Fix Your Requirements(.txt)! A Study of Vulnerable Python Packages in Open-Source Software

Mashal Abbas
smabbasz@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Shahpar Khan
shahpar.khan@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Abdul Monum
amonum@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

The heterogeneity of modern software requires the use of packages that provides useful functionality that otherwise would require custom implementations. Most software is not written with security as a first-class citizen and the code for these packages is no different. In such cases, the vulnerabilities that stem from these packages are inherited by projects that use them. While security patches are frequently released for vulnerable versions, the integration of newer versions in projects is not a standard practice leading to insecure software. In this work, we investigate the prevalence of vulnerable packages in open-source Python software, and their evolution in response to discovered vulnerabilities. On a set of curated Python projects, we find that 53.5% of projects are vulnerable due to insecure package dependencies. We further investigate and find out that around one-third of the projects adopt to safer versions of package dependencies.

1 INTRODUCTION

Open-source software often consists of many different components. The vulnerabilities that exist in each of the components propagate to the software itself. One such component is the packages that are used for functionality in the software. Insecure or vulnerable packages can lead to information leakage or undesired functionalities in the software [13]. Integrating newer versions of packages may lead to changes in code implementation, dependency changes, or the introduction of new bugs. These factors may delay the integration of the newer version of the packages which leads to insecure software.

Python is one of the most popular languages that is used in open-source software. A unique feature of Python is that it has its own eco-system of open-source packages known as the Python Package Index (PyPI) [21]. The open-source nature of the packages leads to more opportunities for vulnerabilities to creep in as software evolves. According to Alfadel et al. there exist at least 252 insecure packages in PyPI as of 2021 [12].

This project aims to investigate the prevalence of insecure Python packages in open-source software. Leveraging WoC [17] and Safety [18], we build a pipeline that can test for vulnerable packages within open-source software. We evaluate our methodology on a suite of a highly curated python projects published on GitHub [23]. We carry out an extensive quantitative analysis to find out the presence of vulnerable packages. We see how software adapts in light of discovered vulnerabilities and the window of vulnerability that persists. This allows us to assess the security of open-source projects and the practices that are being followed by the community when it comes to security fixes of packages.

In particular, we address the following research questions:

RQ1: How prevalent are security vulnerabilities in popular and active open-source Python projects?

We investigate whether popular and active open-source Python projects still use vulnerable Python packages and find that around 53.8% of the projects are vulnerable through insecure packages at the time of this paper. Furthermore, we highlight the most frequently used vulnerable packages and their corresponding CVEs.

RQ2: How often do projects adopt the use of secure versions of Python packages?

We explore the adoption of Python projects to the use of secure packages and find that around one-third of the projects keep their dependencies up to date. In addition to that, we find that some projects do not update their dependencies at all.

RQ3: What factors affect the adoption of secure packages?

We examine whether factors such as the number of contributors, project size, and the number of commits have a significant effect on adopting the secure update of vulnerable Python packages. We do not find any significant correlation between these factors. We believe that a qualitative study would help us answer this question.

In this paper, we will first discuss the background needed to understand the project in § 2 followed by the motivation in § 3. Details of how we carried out our research can be found in § 4. We then discuss our results in § 5.

2 BACKGROUND

In this project, we leverage World of Code (WoC) which is a database that contains the source code of projects from GitHub, BitBucket, Debian, PyPI and GitLab [17]. This corpus is created so that it can be used to carry out research in the Software Engineering realm. It contains information about the commits to the project, the authors, and the blobs therefore, it can encapsulate the evolution of the project. For this project, we use it to extract the dependencies that are needed for a project. These are found in the requirements.txt file.

To detect the vulnerabilities in projects we use Safety which uses SafetyDB [19]. SafetyDB contains the known security vulnerabilities in Python packages and is updated once a month by syncing with PyUp. The list of these known vulnerabilities is known as Common Vulnerabilities and Exposures (CVEs) maintained by the MITRE Corporation. Safety scans for insecure dependencies that are present within the project and the corresponding CVEs that affect the package and its version. It also gives recommendations on what should be done to make the project safer.

