Empirical Software Engineering (2023) 28:139
https://doi.org/10.1007/s10664-023-10355-2

Technical leverage analysis in the Python ecosystem

Ranindya Paramitha[1] · **Fabio Massacci**[1,2]

Accepted: 5 June 2023 / Published online: 13 October 2023

Abstract

Context: Technical leverage is the ratio between dependencies (other people's code) and own codes of a software package. It has been shown to be useful to characterize the Java ecosystem and there are also studies on the NPM ecosystem available.
Objective: By using this metric we aim to analyze the Python ecosystem, how it evolves, and how secure it is, as a developer would perceive it when deciding to adopt or update (or not) a library.
Method: We collect a dataset of the top 600 Python packages (corresponding to 21,205 versions) and used a number of innovative approaches for its analysis including the use of a two-part statistical model to deal with excess zeros, a mathematical closed formulation to estimate vulnerabilities that we confirm with bootstrapping on the actual dataset.
Results: Small Python package versions have a median technical leverage of 6.9x their own code, while bigger package versions rely on dependencies code a tenth of their own (median leverage of 0.1). In terms of evolution, Python packages tend to have stable technical leverage through their evolution (once highly leveraged, always leveraged). On security, the chance of getting a safe package version when choosing a package is actually better than previous research has shown based on the ratio of safe package versions in the ecosystem.
Coclusions: Python packages ship a lot of other people's code and tend to keep doing so. However, developers will have a good chance to choose a safe package version.

Keywords Dependencies · Software libraries · Technical leverage · Empirical analysis · Vulnerabilities · Python ecosystem · Security

1 Introduction

Many software packages are developed using third-party libraries, which are called dependencies in developers' jargon. This practice is in line with the principle of software re-use

Communicated by: Martin Monperrus

✉ Ranindya Paramitha
ranindya.paramitha@unitn.it

Fabio Massacci
fabio.massacci@ieee.org

1 Department of Information Engineering and Computer Science, Università degli Studi di Trento, Trento, Italy

2 Foundational Security, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

(Frakes et al., 2005), but could also bring risks to software projects (Cox et al., 2015; Lauinger et al., 2018), or could be even useless as shown in the study by Soto-Valero et al. (2021b). Yet, it is hard to provide an indicator of such potential security risk that could be easily used by developers (Dashevskyi et al., 2016).

A recent development has been the introduction of the notion of technical leverage by Massacci et al. (2021), which mirrored the financial intuition behind internal code defects and technical debt (Cunningham, 1992). Technical leverage of a software package is the ratio between the lines of code from dependencies and own code, written by the package developer(s). Their research found that if Java libraries have a technical leverage higher than the industry mean (4x leverage), then they are 60% more likely to be vulnerable than libraries with lower technical leverage.

However, the findings relative to an ecosystem might not transfer to a different one for a variety of reasons. For example, some previous works by Decan et al. (2019) and Kikas et al. (2017) have shown the importance of analyzing different ecosystems as the underlying dependency networks may evolve in different ways. The nature of the programming language may also have an impact; for instance, Java is first compiled into bytecode that runs on the Java Virtual Machine (JVM), i.e. a software-based interpreter, while Python is a language that is directly interpreted (Khoirom et al., 2020).

The purpose of this paper is to extend the existing studies in the Maven ecosystem done by (Massacci et al., 2021) and npm ecosystem Anonymous (2022) to the PyPI ecosystem. In comparison to npm and Java, Python is particularly interesting because of its target applications and its usage for data science and related fields. Furthermore, Python's growing popularity and the existence of dependency-related attacks in this ecosystem, including the recent attack on PyTorch (Pernet, 2023) and other types of attack like typosquatting and combosquatting (Vu et al., 2020), makes it an interesting target of analysis.

Further, previous studies on dependencies (Wang et al., 2020; Prana et al., 2021) did not clearly distinguish between packages and versions during the analysis. We found that the usage of package granularity better reflects what developers actually experience (you choose a package first, and then a version) and therefore adopted this perspective.

RQ1: *How is technical leverage distribution in the Python ecosystem?*

To this extent, we analyzed the history of the top 600 packages in PyPI for a total of 21,205 different versions. We found that smaller Python packages (with less than equal to 100 KLOC) have higher statistically significant technical leverage than larger packages. Most of the small package versions ship other people's codes more than 6.88 times bigger than their own code, while most of the big package versions ship other people's codes around a tenth of their own. From a technical perspective, in this work, we elected to use a statistical method (Lachenbruch, 2022) hardly used in software engineering that takes into account the presence of excess zeros so that we can deal uniformly with packages with no leverage and package with large leverage.

While the previous work for Java (Massacci et al., 2021) focused more on static measures with limited dynamic measures (only change direction analysis), we believe the dynamic measures to understand how technical leverage evolves over time are also important. Hence our next research question:

RQ2: *How does the technical leverage metric change when we move to newer versions in a package?*

We regress the technical leverage of package versions and their succeeding version. Our analysis shows that the technical leverage tends to stay stable through the package evolution, in other words: if a package is highly leveraged, it will stay so.

Another gap is the need of understanding the chance of getting a safe/vulnerable package version when choosing a *package*. Most papers that analyze ecosystems typically focus on reporting the ratio between vulnerable or not vulnerable versions (Alfadel et al., 2021; Bagmar et al., 2021). Massacci et al. (2021) used the interquartile distribution of leverage of a package vs. the number of vulnerabilities in the package to show the correlation, but they did not actually estimate the individual probabilities. This calculation would be necessary as in real life, developers are choosing a package first before choosing a package version (Pashchenko et al., 2020; Derr et al., 2017; Kula et al., 2018). In several cases, as noted in Dashevskyi et al. (2016), the choice of the package is not actually a choice as it is dictated by external constraints and only the choice of the version (or whether to update to a new version) is available.

So in this paper we propose to estimate the risk of adopting a vulnerable package as opposed to the risk of choosing a version to propose a better estimation with a *closed mathematical formula*. We also compared this formula to bootstrapping simulations and found it to be a statistically good approximation.

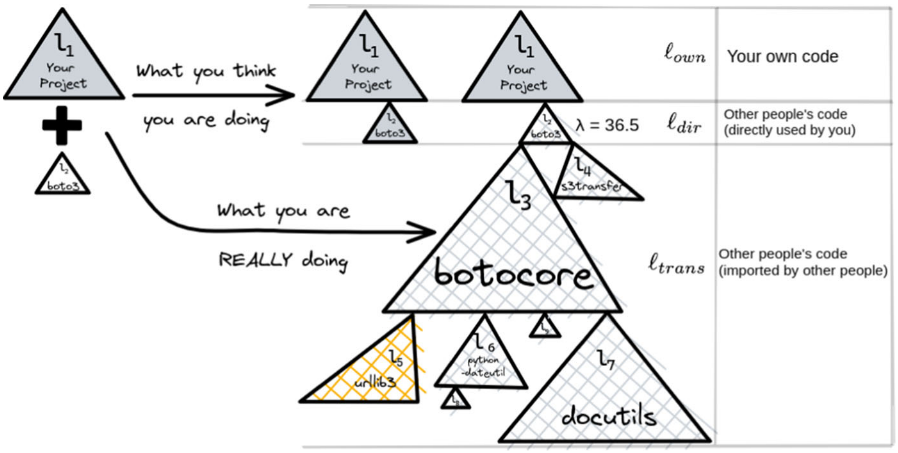
RQ3: *Does technical leverage capture the risk of having vulnerabilities?*

Glancing at the ecosystem, more than a quarter (26.5%) of our package version dataset is vulnerable. Based on this information, one can have the impression that if developers pick randomly 10 package versions, most likely 3 of them would be vulnerable. This percentage would be the reported result *if* we were using the same method as previous works (Alfadel et al., 2021; Wang et al., 2020). Yet this method can be inaccurate as its findings can be dominated by a handful of packages with many vulnerable versions. If you do not choose those packages you might actually be safe. We, therefore, propose a better estimation of what developers actually experience. We also compute the probability of getting a safe package/package version when choosing a package with a *closed mathematical formula*. We compared this formula to bootstrapping simulations on the dataset and found it to be a statistically good approximation. Then we found that the chance of getting a safe package version is actually higher than the direct percentage of vulnerable versions: from 10 package versions, only 2 will be vulnerable (and only 1 for packages with no dependencies).

The rest of the paper is structured as follows: the next section §3 defines several terminologies used in this work, including the notion of technical leverage from Massacci et al. (2021). We then present our analysis procedure in §4 and in §5, we describe the data we used in this analysis. The analysis results are presented and discussed in §6, §7, and §8. We then elaborate on previous related works on dependencies and their impacts on software security, large-scale analysis, and attack surface analysis on §10, followed by several threats to validity of our work in §11. Finally, the last section §12 describes the conclusion and future works of this research.

2 Motivating Example

Figure 1 illustrates a real use case that can happen in the Python ecosystem. As a developer, you may want to adopt a `boto3` as a dependency. It offers some useful functionality, there is a version that seems compatible with your code (1.10.12), the package is considerably small (only 3KLoC), and it has no vulnerability (Limited, 2023). Therefore, you decide to adopt this version of `boto3`, thinking that you only leverage on a small package and slightly increase your uncontrollable risk.



Adopting a "seemingly small" package of three (3) thousand lines of code can bring a humongous amount of dependency code to the project and increase significantly the ratio of uncontrollable risk (the area of triangles is proportional to the corresponding code size). The technical leverage of `boto3` ($\lambda = 36.5$x) captures in a single number this potential risk. Here, the risk is actually realized by a vulnerability in the `urllib3` library.

Fig. 1 What developers *think* they are doing vs. What they are *really* doing

However, what *actually* happens when you adopt this package is that the package you adopt depends on other packages. These dependencies may increase your uncontrollable risk by some orders of magnitude. In this example, the added code base of 3KLoC is multiplied by $\lambda = 36.5$x. Further, one of the "hidden" (transitive) dependencies actually has a vulnerability: package `urllib3` with CVE-2021-33503 (NIST, 2023). These "hidden" vulnerable dependencies and the increase of "unknown" uncontrollable risks motivate our empirical research on how packages are dependent on each other in the Python ecosystem and how it affects the security of the packages.

3 Terminology

In this paper, we use the terminology that is commonly utilized among the users of the Python Package Index (PyPI). Several of these terms are also common to the Java environment, as consolidated in Pashchenko et al. (2018):

- A *package* is a separately distributed software component. In the Python ecosystem, it typically consists of several modules, standard code organization units for non-trivial Python code. They are *.py* files containing a logically grouped set of classes or methods and usually have a page in the Python Package Index (PyPI) Foundation (2022). For a specific version of a package, i.e. `boto3` version 1.22.3, we use the term: *package version*.
- A *dependency* is a package version whose functionalities are used by another package version (*dependent* version).
- A *direct dependency* (ℓ_{dir}) is a specific group of dependencies, that is *directly* used in the dependent package version.

- A *dependency tree* is a representation that connects package versions (as nodes) with other package versions through directed edges, which connect the dependency from dependent package versions to their direct dependencies.
- A *transitive dependency* (ℓ_{trans}) of a package version at a root of a dependency tree includes all package versions in the tree that are connected to the root through a path with more than one edge.
- A *standard dependency* (ℓ_{std}). As the Java ecosystem, Python packages also leverage some functionalities from standard packages of the programming language. These functionality dependencies are considered as the size of the baseline of programming language packages. We do not consider them here as we are comparing packages in the same ecosystem.

