, evaluating over than 500 package vulnerabilities that affect 252 packages. In particular, we explore vulnerabilities propagation, discovery, and fixes. Also, we compare our findings with the *npm* ecosystem [5].

Our results show that *PyPi* vulnerabilities are increasing over time, affecting the large majority of package versions. Moreover, our findings reveal shortcomings in the process of discovering vulnerabilities in *PyPi* packages, i.e., they take more than 3 years to discover them. Additionally, we observe that the timing of vulnerability patches does not closely align with the public disclosure date, leaving open windows and chances for an attacker exploitation. We note that over 50% of the vulnerabilities were patched only after public disclosure. Finally, our comparison to *npm* vulnerabilities reveals in some aspects a departure from the npm's findings, which can be attributed to ecosystems polices.

Future work should focus on broadening our study to other ecosystems and work on the development of package security tools that help practitioners at selecting and using secure software packages.

References

[1] H. H. Thompson, "Why security testing is hard," *IEEE Security & Privacy*, vol. 1, no. 4, pp. 83–86, 2003.

[2] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," *Asp. Secur. Inc*, pp. 1–26, 2012.

[3] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488.

[4] J. Hejderup, "In dependencies we trust: How vulnerable are dependencies in software modules?" 2015.

[5] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 181–191.

[6] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 995–1010.

[7] "Libraries - the open source discovery service," (Accessed on 01/10/2021). [Online]. Available: http://libraries.io/

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

[9] A. Decan, T. Mens, and M. Claes, "On the topology of package dependency networks: A comparison of three programming language ecosystems," in *Proccedings of the 10th European Conference on Software Architecture Workshops*, 2016, pp. 1–4.

[10] ——, "An empirical comparison of dependency issues in oss packaging ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 2–12.

[11] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[12] "Stack overflow developer survey 2020," https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents, (Accessed on 01/10/2021).

[13] E. S. M. Alfadel, D. Costa, "Dataset: Empirical analysis of security vulnerabilities in python packages — zenodo," https://zenodo.org/record/4158611, October 2020, (Accessed on 10/29/2020).

[14] M. Dowd, J. McDonald, and J. Schuh, *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[15] A. Nesbitt and B. Nickolls, "Libraries.io Open Source Repository and Dependency Metadata. v1.2.0," https://doi.org/10.5281/zenodo.808273, 2018, [Online; accessed 10/10/2020].

[16] Snyk.io, "The state of open-source security," 2017, [Online; Available: https://snyk.io/]. [Online]. Available: \url{https://snyk.io/stateofossecurity/pdf/The%20State%20of%20Open%20Source.pdf}

[17] "Vulnerability db — snyk," https://snyk.io/vuln, (Accessed on 10/10/2020).

[18] L. Allodi and F. Massacci, "Comparing vulnerability severity and exploits using case-control studies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 1, pp. 1–20, 2014.

[19] "semver · pypi," https://pypi.org/project/semver/, (Accessed on 10/10/2020).

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

[21] MITRE, "Cwe," https://cwe.mitre.org/about/index.html, (Accessed on 10/10/2020).

[22] O. Aalen, O. Borgan, and H. Gjessing, *Survival and event history analysis: a process point of view*. Springer Science & Business Media, 2008.

[23] E. L. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American statistical association*, vol. 53, no. 282, pp. 457–481, 1958.

[24] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, 2019.

[25] V. Bewick, L. Cheek, and J. Ball, "Statistics review 12: survival analysis," *Critical care*, vol. 8, no. 5, 2004.

[26] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2201–2215.

[27] C. Sabottke, O. Suciu, and T. Dumitraș, "Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 1041–1056.

[28] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[29] J. Thomé, L. K. Shar, D. Bianculli, and L. Briand, "Security slicing for auditing common injection vulnerabilities," *Journal of Systems and Software*, vol. 137, pp. 766–783, 2018.

[30] R. Johari and P. Sharma, "A survey on web application vulnerabilities (sqlia, xss) exploitation and security engine for sql injection," in *2012 International Conference on Communication Systems and Network Technologies*. IEEE, 2012, pp. 453–458.

[31] OWASP, "Owasp," https://www.owasp.org/index.php/Main_Page, 2019, (Accessed on 10/10/2020).

[32] "Reporting a vulnerability in an npm package — npm documentation," https://docs.npmjs.com/reporting-a-vulnerability-in-an-npm-package, (Accessed on 10/10/2020).

[33] PyPi, "Security · pypi," https://pypi.org/security/, 2018, (Accessed on 10/10/2020).

[34] "Issue 27863: multiple issues in _elementtree module - python tracker," https://bugs.python.org/issue27863, (Accessed on 10/10/2020).

[35] "Cwe - cwe-416: Use after free (3.3)," https://cwe.mitre.org/data/definitions/416.html, (Accessed on 10/10/2020).

[36] "Scoring security vulnerabilities 101: Introducing cvss for cves — snyk," https://snyk.io/blog/scoring-security-vulnerabilities-101-introducing-cvss-for-cve/, (Accessed on 10/10/2020).

[37] "Android – google play protect," https://www.android.com/intl/en_ca/play-protect/, (Accessed on 10/27/2020).

[38] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 229–240.

[39] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2020, pp. 509–514.

[40] ——, "Poster: Towards using source code repositories to identify software supply chain attacks," in *CCS '20*, 2020.

[41] C.-A. Staicu, M. Pradel, and B. Livshits, "Understanding and automatically preventing injection attacks on node. js," Tech. Rep. TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science, Tech. Rep., 2016.

[42] G. Mezzetti, A. Møller, and M. T. Torp, "Type regression testing to detect breaking changes in node. js libraries," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[43] "Internet systems consortium," https://www.isc.org/#, (Accessed on 10/10/2020).

[44] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.

[45] E. Larios-Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: The practitioners' perspective," *arXiv preprint arXiv:2005.12574*, 2020.

[46] "lodash - npm," https://www.npmjs.com/package/lodash, (Accessed on 10/10/2020).

[47] "Pillow · pypi," https://pypi.org/project/Pillow/, (Accessed on 10/10/2020).

[48] "Auditing package dependencies for security vulnerabilities — npm documentation," https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities, (Accessed on 10/10/2020).

[49] "Dependabot," https://github.com/dependabot, (Accessed on 10/28/2020).

[50] "How snyk finds out about new vulnerabilities – knowledge center — snyk," https://support.snyk.io/hc/en-us/articles/360003923877-How-Snyk-finds-out-about-new-vulnerabilities, (Accessed on 10/24/2020).

[51] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 559–563.

[52] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the release, adoption, and propagation of npm vulnerability fixes," *Empirical Software Engineering*, 2019.

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

[54] A. M. Fard and A. Mesbah, "Javascript: The (un) covered parts," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 230–240.

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

[56] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 447–456.

[57] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the impact of outdated and vulnerable javascript packages in docker images," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 619–623.

[58] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 559–563.

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

[60] J. Ruohonen, "An empirical analysis of vulnerabilities in python packages for web applications," in *2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 2018, pp. 25–30.

[61] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," *Proc. of CCS'20*, 2020.

[62] S. E. Ponta, H. Plate, and A. Sabetta, "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)*. IEEE, 2018, pp. 449–460.

[63] ——, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, 2020.

[64] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transactions on Software Engineering*, 2020.

[65] ——, "Vulnerable open source dependencies: Counting those that matter," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.