3 MOTIVATION

Open-source software tends to grow large in size with many contributors and hence it is imperative that maintaining it will become a challenge [16]. In addition, it has many external components that need to be kept up-to-date in terms of security and functionality. One of the most commonly used such components are packages. They are widely used in Python projects and due to the open-source nature of these packages they also evolve with time.

Many modern tools can be used to check the state of projects and find out vulnerabilities present in the software. One of the tools that can check the security of packages used in a project is Safety [19]. However, projects with prevailing vulnerable packages still exist.

In the preliminary setup of our project, we wanted to see how prevalent are vulnerable packages and whether our research will yield something substantial. Therefore, we scoured GitHub and randomly tested popular Python projects to see whether there exists a project that is vulnerable and can be flagged using Safety CLI. It did not take us long to come across Flask-WTF — an integration between Flask and WTForms [24]. The popularity of Flask-WTF can be judged from the following statistics on its GitHub repository: 1.3K stars and 301 forks. Upon checking the health of the Flask-WTF repository using Safety CLI client we found out that Flask-WTF is using a vulnerable version of a Python package called py. The vulnerable version corresponds to CVE-2022-42969, which reports the py library allowing remote attackers to conduct a ReDoS (Regular expression Denial of Service) attacks through version 1.11.0 [11].

This demonstrates how imperative it is to answer the research questions we have listed in § 1. Furthermore, it also highlights that despite the presence of these tools, projects still do not use them. It points towards other reasons why open-source software are still vulnerable.

4 METHODOLOGY

This section entails a description of how we carried out our quantitative study. We take a deep dive into the rationale and methodology behind our dataset compilation and data processing that forms our subsequent analysis.

In this project, we aim to see how plagued open-source Python projects are and how they affect the community. Therefore, it is important to investigate the Python projects that are popular and widely used. For this study, we compile our dataset out of GitHub repositories, however, our selection of projects depends on two main requirements:

(1) The repositories need to have high popularity metrics — such as stars and watchers.
(2) The repositories must contain a requirements.txt file specifying all the of the Python dependencies.

To address our first requirement, we use of an open-source curated list of Python projects published on GitHub. This repository is called awesome-python [23]. This list contains approximately 500 projects that are actively monitored by the admin panel. The projects are across different domains such as Hardware and ML. There are over 1.6K commits, 162K stars, and 22.8K forks on this repository making it a highly credible source of popular Python projects. We manually check multiple projects in this repository and confirm that these projects have high popularity metrics. After confirming that and establishing Awesome Python as our main source of projects, we iterate over their list and extract the author and repository names and saved them in a text file.

With the project names list at our disposal, we turn to World of Code (WoC). WoC provides a massive collection of basemaps that enables the users to query mappings and extract the data per their needs. As for our second requirement, we want to investigate GitHub repositories that are primarily Python projects containing a requirements.txt. With WoC, we can achieve this by using the project-to-commit maps where the key is the project name and the value is a list of commit that modified requirements.txt file. For our interaction with the data stored on WoC servers, we use Oscar [22] — a convenience Python library to access World of Code data. With the help of Oscar APIs, we reliably automated our tasks for greater efficiency.

After filtering the projects based on them having a requirements.txt file, we get a final set of 183 projects. We store the result of the project-to-commit maps and move on to the next phase of our data processing. At this stage, we extract the data from requirements.txt blob and save it in a text file. We do this for each commit of each project that modified requirements.txt. The name of each file is a concatenation of three fields we extract from WoC: author, Unix timestamp, and timezone. The final directory structure looked like this.

```
projects
 ├─project_a
 │  ├─author_time_timezone.txt
 │  └─...
 ├─project_b
 │  ├─author_time_timezone.txt
 │  └─...
 └─...
```

Now we are set to collect our vulnerability reports on each of these requirements.txt files. For this task, we use *Safety* — an open-source command-line tool for scanning Python environments for dependency security and compliance risks [18]. Safety utilizes the SafetyDB which is a maintained list of vulnerable Python modules with respect to certain Common Vulnerabilities and Exposures (CVEs) [19]. We analyze all requirements.txt using Safety and create a CSV file out of its result. This CSV file has the following features:

(1) The name of the project
(2) The author of the project
(3) Time
(4) Timezone
(5) A list of names of packages scanned
(6) Total number of packages scanned
(7) Total number of vulnerabilities detected
(8) Vulnerability ID
(9) The name of the vulnerable package
(10) The vulnerable version of the package
(11) The analyzed version of the package
(12) CVE

Package Name	Number of Projects
sphinx	38
wheel	23
requests	23
jinja2	22
py	20

Table 1: Top 5 most vulnerable packages and the number of projects affected by them

CVE	Number of Projects Affected
CVE-2020-11022 [7]	38
CVE-2020-11023 [8]	38
CVE-2018-18074 [6]	35
CVE-2022-40898 [10]	23
CVE-2020-28493 [9]	22

Table 2: Top 5 CVEs prevalent and the number of projects affected by them

Each row represented a vulnerability in a single commit of a requirements.txt file. We have made this data available on our GitHub repository [20].

With the rich data at our disposal, we run our exploratory data analysis using a Python notebook that answers the research question in § 1. After doing extensive analysis, we share our findings and results in § 5.

5 RESULTS

In this section, we present and discuss the results of our empirical study of vulnerable packages in the Python projects curated by awesome-python [23].

RQ1: How prevalent are security vulnerabilities in popular open-source Python projects?

Security vulnerabilities from packages may propagate to projects that use them. In this RQ, we investigate how many projects have and still use vulnerable Python packages, which packages are the most vulnerable, and the number of vulnerabilities that are present in the wild.

By analyzing all the modified versions of the requirements.txt file for a project and running Safety CLI for each version, we can determine whether the project was vulnerable at any given point in time. We find that 75.6% (140) projects were vulnerable at some point throughout their existence and 53.5% (98) are still vulnerable according to the latest commit that modified the requirements.txt file as reported by WoC [17]. For the projects that are still vulnerable, we find that there exists a total of 571 unique CVEs and 613 unique vulnerabilities across 62 packages. Every other project being vulnerable is an alarming situation and shows that security vulnerabilities through insecure packages are prevalent. Moreover, the sheer number of vulnerabilities associated with a relatively small set of packages shows that security is not considered a priority when writing these packages and thus these packages should be used with high caution. The top five most vulnerable packages are shown in Table 1 and the most common CVEs are shown in Table 2.

Given a project is vulnerable, we were also interested to know how many packages out of the total packages required to build the project are vulnerable. We find that the median number of such vulnerable packages in each project are 6.66% and the number increases to 30% when considering the upper quartile. These observations are expected since vulnerabilities often lie at the extreme end of the distribution.

Change in Vulnerabilities	Number of Projects
Decreased	45
Increased	39
Same	44
Removed all	30

Table 3: Evolution of projects in terms of change in vulnerabilities

RQ2: How often do projects adopt the use of secure versions of Python packages?

In an ideal world, project maintainers should ensure that their project is up-to-date not only with respect to system functionality but also in terms of security. With the advent of well document CVEs and safety tools such as Safety CLI [18], this task should be non-trivial. In this RQ, we examine how projects evolve in maintaining the use of secure versions of packages to build their project. This can translate in terms of adopting the newer version of the package with the security update, using a different secure package that provides similar functionality, or writing your own custom implementation.

Since we have all the modified versions of requirements.txt, we can examine the first and latest version of the file to determine how the projects have evolved in terms of using secure packages. The results are summarized in Table 3.

We find that roughly one-third of the projects put an effort in terms of adopting secure versions while two-thirds either maintain or increase the number of vulnerabilities throughout. The increase can either be attributed to newer CVEs discovered for the same version of the insecure package or the addition of new insecure packages. Of the 45 projects that decreased the number of vulnerabilities through insecure packages, 30 of them removed all of the vulnerabilities which reflects good security maintenance practice.

To visualize the evolution of a single project in terms of security vulnerabilities, we analyzed the project tyiannak/pyAudioAnalysis [15] that illustrates all of the above three cases in Fig. 1. We plot the time of the commit of each version of requirements.txt in its existence and the number of CVEs corresponding to each modified version. We can observe that CVEs from the year 2021 have remained

Project	Number of Negligent Commits
ironmussa/Optimus	1284
Kotti/Kotti	263
Miserlou/Zappa	154
nficano/python-lambda	118
flask-admin/flask-admin	95

Table 4: Top 5 projects with the most number of negligent commits

throughout the existence of the project while CVE-2019-10775 and CVE-2020-7598 were introduced just before the end of 2021. On the other hand, the top four CVEs in the graph were present in the years before 2018 but removed after 2020.