To measure the exposure to dependencies we use the *Technical Leverage* metric as introduced by Massacci et al. (2021). Similar papers have used similar concepts such as Gkortzis et al. (2021) on software reuse. We refer to the original paper for the discussion on the parallel with finance and technical debt (Allman, 2012). We only introduce the minimal notation that we use to make the paper self contained. To quantitatively measure the size of package versions and their dependencies, we use the following metrics:

- *Own code size* (ℓ_{own}) as the number of lines of own code in all files of a package.
- *Dependency code size* (ℓ_{dep}) as the number of lines of code in the dependencies. This includes direct (ℓ_{dir}), transitive (ℓ_{trans}), and standard dependencies (ℓ_{std}).
- *Total code size* (ℓ_{total}) as the sum of *own code size* and *dependency code size*.

As Massacci et al. (2021), in this work we also chose Lines of Code (LoC) as a measure of code size as it is a simple metric that can be easy to understand and transferable to different ecosystems. Other metrics may be exclusive to a specific language, e.g. the number of pointers in C or objects in Java. Function points have also been used in the literature to estimate effort (Huijgens et al., 2017) but computing them is a complex process and several approximations have been proposed (Jorgensen et al., 2006). Most importantly, function points are used to estimate the cost *before* the software has been built. After the software has been built, which is our case, both old (Slaughter et al., 1998) and new research Banker et al. (2021) in management science has shown that the number of LoCs is a good proxy for financial outcomes. For example, Banker et al. (2021) has shown that using modified, not compliant LoCs to measure technical debt has a strong correlation with a number of financial indicators for firm performance.

To measure a package, we first download the package from PyPI using `pip download`. We are using the default of `pip download` which by default is using the most recent stable version for dependency resolution unless it is explicitly told otherwise. The package should be downloaded with all its dependencies inside different folders. We then consider all these other package folders as dependencies except the package's main one (named "<PACKAGE_NAME–PACKAGE_VERSION>"). After we separate the own and dependency codes, we then iterate through `.py` files and apply `python-loc-counter` package (Foundation, 2023d) to get the LoCs of the source code (not including comments and blank lines).

To measure technical leverage, we need to identify 'third-party' code. The qualifier 'third-party' is important as noted by Pashchenko et al. (2018): developers might have decided to structure their own code in separate packages. They might be mistakenly counted as other people's code while in reality are developed within the same project and by the same developers, hence, should be counted as their own code. In Maven, all $<$ group $>$.* packages should be counted as the same project, even though they are dependencies. This gives us a technical means to easily assess direct vs transitive dependencies. However, assessing this is much harder in Python. This possibility of counting first-party dependencies as third-party ones is a potential threat to the validity of any analysis in Python so we also discuss it at length in Section 11. For this work, we only handle the case that can be processed automatically with close to no errors: a package version n depending on a previous version $n - x$ of exactly the same package. For this case, we remove the previous version of the same package from the dependency list which consequently reduces the technical leverage.

Definition 1 The original *technical leverage* λ of a library is the ratio between the size of code imported from third-party libraries including the baseline of programming language libraries ℓ_{std} and the own code size of the library:

$$\lambda = \frac{\ell_{dir} + \ell_{trans} + \ell_{std}}{\ell_{own}} \tag{1}$$

In this work, we omit the standard dependencies ℓ_{std} from the application of Definition 1. The calculation of these standard dependencies only matters for comparing projects across ecosystems, i.e. a single analysis putting together Java *and* Python *and* C projects. Since the ℓ_{std} is the same for all packages running Python, this will just be a uniform constant shift that does not require calculation. Besides, standard libraries are generally mature and rarely contain vulnerabilities. Therefore, including standard libraries will not add any value to the analysis besides making technical leverage look artificially bigger.

Definition 2 The size changes $\Delta\ell_X$ are the size difference between two consecutive versions (i and $i+1$) of a package: $\Delta\ell_{dep}$ is dependency size change and $\Delta\ell_{own}$ is own size change.

$$\Delta\ell_X = \ell_{X_{i+1}} - \ell_{X_i} \tag{2}$$

Definition 3 The change direction θ characterizes the type of evolution of a library between two consecutive versions.

$$\theta = \arccos\left(\frac{\Delta\ell_{dep}}{\sqrt{\Delta\ell_{own}^2 + \Delta\ell_{dep}^2}}\right) * \begin{cases} +1 \; if \, \Delta\ell_{own} > 0 \\ -1 \text{ otherwise} \end{cases} \tag{3}$$

Figure 2 from Massacci et al. (2021) also categorized the change direction (θ) into four main directions of a library evolution.

In addition to these definitions, we also use the difference of technical leverage to show changes in dependence (to others code) from one package version to the next.

Definition 4 ***(NEW)*** The technical leverage differential $\Delta\lambda$ shows the dependency-own ratio change between two package versions r_0 and r_1 :

$$\Delta\lambda = \lambda_{r_1} - \lambda_{r_0} \tag{4}$$

When the technical leverage of a new package version (r_1, the version to be adopted by developers), is bigger than the older package version (r_0), it says that developers become exposed to others' code instead of their own ($\Delta\lambda > 0$). On the other hand, if the difference is negative ($\Delta\lambda < 0$), developers become more dependent on their own code instead of others', which lowers the ratio of uncontrollable risk in the software project.

While the amount of risk depends on the total amount of code, what we are interested in here is the *ratio* of "uncontrollable" risk that comes from codes that are not our own wrt the risk due to one's own attack surface. If "both" internal and external code increase, the risk does increase, but the "ratio" is the one that remains constant.

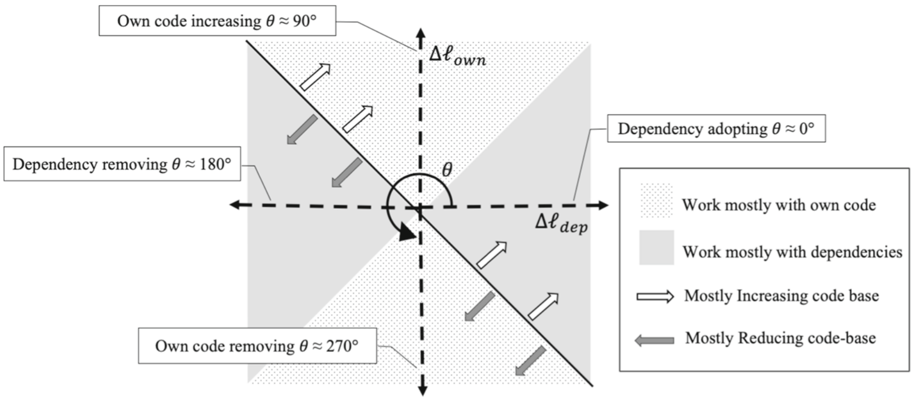

Own code increasing $\theta \approx 90°$
$\Delta\ell_{own}$
Dependency adopting $\theta \approx 0°$
Dependency removing $\theta \approx 180°$
θ
$\Delta\ell_{dep}$
Own code removing $\theta \approx 270°$
Work mostly with own code
Work mostly with dependencies
Mostly Increasing code base
Mostly Reducing code-base

Fig. 2 Change direction and developers' behavior (Massacci et al., 2021)

4 Methodology and Analysis Procedure

In this section, we describe the methodology that we have used for the analysis of the dataset.

4.1 RQ1: Python ecosystem overview.

To answer RQ1, we want to determine whether there is a statistically different distributions of leverages between small and large packages in terms of own size. Since there are many package versions with no dependencies ($\lambda = 0$), we use a statistical test by Lachenbruch (2022), composed by two separated tests for zero and non-zero data. To the best of our knowledge this is the first time that such a test has been used in mining software repositories and we think it is also an innovative contribution of this paper.

Lachenbruch's statistical test addresses the problem in statistical analysis for datasets with excess zeroes. It combines a discrete point-mass variable (binomial test) for the zero part and a continuous random variable (can be t-test, Wilcoxon test, Kolmogorov-Smirnov test, or some other tests) for the rest. The idea is that the distribution of the packages belonging to a class c is captured by the following formula:

$$f_i(\lambda, d) = [p_{\lambda=0_i}^{1-d}\{(1 - p_{\lambda=0_i})h_i(\lambda)\}^d] \tag{5}$$

For our case, we have $d = 1$ for the case when package versions have dependencies ($\lambda > 0$) and $d = 0$ otherwise. $p_{\lambda=0}$ is the probability of having no dependencies and $h(\lambda)$ is a lognormal fit of the subset of the data which has dependency/ies ($\lambda > 0$). To compare two classes with this distribution we run two tests for λ from (5) both when $d = 0$ and $d = 1$ where the classes to compares are small libraries ($i = 1$) and big libraries ($i = 2$). Lachenbruch (2022) has shown that we can use a binomial proportion test as the first part of the test (B) and T-test for the second part (T), which is then combined into a test that has a chi-square distribution using (6).

$$X^2 = B^2 + T^2 \tag{6}$$

4.2 RQ2: Technical leverage evolution across versions.

When analyzing the evolution of a package, we want to find out how similar two consecutive versions are. For this purpose, we order the package versions by their release date. In rare cases where there are any faulty intermediary versions for that we do not have the data, we skip those faulty versions. We cannot use the preceding test as-is because it is a test for difference between two classes. However, we can use the same idea of proving first the case for package versions with no dependencies ($\lambda = 0$) and the the case for package versions with dependencies ($\lambda > 0$).

The first test is therefore to determine if packages with no dependencies will remain so and packages with dependencies will continue to use them rather than dropping them. These scenarios generate a latin square with the two states of the present version ($\lambda = 0, \lambda > 0$) and the two states of the next version, on which then we compute the Fisher Exact Test. Regarding package versions with dependencies ($\lambda > 0$), we regress the technical leverage of package versions with the technical leverage of their succeeding package versions.

A further restriction would be to determine whether the generation of the next leverage state is simply a Markov process: the leverage of the next state *only* depends on the previous leverage state. To this extent, we can use the tests mentioned by Bickenbach et al. (2001).

4.3 RQ3: Technical leverage vs. vulnerabilities in Python.

Our focus here is the relation between technical leverage and security risk, measured by the presence of security vulnerabilities. For each package version in our database, we count the vulnerabilities based on the vulnerabilities published in Snyk (SnykDB, 2022) until May 2022. When the data was gathered, Snyk was still publishing all vulnerabilities back until 2013. The number of vulnerabilities considered in this work is the sum of the number of direct vulnerabilities (vulnerabilities on the project itself) and indirect vulnerabilities (vulnerabilities from dependencies). On self-introduced vulnerabilities (vulnerabilities that are deliberately introduced by developers in their own code (Massacci et al., 2022)), we consider them a relatively new phenomenon and they are so rare to make the news. It is unclear how many of those kinds are actually present so we did not make this distinction.

The first step is to analyze the ecosystem considering the ensemble of all package versions as usually reported in ecosystem empirical analysis (Alfadel et al., 2021; Bagmar et al., 2021). This view is good to understand the view of the ecosystem in general. However, any analysis based on versions will be biased by packages with many versions. Different packages from different developers can have different ways of versioning: some generate minor patches for a really small change, and some others only generate minor patches for a huge set of changes.