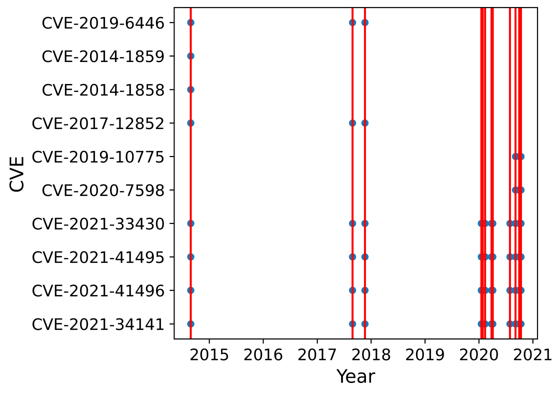

Figure 1: Time when requirements.txt was modified and the number of CVEs associated to each modified version

We can also note from Fig. 1 that there exist CVEs associated with packages that were discovered in later years after their use. While we emphasize that the projects were still vulnerable during that time period, the lack of discovery during that time is not to blame on the project maintainers. As such, we proceed to analyze the vulnerabilities associated with commits that modified requirements.txt after the CVE was discovered. In particular, it means those commits where they updated the requirements.txt (add or remove packages) but the packages with known vulnerabilities still exist in the list. We extracted the year in which the CVE was published from the CVE name and compared it with the year of the commit which modified requirements.txt. We find that approximately one-third (52) of the projects were guilty of such negligence at some point in their existence and there were a total of 2535 such commits. The top five projects with the most number of negligent commits are shown in Table 4. To probe the current state of these projects, we analyze the latest version of requirements.txt. We find that the same number of projects still contain at least one CVE that came earlier than when the packages list was updated. These results highlight that maintainers of projects do not prioritize the use of secure packages even when they are updating the list of packages for that project.

Project	CVE Count
ironmussa/Optimus	59
TheAlgorithms/Python	10
scanny/python-pptx	8
flask-admin/flask-admin	7
rochacbruno/quokka	7

Table 5: Top 5 projects with most number of CVEs that should not have been present

CVE	Negligent Years
CVE-2013-7489 [2]	8
CVE-2016-9243 [3]	5
CVE-2017-3735 [4]	4
CVE-2013-2132 [1]	4
CVE-2017-3736 [5]	4

Table 6: Top 5 CVEs with most number of negligent years

In Table 5 we summarize the top five projects that contain CVEs that should have not been present if a secure list of packages were used. The number of total such CVEs across all projects goes up to 107. Since we only compare the CVE year and the commit year, we suspect the actual number to be much higher. Even this restrictive number suggests that older CVEs are still prevalent in open-source projects.

We were also curious to know the distribution of time when the CVE was discovered compared to the year package list was last updated. We find that a project contained a CVE which was discovered a median of two years before the use of package. The list of top 5 such CVEs with the most number of such negligent time is summarized in Table 6.

RQ3: What factors affect the adoption of secure packages?

There can be a lot of factors at play which can affect the adoption of secure packages. In this RQ, we intend to investigate this by finding any characteristics in the nature of projects that can help distinguish whether this project maintains a list of secure packages or not. As such, we analyze the number of commits that modified requirements.txt and the number of authors associated with updating the requirements.txt. We compare projects that decreased the number of vulnerabilities or had no vulnerabilities throughout (maintaining a secure packages list) with the projects that either increased vulnerabilities or had the same number (not maintaining a secure packages list). As you can observe in Fig. 2 and Fig. 3, the distribution of the number of authors and the number of commits that modified requirements.txt for both sets of projects is roughly the same. The median number of authors for projects that adopt the use of secure packages and vice versa is 5 and 4 respectively and the median number of such commits does not have a significant difference (17 and 10) given the high amount of standard deviation. Since all sampled projects are popular, and have a high number of stars and watchers associated with them, popularity is also not a distinguishable metric to judge. Hence, we cannot find any distinguishable characteristics between the nature of the two sets of projects such that we can determine whether a project is likely to adopt secure packages. Therefore, if any such factors exist that affect the adoption of secure packages, we are unable to find them in this empirical study and thus we do not have a concrete answer to this research question.