Let's see an example of a small "ecosystem" with 3 packages: `ansible`, `anyio`, and `antlr4-python3-runtime`. `ansible` has 158 versions, all of which are vulnerable (0% safe). `anyio` has 40 versions and `antlr4-python3-runtime` has 16 versions, all of which are not vulnerable (100% safe). If we use package version granularity, we will get $56/214 = 26.17\%$ are safe. However, this result is heavily influenced by `ansible` which has 158 vulnerable versions. If we instead use package granularity, we have a package that is 0% safe and two packages that are 100% safe. Therefore, if we take the average among these three, we will get $0\% + 100\% + 100\% = 200\%$; $200\%/3 = 66.67\%$, which is much higher than the previous result with package version granularity.

Another reason to use package granularity instead of package version is that it reflects better how developers are choosing dependency on the real-life development process Dashevskyi et al. (2016). A survey by Derr et al. (2017) on Android developers shows that most of them use search engines to search for libraries. Therefore, it is intuitive that they will pick first a library (or in this paper, we use the terminology *a package*), then a version of it. Other interviews with developers by Pashchenko et al. (2020) mentioned that developers rely more on high-level information to select a package than on the package's source code. This shows us that developers are *not* directly choosing a package version. They first choose a *package*, and then choose one version of it. Therefore, we want to observe a broader view from packages (coarser granularity), specifically to know whether a group of packages with a given average technical leverage is statistically more prone to being vulnerable than another.

To accomplish this observation, we conducted two steps: (1) mathematical probability calculation and (2) simulations. Before conducting these two steps, we cluster the packages into three clusters based on their average technical leverage:

- No dependencies ($\lambda = 0$)
- Below the industry average ($0 < \lambda \leq 4$)
- Above the industry average ($\lambda > 4$)

The choice of $\lambda = 4$ as the boundary for group division is based on the reported value 400% for the industry average from BlackDuck (Pittenger et al., 2016). To statistically test the differences among these clusters, we compute a *Chi-square contingency test* on the number of packages with/ without vulnerabilities in these three groups.

4.3.1 Probability of selecting an unsafe version

At first, we need to calculate from the data the probability that a package with a given technical leverage is vulnerable and then whether the such probability is actually significantly different among the distinct technical leverage clusters.

To calculate the probability of getting a safe package version with a given average technical leverage we identified three conditions as described in the first part of Table 1. The variables we used are then described in the second part of the same table.

At first, we compute the probability that a package will be totally safe: no version of the package has ever had a vulnerability and, if past is prologue, will be unlikely to have one.

$$Pr[safe\ pkg|\lambda] = \frac{N_{V=0,\Lambda=\lambda}}{N_{\Lambda=\lambda}} \tag{7}$$

The next case is picking a vulnerable version of a package. Unfortunately, one cannot simply gather all package versions belonging to a package with a given leverage and calculating the number of safe versions over the total number of package versions, as this can be biased by a big package with many package versions.

So we estimate the probability by multiplying the probability of a package being chosen and the sum of the probabilities of getting a safe version in each package. As a first approximation, which we will refine later in the paper, we assume that the probability of a package being chosen in the first place is the same across different packages.

$$Pr[safe\ vers|\lambda] = \frac{1}{N_{\Lambda=\lambda}} \sum_{\substack{\text{package } i \\ \wedge \mathbb{E}[\lambda_{i,j}|i] = \lambda}} \left(1 - \frac{v_i}{n_i}\right) \tag{8}$$

Finally, instead of assuming that the probability of a package being chosen is always the same, we use the *downloads* of the packages as a reference (from van Kemenade et al.

Table 1 Variables for the Estimation of Vulnerabilities

Probability variable	Description
$Pr\left[safe\ pkg\|\lambda\right]$	Probability that a package with a given average leverage will have no vulnerable version
$Pr\left[safe\ vers\|\lambda\right]$	Probability that one will get a version with no vulnerabilities for a package with given average leverage
$Pr\left[safe\ vers\ downl\|\lambda\right]$	Probability that one will get a version with no vulnerabilities for a package with given average leverage taking into account the popularity (number of downloads) of the package
Variable	Description
$\lambda_{i,j}$	Technical leverage of individual version j of package i
$\mathbb{E}[\lambda_{i,j}\|i]$	Average leverage of versions j for package i
$\delta_{i,j}$	If version j of package i is vulnerable and zero otherwise
n_i	Number of version j of package i
v_i	Number of *vulnerable* versions j for package i
d_i	Number of downloads of package i
$N_{V=0}$	Number of packages i where $v_i = 0$
$N_{\Lambda=\lambda}$	Number of packages i where λ is the average of leverage across versions $\mathbb{E}[\lambda_{i,j}\|i] = \lambda$
$N_{V=0,\Lambda=\lambda}$	Number of packages i where where $v_i = 0$ and $\mathbb{E}[\lambda_{i,j}\|i] = \lambda$
$D_{\Lambda=\lambda}$	Number of downloads of all packages i where λ is the average of leverage across versions $\mathbb{E}[\lambda_{i,j}\|i] = \lambda$
N_{tot}	Total number of package under observation

(2022) v. August 4 th, 2022). We use the number of downloads as it gives the view of how many people are using the package, assuming that the more people using a package, the more interested attackers are in exploiting the package. Pashchenko et al. (2020) shows that developers most likely look directly at PyPI to determine which package to download, as well as mentioned in other studies (Derr et al., 2017; Kula et al., 2018).

$$Pr[safe\ vers\ downl|\lambda] = \frac{1}{D_{\Lambda=\lambda}} \sum_{\substack{\text{package } i \\ \wedge \mathbb{E}\left[\lambda_{i,j}|i\right] = \lambda}} d_i \left(1 - \frac{v_i}{n_i}\right) \tag{9}$$

At this point, we want to know the difference in the probabilities of getting a safe version among the three groups that we have identified ($\lambda = 0, 0 < \lambda \leq 4, \lambda > 4$). To this extent, as we can not assume that the probability distribution would be Gaussian, we run the *Kruskal-Wallis test* (one-way ANOVA on ranks). To account for the multiplicity of downloads, for each "sample" (package) in the three groups, the probability of the corresponding sample $(1 - \frac{v_i}{n_i})$ was replicated for its number of downloads (d_i). We then have 3 lists of values: a list of probabilities that a package is safe for $\lambda = 0$, a list of probabilities that a package is safe if λ is between zero and four, and a list for the remaining packages. On these lists we ran a Kruskal-Wallis test to determine if there is a statistically significant difference between them.

The formula in (8) ignores the popularity of packages (assuming all packages are being equally used by developers) and (9) considers it, assuming more popular packages will be subject to more intense scrutiny. The first formula ignores the fact that security researchers are unlikely to obtain fame for the discovery of a vulnerability in packages nobody uses. Therefore a package might not have *known* vulnerabilities for the simple reason that nobody looked at it. The second equation emphasizes it. We keep both formulas to cover both sides of the spectrum.

4.3.2 Simulation

To see whether our equations reflect what developers will actually get, we ran simulations to simulate how developers choose a package version. First, we ran a simulation where all packages have the same probability of being chosen, and then we ran a second simulation where the probability of being selected is proportional to the number of downloads. The algorithm of the simulation is shown in Algorithm 1. The weighted random function we use to get the weighted probability is then shown in Algorithm 2.

Algorithm 1 Simulation of developers choosing random package version.

```
    input  : List of package versions with technical leverage and number of vulnerabilities ($data$)
    input  : Category of observation ($cat$)
    input  : Dictionary of packages and their downloads count ($downloads\_dict$) opt.
    input  : Number of trials ($trials$) opt.
    output : Mean ($mean$) and standard deviation ($stdev$) of the simulation
1   $packages \leftarrow getAllPackagesInCategory(data, cat)$ // Get the list of all
        packages in the $cat$
2   $sample \leftarrow length(packages)$ // Default sample size is the number of
        packages in the $cat$
3   if $isNone(trials)$ then $trials \leftarrow 10000$ // Default trials to be done
4   ;
5   ;
6   $trial\_results \leftarrow [\,]$ // Initialise result list
7   for $j|j \in [0, trials)$ do
        // Do sampling
8       $count\_safe \leftarrow 0$ // Initialise counter
9       for $i|i \in [0, sample)$ do
10          $chosen\_pkg \leftarrow random(packages)$ // Randomly select a package
11          if not $isNone(downloads\_dict)$ then
                // Do weighted random if the list of downloads is given
12              $chosen\_pkg \leftarrow weightedRandom(packages, downloads\_dict)$
13          $data\_pkg \leftarrow getAllversions(data, chosen\_pkg)$ // Get all versions of the
                chosen package
14          $chosen\_inst \leftarrow random(data\_pkg)$ // Randomly select an version
15          if not $isVuln(chosen\_inst)$ then
                // Add the counter if chosen version is not vulnerable
16              $count\_safe \leftarrow count\_safe + 1$
17      $prob \leftarrow count\_safe/sample$ // Calculate probability
18      $trial\_results.addElement(prob)$ // Add trial result to list
19  $mean \leftarrow getMean(trial\_results)$ // Calculate mean of trial results
20  $stdev \leftarrow getStdDev(trial\_results)$ // Calculate std. dev. of trial results
```

We ran both simulations for the 3 groups of package's technical leverage we defined earlier, each with 10,000 trials, and the number of samples is the number of packages in each group. We then check whether the results of the formula are inside the confidence interval of the simulations in Subsection 4.3.2 to see whether the formula reflects the simulation result.

We also want to understand the difference between using the downloads in calculation vs. assuming all packages have the same probability of being chosen. For this reason, we run a *Chi-square contingency test* on the number of safe versions we got from normal vs. weighted simulation (Subsection 4.3.2) for the 3 groups of technical leverage.

Algorithm 2 Weighted random using downloads data.

```
input  : Dictionary of packages and their downloads count ($downloads\_dict$)
input  : List of all packages in a specific category ($packages$)
output : Weighted randomized chosen package ($chosen\_package$)
1  Function $weightedRandom(packages, downloads\_dict)$ :
2      $cumulative\_weight \leftarrow \{\}$ // Initialise cumulative weight dictionary
3      $total \leftarrow 0$ // Initialise cumulative counter
4      forall $pkg \in packages$ do
           // Iterate packages
5          $total \leftarrow total + downloads\_dict[pkg]$ // Add to cumulative counter
6          $cumulative\_weight.addElement(total, pkg)$ // Add {total: package} to
               dictionary
7      $random\_num \leftarrow random(1, cumulative\_weight.getLastKey())$ // Randomly select
           a number
8      forall $elmt \in cumulative\_weight$ do
           // Iterate cumulative_weight, continue while
               $random\_num > elmt.key()$
9          if $random\_num < elmt.key()$ then
               // Return the package name if found
10             return $elmt.value()$
```

5 Data Selection

We analyzed the top 600 Python packages by van Kemenade et al. (2022) (v. May 2022) which resulted in 21,205 package versions in PyPI (Foundation, 2022) from 482 different packages. Half of the packages have at least 24 package versions, with a maximum of 1438 package versions in one package. While the maximum of own size (l_{own}) and dependency size (l_{dep}) are around the same number, the median of dependency size is twice the own size. Regarding the dependency adoption, 61.63% package versions in our dataset have at least one dependency, and the rest (38.37%) have no dependency (Table 2).

6 RQ1: Python ecosystem overview

6.1 Technical Leverage

The distribution of technical leverage in the Python ecosystem is shown in Fig. 3. We categorized the package versions into two categories based on their own size (l_{own}): small package versions with $l_{own} \leq 100KLOC$ and big package versions with $l_{own} > 100KLOC$. The details on categorized package versions in our data selection are shown in Table 3, which

Table 2 Descriptive Statistics of Python Dataset

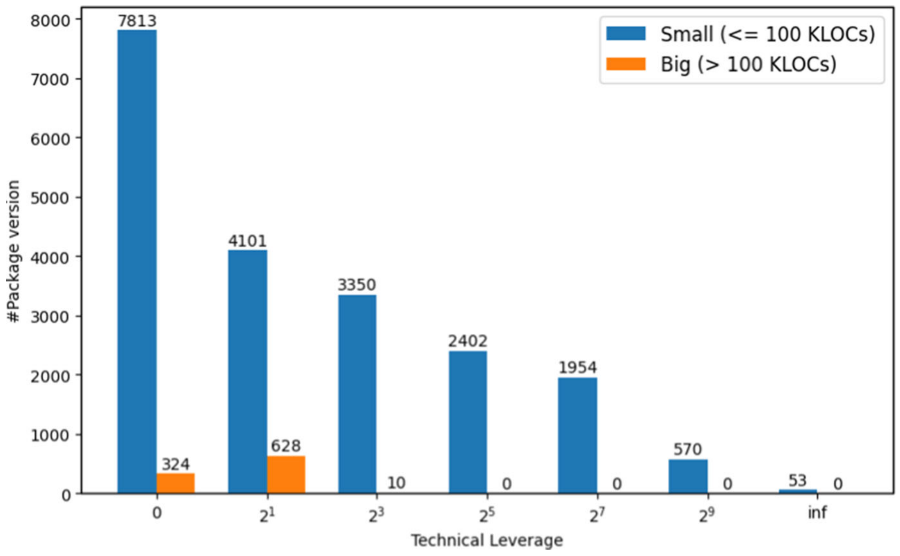
	mean	st.dev	min	Q25%	median	Q75%	max
#Pkg. version	43.99	108.85	1	11	24	44	1438
l_{own} (KLOC)	27.81	113.75	0.02	1.83	5.98	20.31	2442.05
l_{dep} (KLOC)	51.22	78.19	0	0	8.66	88.94	1337.10

This statistical data is inferred from 21,205 package versions of the top 482 Python packages (already considering transitive dependencies). Most Python packages have less than 50 versions, but the maximum can reach more than 1400 versions. Regarding dependencies, most Python packages depend on other people's code twice as big as their own

shows that 95.46% package versions in our data are small package versions, while the rest are considered big (4.54%).

Exploiting the nature of Python package manager (pip), we have already considered transitive dependencies in our calculations, which was not accounted for in the previous work in Java (Massacci et al., 2021). A third of big package versions (33.68%) has no dependencies ($\lambda = 0$), which is similar also to small package versions (38.60%).

Over half of the small package versions rely on a 6 times bigger code base as dependencies ($\lambda_{median_{small}} = 6.88$). This number drops to 0.13 ($\lambda_{median_{big}}$) in big package versions (own size l_{own} greater than 100 KLOC). In other words, the total dependency size in 50% of bigger package versions is a tenth of their own code base. This drop of technical leverage as the



Small package versions has own size (l_{own}) $\leq$ 100 KLOC, while big package versions have own size (l_{own}) $>$ 100 KLOC. We clustered the package versions into buckets of technical leverage along an exponential scale: $(2^{i-1}, 2^i)$. Both for small and big package versions, a third of them have no dependencies ($\lambda = 0$). More than half of the big package versions have technical leverage between 0 and 1, and none of them has technical leverage of more than 8.

Fig. 3 Overview of technical leverage metrics in Python ecosystem

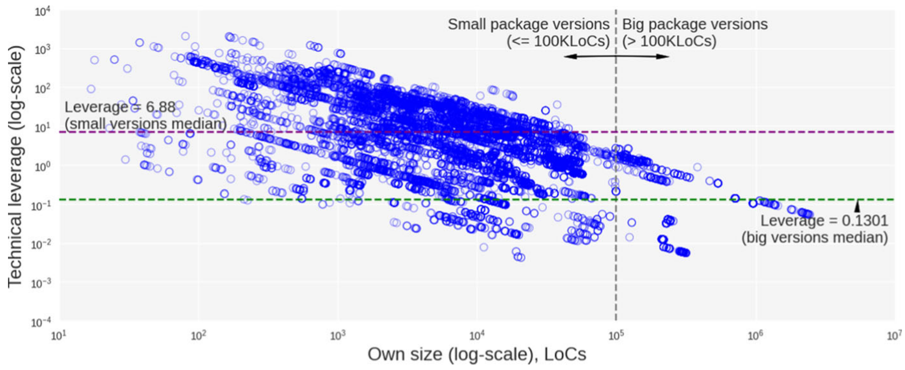
Table 3 Statistics on small vs. big package versions

Small package versions	mean	st.dev	min	Q25%	median	Q75%	max
$\lambda = 0$ (7813 versions)							
l_{own}	9.75	17.26	0.02	0.87	3.07	10.90	99.50
$\lambda > 0$ (12,430 versions)							
λ	29.94	90.23	0.004	1.22	6.88	27.62	2037.68
l_{own}	15.00	16.88	0.02	2.76	7.72	20.81	99.71
l_{dep}	82.33	85.62	0.03	17.40	59.84	128.50	133.71
Big package versions	**mean**	**st.dev**	**min**	**Q25%**	**median**	**Q75%**	**max**
$\lambda = 0$ (324 versions)							
l_{own}	158.47	39.34	100.98	131.20	149.89	183.51	248.79
$\lambda > 0$ (638 package versions)							
λ	0.50	0.62	0.006	0.03	0.13	0.94	2.64
l_{own}	432.09	491.01	100.07	162.13	228.85	335.43	2442.05
l_{dep}	98.52	76.52	1.73	7.37	99.47	156.47	332.98

This table compares small and large packages for their leverage and lines of codes, measured in KLOC - thousands of lines of code. Small package versions are package versions with $l_{own} \leq 100KLOC$, while big package versions have $l_{own} > 100KLOC$. Most small package versions (more than half) ship other people's code more than 6 times bigger than their own code. For big package versions, most package versions ship dependencies ten times smaller than their own codes

size increases is in line with the negative Pearson correlation mentioned by Massacci et al. (2021), which also can be observed in Fig. 4 in which the distributions of technical leverage have the trend from top-left to bottom-right.

To understand whether our grouping of small and big package versions makes sense, we use the two-part (Lachenbruch, 2022) test as mentioned in Section 4.1. The results of our calculation are shown in Table 4. From these results, we reject the H_0 that both small and big package versions have the same distribution. Small and big packages appear to have



For small package versions, only less than 13% of the code they ship are actually their own (correspond to the technical leverage median 6.88), while for big package versions, the proportion is higher (technical leverage median = 0.13).

Fig. 4 Technical leverage in comparison to the own size of a library

Table 4 Two-part test in comparing small vs. big package versions distributions regarding technical leverage

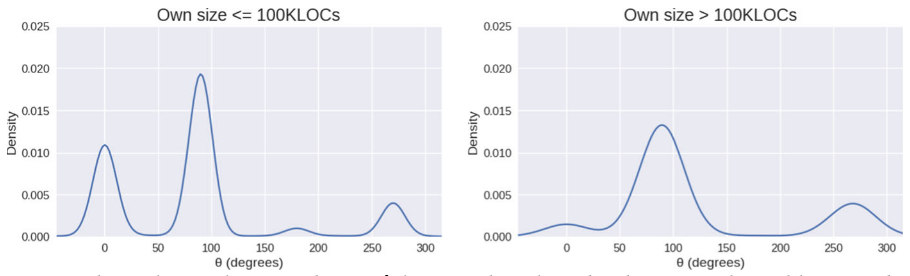
Var	Calculation	Result
p_1	$P[\lambda = 0 \mid l_{own} \leq 100,000]$	0.3860
p_2	$P[\lambda = 0 \mid l_{own} > 100,000]$	0.3368
$h_1(\lambda)$	Fit lognormal on small package versions with dependencies	$\mu = 1.6938, \sigma = 2.0179$
$h_2(\lambda)$	Fit lognormal on big package versions with dependencies	$\mu = -2.3823, \sigma = 2.6131$
B	Binomial proportion test	$z = 3.0636$ $p = 0.0022$
T	T-test of the lognormal means	$t = 60.2953$ $p \approx 0$
X^2	Result of the two-part test	$dof = 2$ $p \approx 0$ $X^2 = 3644.9101$

The two-part test was done to understand whether it makes sense to divide the ecosystem data into small and big packages or not. The result of the test shows a significant difference between the two groups (p-values < 0.05) which implies that the grouping does make sense

different tendencies. The distinction into two categories does make (statistical) sense for what concerns technical leverage.

6.2 Change direction

In addition to technical leverage, we also observe *the change direction metrics* (θ) through the Kernel Density Estimation (KDE) plot in Fig. 5. This figure shows that developers of small package versions (Fig. 5a) tend to add codes into their own code base (the peak at 90°) more than adding dependencies (the other peak at 0°). The second peak is at 0° followed by 270° which shows that developers of small packages in Python tend to change (either add or reduce) their own code. Regarding big package versions, developers of bigger Python package versions (Fig. 5b) tend to add more codes into their own code base as shown by the peak at 90°.



Large packages have a large peak at 90°showing that their developers tend to add own code. Small packages have *two* peaks: they add their own code (90°) but also add noticeably more code from dependencies (0°).

Fig. 5 KDEs of change direction for small and big package versions

Main finding for RQ1:
As in Java, Python developers also tend to ship other people's code, but with a tendency to increase their own code base.

7 RQ2: Technical leverage evolution across versions

The changes of technical leverage (λ) between two consecutive package versions are shown in Table 6. We sort the package versions using the release dates of the package versions. Table 6 shows the changes in technical leverage over the data we collected, grouped by powers-of-two intervals. The data show us that λ is very stable, as most of the time a release version stays in the same technical leverage group as its predecessor. The table shows the pattern on the diagonal from the upper-left to the bottom-right (95.48%). Regarding adding vs. reducing dependencies, the proportion of package versions that reduce dependencies appears to be more than the package versions that add dependencies, but it is still small (2.79% vs. 1.73%). These technical leverage changes are summarized in Table 5.

Fisher-exact test on this contingency table returns p-value nearly 0, which shows that the stability of technical leverage across versions is statistically significant. In other words, if one does not adopt any dependencies, they are most likely to stay so. On the other hand, if they already adopted several dependencies, they will keep leveraging on those dependencies instead of dropping them altogether.

Other than the diagonal pattern where the technical leverage does not change, most changes only happen to the preceding or succeeding interval (3.03%), while a jump of technical leverage (from really small to really big and vice versa) is rarer (1.49%). To explain these rare jumps, we take three examples (highlighted in Table 6):

1. **`kiwisolver (*)`** moves from leveraging on a huge zamount of code from dependencies (v1.1.0: $\ell_{dep} = 52.97$ KLOC, $\lambda > 256$) to not adopting dependencies at all (v1.2.0: $\lambda = 0$). In this example, the previous version (1.1.0) adopts a huge dependency: `setuptools-61.2.0` (52.97 KLOC, 2.5 MB (Foundation, 2022)) which makes $\lambda > 256$ and in the next version (1.2.0) the developers decided to remove this dependency and not replacing it with any other packages.
2. **`python-swiftclient`** (†) moves from leveraging on big dependencies codebase (v1.7.0: $\ell_{dep} = 141.85$ KLOC, $64 < \lambda \leq 128$) to reduce such dependency drastically (v2.1.0: $\ell_{dep} = 3.15$ KLOC, $1 < \lambda \leq 2$). In this example, the previous version (1.7.0) has three dependencies but in the next version, the developers decided to remove one dependency `pbr-0.11.1` which consequently also dropped its transitive dependency: `pip-22.0.4` which in total reduce the size of dependencies significantly. They also (slightly) increase the size of their own code, which reduces λ quite significantly.

Table 5 Current vs. Next package version's technical leverage

Current	Next	
	$\lambda = 0$	$\lambda > 0$
$\lambda = 0$	7759	130
$\lambda > 0$	91	12,743

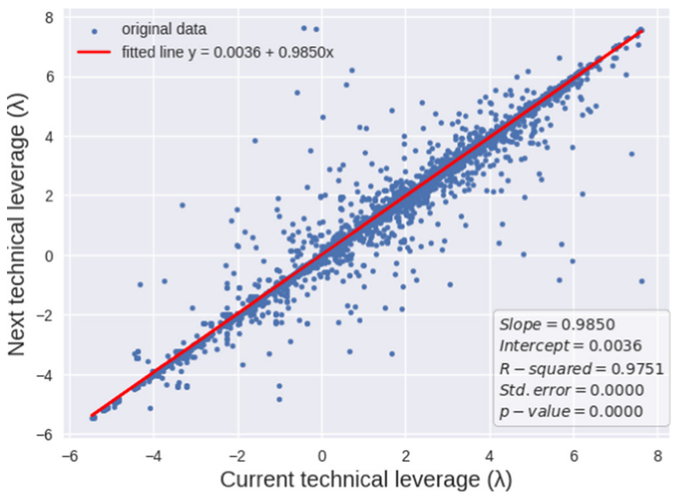
We divided our data into two categories: no dependencies ($\lambda = 0$) and have dependencies ($\lambda > 0$). Most of the package version transitions (99%) do not change the state of the package based on dependency adoption

Table 6 Technical leverage changes from one package version to the next

$\lambda_{current}$	λ_{next}										
	0	(0, 1]	(1, 2]	(2, 4]	(4, 8]	(8, 16]	(16, 32]	(32, 64]	(64, 128]	(128, 256]	(256, ∞]
0	7759	54	10	12	19	12	5	9	4	2[‡]	3
(0, 1]	35	2748	32	9	6	1		2		1	2
(1, 2]	6	49	1723	13	14	1	2		1		1
(2, 4]	10	10	30	701	23	4	2		2		1
(4, 8]	11	6	20	52	2401	26	4	2		1	
(8, 16]	11	2	5	6	62	985	14	5	2	1	
(16, 32]	4	1		2	4	52	1185	16	7		
(32, 64]	6					10	45	1333	10	4	1
(64, 128]	1		2[†]	1	1	3	6	44	432	12	1
(128, 256]	5					2		9	32	286	5
(256, ∞)	2*	3	1		1		1	1	2	22	234

We clustered the package versions based on their technical leverage with the power of 2. Most package versions (95.48%) stay in their state, as shown by the pattern from top-left to bottom-right. We also give an example for each rare jump case (cells with symbol) in the text of Section 7

3. **`lockfile`** (‡) moves from not adopting dependencies at all (v0.10.2, $\lambda = 0$) to leverage on a big amount of code from dependencies (v0.11.0: $\ell_{dep} = 138.71$ KLOC, $128 < \lambda \leq 256$). In this example, the previous version does not adopt any dependencies, but the next version adopts a new dependency called `pbr` which brings also another dependency: `pip`. Apparently both packages in total add 138,705 lines of code to the project and the increase of the own code is really small compared to this.

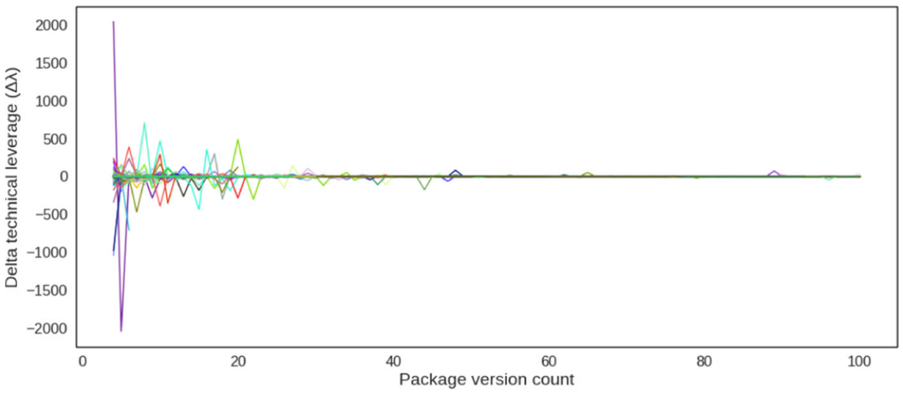


All axis are using a log scale. Current package versions for all data points in this regression have dependency/ies ($\lambda > 0$). The regression shows that the next package versions' technical leverage has strong relations with their preceding package versions'.

Fig. 6 Regression of the technical leverage between two consecutive package versions

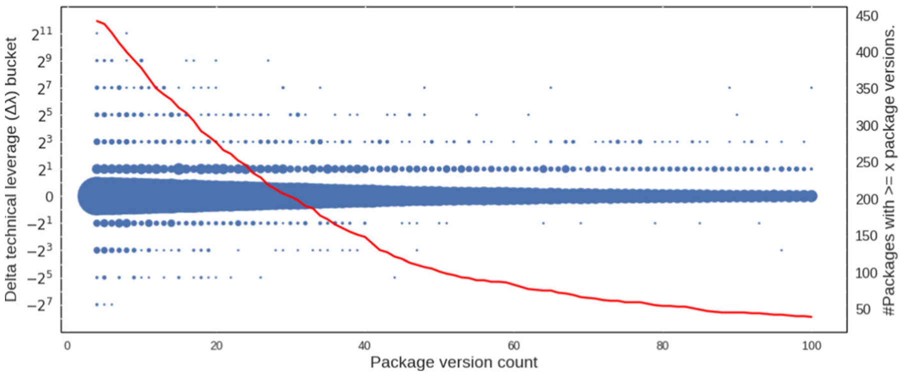
The regression of package versions with dependencies is shown in Fig. 6. The regression returns a very low p-value ($p-value \approx 0$) and high R-squared ($R^2 = 97.51\%$) which shows that technical leverage has a high tendency to be stable across versions. The tests to find out whether this relation is a Markov process result on low χ^2 values and high p-value so we cannot conclude that this relation is a Markov process.

After analyzing two consecutive package versions, we want to observe the entire evolution of technical leverages in different packages. For this reason, we use the notion of $\Delta\lambda$ (Equation 4). We show how $\Delta\lambda$ evolves through package versions (versions) for all packages in our dataset in Fig. 7a, where each line depicts a single package. In this figure, there are some peaks of $\Delta\lambda$ in the positive (> 0) side (i.e. adopting dependencies or reducing own



One line depicts the evolution of the delta lambda of an individual package. This figure is misleading as only some packages have big transitions among their first few versions (the peaks on the left part of the first graph), but they become stable as they grow (tend to have really small changes between two consecutive changes). The lower figure provides the correct illustration.

(a) Individual trajectories over time.



The size of the bubbles is proportional to the number of packages with $\Delta\lambda$ in the bucket $(2^{j-1}, 2^j)$ at their i^{th} version. The big bubbles are cluttered in the bucket with 0 delta technical leverage, which shows that most of the packages stay in their state and the peaks in the previous graph only resemble a minority of packages.

(b) Global distribution of trajectories over time.

Fig. 7 Evolution of technical leverage changes ($\Delta\lambda$) in 100 first versions across packages

code size) and the negative (< 0 side) (i.e. reducing dependencies or adding own code size). These peaks are observed to be really high/low in the first 20 versions of a package. This figure is actually *misleading* and we show precisely to illustrate our point that few outliers may give a wrong impression.

However, in accordance with the previous result, we displayed in Table 6, technical leverage mostly does not change between package versions: *most of the* $\Delta\lambda$ *are 0*. The flattening of the lines in Fig. 7a shows that this phenomenon is even more prevalent as the package becomes more mature (high number of package versions. Nevertheless, we cannot observe its prevalence for early package versions in Fig. 7a as the lines are too cluttered. Therefore, we display Fig. 7b where the size of the bubbles depict the number of packages with $\Delta\lambda$ in a specific group at their specific i^{th} version. This figure clearly shows that even for the earlier package versions, the phenomenon of stable technical leverage is also prevalent. The size of the bubbles gets smaller when the number of package versions are getting bigger because the number of packages with more than equal to that number of package versions is also getting smaller, as shown by the red line in Fig. 7b.

Main findings for RQ2:
The technical leverage tends to stay stable through the package evolution in the Python ecosystem. If you are highly leveraged, you will stay so.

8 RQ3: Technical leverage vs. vulnerabilities in Python

8.1 A finer granularity: from the view of *package versions*

Table 7 shows a view of the ecosystem when each version is considered an independent unit. The number of vulnerabilities is always higher in package versions with dependencies ($\lambda > 0$) than in package versions without dependencies. This observation holds true for both small and big package versions. We argue that this happens because technical leverage captures the interplay of risk due to indirect vulnerabilities. Consider two packages A and B that both have some (different) dependencies. Supposed package A contains 1 direct vulnerability and 9 indirect vulnerabilities, whereas package B contains 9 direct vulnerabilities and 1 indirect vulnerability. Then intuitively, we would say that package A and package B suffer differently from their vulnerable dependencies. If technical leverage would not be related to the presence of indirect vulnerabilities, we would expect cases similar to A and cases similar to B to occur with roughly the same frequency for a given value of technical leverage. However, this is not the case. As you can see from the table, where $\lambda = 0$, the bigger package versions (own code $\geq$ 100 KLOC) also have more vulnerabilities than the smaller ones because they have bigger code bases.

We also show the boxplots of the number of vulnerabilities vs. own size (for $\lambda = 0$) and technical leverage (for $\lambda > 0$) in Fig. 8 (small package versions) and Fig. 9 (big package versions). For small package versions (Fig. 8) when the number of vulnerabilities is high, both own size (for package versions without dependencies) and technical leverage (for package versions with dependencies) tend to be higher than the median. On the other hand, for big package versions, Fig. 9a shows no significant correlation between the own size and the number of vulnerabilities in package versions without dependencies. On the other hand,

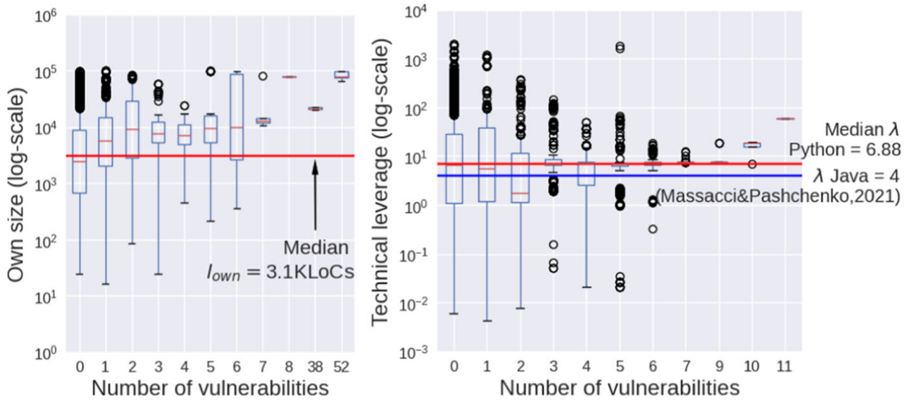
Table 7 Statistics on small vs. big package versions regarding vulnerabilities

Small package versions	mean	st.dev	min	Q25%	median	Q75%	max
#Vuln. ($\lambda = 0$)	0.47	2.42	0	0	0	0	52
#Vuln. ($\lambda > 0$)	0.82	1.71	0	0	0	1	11
of which #Direct	0.19	0.63	0	0	0	0	6
of which #Indirect	0.63	1.56	0	0	0	0	11
Big package versions	**mean**	**st.dev**	**min**	**Q25%**	**median**	**Q75%**	**max**
#Vuln. ($\lambda = 0$)	11.59	15.33	0	1	1	22	53
#Vuln. when $\lambda > 0$	6.22	9.21	0	0	0	11	37
of which #Direct	6.17	9.24	0	0	0	11	37
of which #Indirect	0.05	0.43	0	0	0	0	4

Small package versions are package versions with $l_{own} \leq 100$ KLOC, while big package versions have $l_{own} > 100$ KLOC. The proportion of vulnerable package versions is higher in the big package version group, especially big package versions with dependencies (more than half of them have at least one vulnerability, including vulnerability in the dependencies). By #Direct vulnerabilities we mean the vulnerabilities that occurred in the project's own code and by #Indirect vulnerabilities we mean the vulnerabilities that occurred in the project's dependencies

Fig. 9b shows that when the number of vulnerabilities gets really high (>27), technical leverage tends to be higher than the median.

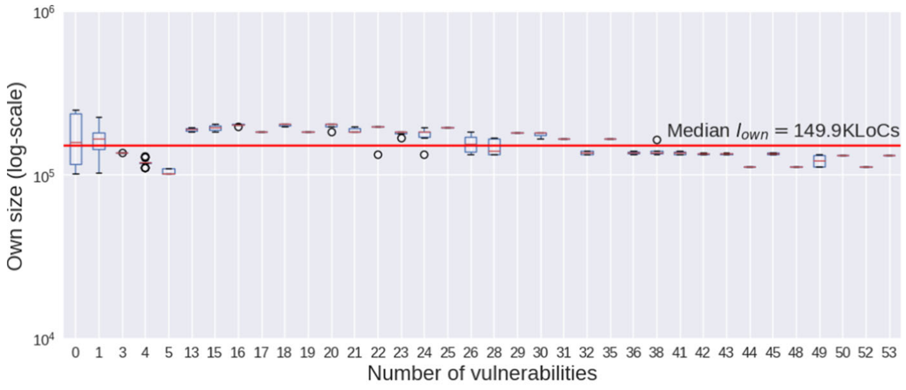
Glancing at the ecosystem, taking the number of safe package versions over the number of package versions available, the chances of picking a safe package version are shown in Table 8. This table shows that the chance of getting a safe package version seems to decrease when the technical leverage increases. However, this is a *misleading* view because it can be biased by packages with a lot of versions as we mentioned in Subsection 4.3. Therefore, in



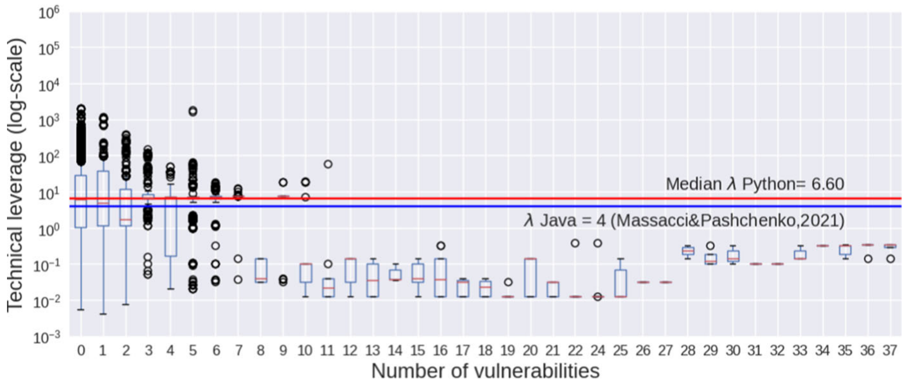
(a) l_{own} vs. v_i when $\lambda = 0$ (b) λ vs. v_i when $\lambda > 0$

For package versions without dependencies ($\lambda = 0$), when the number of vulnerabilities is considerably high (>6), the own size is higher than 10KLOC. A similar phenomenon for package versions with dependencies ($\lambda > 0$), when the number of vulnerabilities is high, technical leverages are also higher than the median $\lambda = 6.8$.

Fig. 8 Vulnerabilities in small package versions ($l_{own} \leq 100KLOC$)



(a) l_{own_i} vs. v_i when $\lambda_i = 0$



(b) λ_i vs. v_i when $\lambda_i > 0$

For package versions without dependencies ($\lambda = 0$), there is no significant correlation between own size and the number of vulnerabilities. However, for package versions with dependencies ($\lambda > 0$), when the number of vulnerabilities is really high (>27), technical leverage tends to be higher than the median $\lambda = 0.13$.

Fig. 9 Vulnerabilities in big package versions ($l_{own} > 100KLOC$)

the next subsection, we observe the ecosystem in a different way with a coarser granularity (package instead of package version) to avoid this bias.

8.2 A coarser granularity: from the view of *packages*

We clustered the packages as we defined in Subsection 4, and we run a *Chi-square contingency test* on the number of packages with/ without vulnerabilities in these three groups (third and fourth column in Table 9). The result returns $\tilde{\chi} = 12.08$ with p-value = 0.0024 and degree of freedom = 2. In the “no dependencies” group, the number of packages with vulnerability/ies

Table 8 The security of the ecosystem from the view of package versions

λ	#Versions	#Safe	$\frac{\#Safe.vers}{\#Versions}$
0	8137	6416	78.85%
(0, 4]	5539	3982	71.89%
(4, inf)	7529	5190	68.93%

The chances to get a vulnerable package version seems to be higher when technical leverage increase. However, this observation is *misleading* because it can be dominated by packages with a lot of versions

is less than average. In the second group ("below industry average"), the number of packages with vulnerability/ies is more than average. Intuitively, when a package depends on more dependency (higher technical leverage), the more likely it is to have more vulnerabilities. Unexpectedly, the third group, which contains packages with the largest technical leverage (above the industry average), has a lower number of packages with vulnerability (around the average) compared to the second group (technical leverage below the industry average).

We then applied the formulas we defined in Section 4 to get the probability values. The results are shown in Table 9. The probabilities of choosing a package with no vulnerabilities at all are high for all technical leverage buckets. It does seem higher for packages with no dependencies ($\lambda = 0$). The probability of getting a safe version reaches 89.41% for packages with no dependencies. On the other hand for packages with dependencies, the probability of getting a safe package version happens to be higher when the package is heavily leveraged.

However, the trend slightly changes when we take into account the number of downloads. In this calculation, the probability of getting a safe version for packages with no dependencies ($\lambda = 0$) is lower than for packages that are heavily leveraged ($\lambda > 4$). Comparing the calculation/simulation with and without "download" consideration, the chances of getting a safe package version when choosing a package are actually lower when developers use package popularity (downloads) in consideration.

The *Kruskal-Wallis test* on the probabilities of getting a safe package version in the 3 different groups of package's technical leverage (taking downloads into account) gives us statistic value = 10.71 and p-value = 0.0047.

We also ran our simulations as mentioned in Subsection 4.3.2 to see whether the formulas reflect the result developers will actually get. The results of our simulations are shown in Table 10. The mean of the simulations is very close to the value we got from the equations. After the simulations, we ran a *Chi-square contingency test* on the number of safe versions

Table 9 Probability of being safe

$\mathbb{E}[\lambda_{i,j}\|i]$	$N_{\Lambda=\lambda}$	$N_{V=0,\Lambda=\lambda}$	$N_{V>0,\Lambda=\lambda}$	$Pr[safe\ pkg]$	$Pr[safe\ vers]$	$Pr[safe\ vers\ downl]$
0	197	169	28	85.79%	89.41%	78.60%
(0,4]	122	85	38	69.67%	81.01%	77.05%
(4, ∞)	163	126	37	77.30%	89.91%	81.65%

We grouped the packages based on their average technical leverage. The calculation of our formula shows that the probabilities of getting a safe package version are actually higher than just reporting the percentage of vulnerable package versions: from 10 package versions, more than 8 are most likely to be entirely safe (i.e. no vulnerability at all)

Table 10 Simulation results: Probability of getting a safe package version when choosing a package

$\mathbb{E}[\lambda_{i,j}\|i]$	#Vers	$Pr[\frac{safe}{vers}]$	Simulation			d_i		$\Pr[\frac{safe\ vers}{downl}]$	Weighted Simulation		
			mean	sdev	conf.interval	mean	sdev		mean	sdev	conf.interval
0	5088	89.4%	89.4%	2.2%	(85%, 93%)	22.9	33.3	78.6%	78.6%	2.9%	(73%, 84%)
(0, 4]	7360	81.0%	81.0%	3.6%	(74%, 88%)	28.8	38.8	77.1%	77.0%	3.8%	(70%, 84%)
(4, inf]	8757	89.9%	90.0%	2.4%	(85%, 94%)	20.6	41.2	81.7%	81.7%	3.0%	(75%, 87%)

For each group of $\mathbb{E}[\lambda_{i,j}|i]$, we ran 10,000 trials with #Pkg. sample each. Downloads d_i are in millions

from normal vs. weighted simulation for the 3 groups of technical leverage. It returns $\tilde{\chi} = 12.77$ and p-value = 0.0017. This result shows that the number of downloads has a statistically significant effect on the probability of getting a safe version. The results of the formula always fall inside the 95% confidence interval of the simulation result, which shows that the formula reflects what developers will most probably get in real-life dependency adoption.

Main finding for RQ3:
Package granularity portrays a different picture of the ecosystem.

The chance of getting a safe package version when choosing a package is *higher* than the chance that would be reported when just calculating the percentage of vulnerable package versions in the ecosystem.

9 Implications of the Findings

Table 11 summarize the implications for developers and researchers matched with the main findings from each research question. We discussed them in more detail in the following subsections.

Table 11 Summary of Findings and Implications

RQ	Main Finding	Implication for Research and Practice
RQ1	As in Java, Python developers also tend to ship other people's code, but with a tendency to increase their own code base.	Understanding what else will come along when a package is adopted is key to assessing its level of uncontrollable risk.
RQ2	The technical leverage tends to stay stable through the package evolution in the Python ecosystem. If you are highly leveraged, you will stay so.	Once a package has been adopted, the level of uncontrollable risk is unlikely to change over time (this is good news or bad news if that risk was consciously or implicitly accepted)
RQ3	The chance of getting a safe library when choosing a package is *higher* than the chance that would be reported when calculating the percentage of vulnerable package versions in the ecosystem.	Making statistics based on versions is good for claiming to have done a 'large case study' but is not representative of the reality on the field. Life can be better than researchers depict it.

9.1 Implications for Practice

First, most of the top Python packages in the ecosystem are highly leveraged (RQ1, §6) This means that companies or developers choosing a library will have a high uncontrollable risk. While suggesting to not adopt anything would be unrealistic, we suggest developers to check what other libraries will come along when they adopt a certain package. In this respect, technical leverage is an easy-to-compute indicator. We believe that checking how big are these packages and whether they (currently) have vulnerabilities or not would be useful to understand the risk of adopting a certain package.

Second, once a development project has adopted a package, the percentage of uncontrollable risk is unlikely to change over time (RQ2, §7). This awareness is important during the consideration of adopting a package. If the choice was subject to a conscious deliberation in terms of risk and opportunities this is actually good news as the accepted risk is unlikely to change over time. If the risk was accepted tacitly, this is bad news.

Third, even when adopting a package in Python seems to dramatically increase one's own project, our empirical result shows that the chance of getting a safe version when adopting a package is still considerably high (RQ3, §8). Therefore, life can be better than depicted by researchers. If developers have to adopt 10 package versions from different packages with no dependency, 9 of them would most likely be safe. However such difference is not so big when dependencies are considered so weighting opportunities and risk is a rational decision.

An interesting option would be to apply *dependency vendoring* (copy-pasted dependencies code to own code). Dependency vendoring is done *in purpose* to gain more control and internalize the risk (Rosen et al., 2022). This makes sense because if one did dependency vendoring, then the copy-pasted code becomes under the control of him/herself. In fact, people do this to improve code quality and get reproducible builds (Fitzpatrick, 2015). However, dependency vendoring is not cheap and our result in RQ3 (§8) shows that the probability of having a safe library when choosing a package is higher than 80%, and therefore dependency vendoring might not be the best option for small companies.

9.2 Implications for Research

At first, one result shows that more care should be given when applying default statistical tests in the analysis of ecosystems. For example, the usage of Lachenbruch's statistical test (Lachenbruch, 2022) in mining software repository studies is extremely rare, while it is a natural test to compare two classes in datasets with excess zeroes such as most software repositories.

Second, our result in RQ3 (§8) shows that the usage of the correct granularity (e.g. package vs version) for the analysis is important to present results that better reflect what developers experience in real-life development. Any developer always chooses first a package that offers the desired functionality (Dashevskyi et al., 2016; Pashchenko et al., 2020) and then a version of that package. Sampling directly on versions thus creates a process for statistical analysis that is not representative of reality and may offer misleading results.

10 Related Works

Dependencies and vulnerabilities. Dependencies are abundant in software ecosystems. (Decan et al., 2019). Soto-Valero et al. (2021a) found that 89.2% of dependencies that are loaded but not actually used will keep being loaded and still not used. This is aligned with our result in §7, that once you are (heavily) leveraged, you will stay so (Decan et al., 2019; Hejderup et al., 2022). Leveraging on dependencies can bring risks when the dependencies are not maintained properly: it has been found that up to 81% of projects use outdated dependencies (Kula et al., 2018; Hejderup, 2015; Pashchenko et al., 2022). The risk of these outdated dependencies increases even more when they are the central packages that are being used by a lot of packages in the ecosystem (Kikas et al., 2017).

However, Kula et al. (2018) mentioned that 69% of developers they interviewed are unaware of the fact that they are using vulnerable dependencies. This finding is aligned with the results found by Cox et al. (2015) and Lauinger et al. (2018). Cox et al. (2015) applied their metric to 1642 Maven projects and found that security issues are more likely to be detected in software with low dependency freshness. In the Javascript (npm) ecosystem, Lauinger et al. (2018) utilized causality tree analysis and reported that a third of the websites they analyzed are using a significant portion of vulnerable dependency inclusions.

Adopting dependencies. Thus, how could developers know the risk that dependencies bring to their projects? Manadhata et al. (2010) proposed an *attack surface metric* to systematically measure the attack surface of software. They formally defined *attack surface* as the triplets of system, channel, and data attackability. The "technical leverage" metric proposed by Massacci et al. (2021) focuses on the system part of these triplets. This metric measures how big the part of software is covered by dependency compared to the original code by the developer, bringing the concept of leverage in finance to the technical software development world. Using this metric, they found that in the Java ecosystem, a library with a dependency size 4x their own code size increases the risk of having security vulnerability by 60%.

However, after using this metric, developers still need to decide which dependencies to adopt, which versions to use, or even whether they should update to a version or not. Bonaccorsi et al. (2003) introduced a model to assist developers in choosing dependencies to adopt. This model gives scores to dependencies based on the support developers can get from the FOSS community if they use these dependencies. However, this model does not consider the security risk in the decision-making process.

Security-related empirical analysis on software ecosystem. The way of reporting vulnerable or safe packages/ package versions varies from paper to paper. Decan et al. (2018) reports the absolute number of vulnerable packages in their dataset of npm packages, taking into account the evolution throughout time. Similarly, in the Python ecosystem, Alfadel et al. (2021) and Bagmar et al. (2021) also report the absolute number of vulnerable artifacts. The difference is that Alfadel et al. (2021) considered a more fine-grained view from package versions instead of packages.

Also using percentages, Wang et al. (2020) and Prana et al. (2021) report the percentage of safe or/ and vulnerable versions in Java and other ecosystems. For the Python ecosystem, Ruohonen et al. (2021) conducted a large-scale static analysis on over 197 thousand Python packages and reported the percentage of packages affected with at least one security issue (46%). By using a similar concept, Zimmermann et al. (2019) defined the "vulnerability reporting rate" which is the ratio between the number of vulnerable packages over the total number of packages at a specific point in time in the npm ecosystem.

The way of reporting the absolute number of vulnerable package/ package versions and the percentage happen to be insufficient to reflect what developers will actually face when they are going to adopt a package version. Therefore, this work addresses the gap by looking at the ecosystem from the developers' point-of-view and finds out how is the actual chance of getting a safe or vulnerable package version as we illustrated in Table 10 (Section §8).

11 Threats to Validity

Package selection. From the top 600 Python packages, our dataset eventually included only 482 packages. This limitation happens because in the process of computing technical leverage, we use `pip install` command (to download the package version and its dependencies) and for some packages, the command executions failed due to a variety of reasons (i.e. ModuleNotFoundError, AttributeError, etc.). These errors should be investigated in future work to improve the dataset. Still, the surviving packages provide us 21,205 versions which are already 2.5 times more than the previous work in Java. Further, packages that do not easily installare less likely to be used in practice.

Direct vs. transitive dependencies. Given the nature of `pip install`, we already considered transitive dependencies in our computation. This makes it difficult to compare our result with the previous work in Java which only considered direct dependencies. However, we believe that the results would be relevant for the general readers as transitive dependency inclusion was also described as future work in the previous Java study.

Dependency resolution. As mentioned, we are using `pip install` to get dependencies and calculate the technical leverage metric. In the case when the requirement of a package allows several versions of the dependencies, we are using the default setting of `pip`, which will get us the latest stable version (unless it is explicitly told to use an unstable version). Hence, when the analysis is being rerun in the future, the result might be different (as the next stable version will be selected). However, based on our results on RQ2: "A package's technical leverage tends to stay stable across versions", we believe that the change will not be significant.

Syntactic sugar. Codes in Python contain a lot of syntactic sugars and we acknowledge that this might impact the calculation of lines of code. However, this would not make a difference *within* the same ecosystem as every member will likely use the same syntactic sugar.

Vulnerabilities in non-python files. While we only count lines of codes from `.py` files, we acknowledge the possibility of a vulnerability occurring in files written in other languages that come with a package. However, we consider this threat negligible.

First-party considered as third-party. Third-party dependencies might actually be first-party dependencies in the sense of Massacci et al. (2021). This is a problem that is inherent to Python.

At first, there is no hierarchical structure and therefore one would need to look at different features to determine whether two packages belong to the same team of developers for which it can make sense to claim that the dependency is actually a first-party dependency. The most obvious ones are looking at the maintainers, the source code repository, the developers, the times of deployment, and even looking at the package name.

For our running example, `boto` and `botocore` are both maintained by AWS, but this level of granularity is too coarse (AWS maintains 18 very different packages in PyPI (Foundation, 2023a)). Another PyPI maintainer, Microsoft, maintains 510 different packages (Foundation, 2023c). It is hard to assume that all 510 packages of Microsoft should be considered as coming from the same project.

We also cannot use the software repository (e.g. Github) as a proxy because some repositories are huge and have too many developers to be really as a single project (e.g. AWS has 386 repositories in Github (GitHub, 2023a))

Another possible way is grouping packages with the same release dates, but we also cannot use this as a proxy as this is true for all projects with huge maintainers. For example, `botocore` and `awscli` have the same release dates in PyPI but we investigated their Github repositories (GitHub, 2023b, c) and found that the respective developers are significantly different. They just always release at the same time as it is the policy of the organization to which the developers belong.

One cannot even use the name of the package as a proxy as `aiobotocore` is maintained by Jettify/ Nikolas Novik (Foundation, 2023b) and this is a different group from `botocore`.

From the perspective of the results in this paper, this difficulty does not impact the notion of transition (if leveraged, always leveraged) as a library wrongly attributed to be 3rd party will uniformly move from numerator to denominator for all packages using the 'root' library: you will become less leveraged and will stay less leveraged.

12 Conclusions and Future Works

Technical leverage is the ratio between the lines of code of third-party libraries and own code line used in a software package. It has been used to analyze the relationship between the risk and opportunity in the Java Maven ecosystem (Massacci et al., 2021) and the npm ecosystem (Anonymous, 2022). In this work, we performed an empirical analysis that utilizes the same metric on the Python ecosystem.

We first investigated the ecosystem as a whole to find the distribution of technical leverage among packages. We gathered the data of a total of 21,205 package versions from 482 (out of 600) top Python packages. Smaller Python package versions (less than 100 KLOC) happen to have higher technical leverage than bigger package versions (6.9x vs 1x).

Our analysis of the evolution of technical leverage across versions in Python packages shows that Python packages tend *not to* change through versions. If a package is already highly leveraged, it will most probably stay so. Only at the very beginning of a package history there are some oscillations. An interesting hypothesis to study for future work is whether developer changes could be explained by Stokey's economic theory of inaction in the presence of fixed costs (Stokey et al., 2008). In computer science, this theory has been shown to fit the development of exploits (Allodi et al., 2022).

The final investigation was related to security. To accurately reflect developers' actual probability of getting a vulnerable package we identified formulae to compute the probability of getting a safe package when choosing a package (and then a safe version for that package). We then complemented the formulas with simulations: at first when all package versions have equal chance to be chosen, and then by using downloads as a metric of popularity-based choices. The result from our formulas and simulations show that the chance of picking a safe version vs. a vulnerable version is higher (8-to-2) than the traditionally reported ratio of safe package versions based on the ratio between safe version and vulnerable versions (7-to-3). This is most likely due to some outlier packages with several persistently vulnerable versions that bias coarser calculation based on ensemble means.

For future work, we plan to improve our dataset with more package versions to have a broader view of the ecosystem. Regarding our metrics on the probability of getting a safe package version, a possible interesting future work is to observe different ways of approximating how developers choose a package other than download count, i.e. SourceRank (GitHub, 2023c). We also plan to analyze the security risk in updating one's libraries with more precise granularity. Does it always yield to a not vulnerable state? Most analyses provide information in retrospect but this information does not correspond to what the developer knew at the time the new version was made available.

Another interesting future work we are planning is to compare different ecosystems in the same analysis (e.g. Java vs. Python). In this kind of comparison study, standard libraries can potentially impact the comparison and it would be interesting to analyze them. When analyzing different ecosystems, it can also be interesting to observe if low-level metrics such as code entities and function points do affect technical leverage.

Acknowledgements We would like to thank Ivan Pashchenko for useful discussions on technical leverage and Batbayar Narantsogt for providing the first initial code base for extracting libraries for Python. We also thank the anonymous reviewers whose comments greatly helped to improve the paper. Any remaining mistakes were ours. This work was partly funded by the EU under the H2020 Program AssureMOSS (Grant n. 952647).

Author Contributions *Conceptualization:* RP, FM; *Methodology:* RP, FM; *Software:* RP; *Validation:* RP, FM; *Formal analysis:* RP, FM; *Investigation:* RP; *Data Curation:* RP; *Writing - Original Draft:* RP, FM; *Writing - Review & Editing:* RP, FM; *Visualization:* RP; *Supervision:* FM; *Project administration:* FM; *Funding acquisition:* FM.

Funding This work was partly funded by the EU under the H2020 Program AssureMOSS (Grant n. 952647).

Data Availability We provide an online demo for computing technical leverage of Python package versions for interested readers at the following URL: https://techleverage.eu/livedemo/pypi. The full dataset generated during and/or analyzed during the current study can be accessed through this Zenodo link: https://doi.org/10.5281/zenodo.7186627.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

Employment Ranindya Paramitha is employed by the University of Trento, Trento, Italy. Fabio Massacci is employed by the University of Trento, Trento, Italy and Vrije Universiteit Amsterdam, Amsterdam, The Netherlands.

Financial/ Non-financial Interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

References

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

Allman E (2012) Managing technical debt. Commun ACM 55(5):50–55

Allodi L, Massacci F, Williams J (2022) The work-averse cyberattacker model: theory and evidence from two million attack signatures. Risk Anal 42(8):1623–1642

Anonymous (2022) Opportunities and Security Risks of Technical Leverage: A Replication Study on the NPM Ecosystem. https://doi.org/10.5281/zenodo.6585292

Bagmar A, Wedgwood J, Levin D, Purtilo J (2021) I know what you imported last summer: A study of security threats in the python ecosystem. arXiv:2102.06301

Banker R, Liang Y, Ramasubbu N (2021) Technical debt and firm performance. Manage Sci 67(5):3174–3194

Bickenbach F, Bode E, et al. (2001) Markov or not markov-this should be a question. Tech. rep., Kiel working paper

Bonaccorsi A, Rossi C (2003) Why open source software can succeed. Res Policy 32(7):1243–1258

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

Cunningham W (1992) The wycash portfolio management system. ACM SIGPLAN OOPS Messenger 4(2):29–30

Dashevskyi S, Brucker AD, Massacci F (2016) On the security cost of using a free and open source component in a proprietary product. In: International Symposium on Engineering Secure Software and Systems, Springer, pp 190–206

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

Decan A, Mens T, Grosjean P (2019) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir Softw Eng 24:381–416

Derr E, Bugiel S, Fahl S, Acar Y, Backes M (2017) Keep me updated: An empirical study of third-party library updatability on android. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp 2187–2200

Fitzpatrick B (2015) Dependencies & vendoring. https://groups.google.com/g/golang-dev/c/nMWoEAG55v8/m/pjZVLIzKX9EJ?pli=1

Foundation PS (2022) Pypi · the python package index. https://pypi.org/, Accessed 07 March 2022

Foundation PS (2023a) Aws. https://pypi.org/user/aws/

Foundation PS (2023b) Jettify. https://pypi.org/user/jettify/

Foundation PS (2023c) Microsoft. https://pypi.org/user/microsoft/

Foundation PS (2023d) python-loc-counter. https://pypi.org/project/python-loc-counter/

Frakes WB, Kang K (2005) Software reuse research: Status and future. IEEE Trans Softw Eng 31(7):529–536

GitHub I (2023a) Amazon web services. https://github.com/aws

GitHub I (2023b) boto/botocore. https://github.com/boto/botocore

GitHub I (2023c) nice-registry/sourceranks. https://github.com/nice-registry/sourceranks

Gkortzis A, Feitosa D, Spinellis D (2021) Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. J Syst Softw 172:110653

Hejderup J (2015) In dependencies we trust: How vulnerable are dependencies in software modules? PhD thesis, Computer Science, TU Delft

Hejderup J, Beller M, Triantafyllou K, Gousios G (2022) Präzi: from package-based to call-based dependency networks. Empir Softw Eng 27(5):102

Huijgens H, Van Deursen A, Minku LL, Lokan C (2017) Effort and cost in software engineering: A comparison of two industrial data sets. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, pp 51–60

Jorgensen M, Shepperd M (2006) A systematic review of software development cost estimation studies. IEEE Trans Softw Eng 33(1):33–53

van Kemenade H, Si R (2022) hugovk/top-pypi-packages: Release 2022.03. https://doi.org/10.5281/zenodo.6319631

Khoirom S, Sonia M, Laikhuram B, Laishram J, Singh TD (2020) Comparative analysis of python and java for beginners. Int Res J Eng Technol 7(8):4384–4407

Kikas R, Gousios G, Dumas M, Pfahl D (2017) Structure and evolution of package dependency networks. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, pp 102–112

Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empir Softw Eng 23(1):384–417

Lachenbruch PA (2002) Analysis of data with excess zeros. Stat Methods Med Res 11(4):297–302

Lauinger T, Chaabane A, Arshad S, Robertson W, Wilson C, Kirda E (2018) Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. arXiv preprint arXiv:1811.00918

Limited S (2023) boto3 vulnerabilities. https://security.snyk.io/package/pip/boto3/versions?page=8

Manadhata PK, Wing JM (2010) An attack surface metric. IEEE Trans Softw Eng 37(3):371–386

Massacci F, Pashchenko I (2021) Technical leverage in a software ecosystem: Development opportunities and security risks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, pp 1386–1397

Massacci F, Sabetta A, Mirkovic J, Murray T, Okhravi H, Mannan M, Rocha A, Bodden E, Geer DE (2022) “free” as in freedom to protest? IEEE Secur Priv 20(5):16–21

NIST (2023) Cve-2021-33503 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-33503

Pashchenko I, Plate H, Ponta SE, Sabetta A, Massacci F (2018) Vulnerable open source dependencies: Counting those that matter. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Association for Computing Machinery (ACM), ESEM ’18, https://doi.org/10.1145/3239235.3268920

Pashchenko I, Vu DL, Massacci F (2020) A qualitative study of dependency management and its security implications. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp 1513–1531

Pashchenko I, Plate H, Ponta SE, Sabetta A, Massacci F (2022) Vuln4real: A methodology for counting actually vulnerable dependencies. IEEE Transactions on Software Engineering 48(5):1592–1609. https://doi.org/10.1109/TSE.2020.3025443

Pernet C (2023) Machine-learning python package compromised in supply chain attack. https://www.techrepublic.com/article/pytorch-ml-compromised/

Pittenger M (2016) Open source security analysis: The state of open source security in commercial applications. Black Duck Software, Tech Rep

Prana GAA, Sharma A, Shar LK, Foo D, Santosa AE, Sharma A, Lo D (2021) Out of sight, out of mind? how vulnerable dependencies affect open-source projects. Empir Softw Eng 26(4):1–34

Rosen S (2022) Can vendoring dependencies in a build be officially supported? https://discuss.python.org/t/can-vendoring-dependencies-in-a-build-be-officially-supported/13622

Ruohonen J, Hjerppe K, Rindell K (2021) A large-scale security-oriented static analysis of python packages in pypi. 2021 18th International Conference on Privacy. Security and Trust (PST), IEEE, pp 1–10

Slaughter SA, Harter DE, Krishnan MS (1998) Evaluating the cost of software quality. Commun ACM 41(8):67–73

SnykDB (2022) Snyk vulnerability DB. https://snyk.io/vuln, Accessed 03 Feb 2022

Soto-Valero C, Durieux T, Baudry B (2021a) A longitudinal analysis of bloated java dependencies. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 1021–1031

Soto-Valero C, Harrand N, Monperrus M, Baudry B (2021) A comprehensive study of bloated dependencies in the maven ecosystem. Empir Softw Eng 26(3):1–44

Stokey NL (2008) The Economics of Inaction: Stochastic Control models with fixed costs. Princeton University Press

Vu DL, Pashchenko I, Massacci F, Plate H, Sabetta A (2020) Typosquatting and combosquatting attacks on the python ecosystem. In: 2020 ieee european symposium on security and privacy workshops (euros&pw), IEEE, pp 509–514

Wang Y, Chen B, Huang K, Shi B, Xu C, Peng X, Wu Y, Liu Y (2020) An empirical study of usages, updates and risks of third-party libraries in java projects. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 35–45

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

Ranindya Paramitha is a Ph.D. student at the University of Trento, Trento, 38123, Italy. She received her master's degree in informatics with distinction from Institut Teknologi Bandung, Bandung, 40132, Indonesia. Her main research interest is in software security, focusing on empirical analysis of secure software ecosystems, mining software repositories, and how developers can apply security. She is involved in an H2020 European Project: AssureMOSS and has also started to actively serve the research community in several IEEE/ACM International Conferences/Workshops, such as by being a student volunteer (ICSE'22) and program committee (ICSE SVM'23).

Fabio Massacci is a professor at the University of Trento, Trento, 38123, Italy, and Vrije Universiteit, Amsterdam, 1081 HV, The Netherlands. Massacci received a Ph.D. in computing from the University of Rome "La Sapienza." He received the IEEE Requirements Engineering Conference Ten Year Most Influential Paper Award on security in sociotechnical systems. He participates in the FIRST special interest group on the Common Vulnerability Scoring System and the European pilot CyberSec4Europe on the governance of cybersecurity. He coordinates the European A ssureMOSS project. He is a Member of IEEE, the Association for Computing Machinery, and the Society for Risk Analysis.