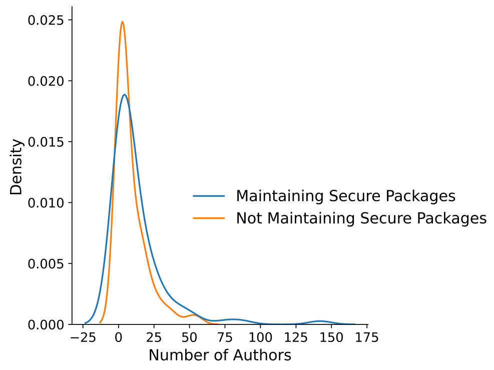

Figure 2: Author Distribution of projects that maintain secure packages vs those that do not

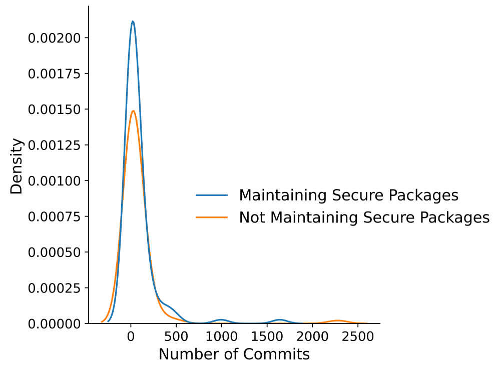

Figure 3: Commit Distribution of projects that maintain secure packages vs those that do not

6 DISCUSSION

Through our study, we learn that there are projects that do not update their dependencies when vulnerabilities are found within them. Since these vulnerabilities are publicly known there is the risk of attacks if an older version of the package is being used. These risks are even greater if the software is using or has access to private data. There are tools that developers can use to check the state of their projects regularly. These tools are easy to use and can be automated so that once a vulnerability is found it can alarm one of the developers.

We find all the CVEs that affect a particular version of the package. These CVEs might not be relevant to the scope in which the package is being used but as the project develops more functionality might be added that uses that package and hence it is better if all the packages are up to date. Furthermore, some vulnerabilities might not have a fix, and hence developers using these packages must be cautious about whether this vulnerability will affect their functionality or not.

We believe that it is the responsibility of the developers to keep track of the dependencies that are used in the project and that updates should be integrated as soon as possible. In conclusion, we urge you to fix your requirements.txt!

7 RELATED WORKS

There have been various studies previously that quantitatively study and research the repercussions of using vulnerable software dependencies. In this section, we will talk about two past studies on the implication of using vulnerable software packages.

The open-source nature of the packages leads to more opportunities for vulnerabilities to creep in as software evolves. According to Alfadel et al., there exist shortcomings in discovering vulnerabilities in Python packages [12]. The authors come to this conclusion by conducting an empirical study that discovers over 500 vulnerabilities in 252 Python packages. The authors also emphasize that there is a window between the release of the vulnerability patch and its public disclosure leaving an interval for malicious entities to act.

Similarly, the evolution of security vulnerabilities through package dependencies can lead to catastrophic consequences. Vulnerability in one package can easily plague all the other packages linked to it. The evolution of security vulnerabilities in Node Package Manager (npm) dependency networks is quantitatively studied by Decan et al. [14]. The authors studied almost 400 security reports that affected over 72K unique package releases. The research team empirically establishes that it takes a long time to discover vulnerabilities and 15% of these vulnerabilities are either fixed after the public announcement of the vulnerability, or not fixed at all.

Our work builds on the insights of the aforementioned works and aims to find how many vulnerabilities through insecure packages are prevalent and propagated through open-source software that uses them. We further investigate the rate of adoption of secure versions and the causes of the delay.

8 THREATS TO VALIDITY

We extracted almost 500 project names from the list curated by the authors of Awesome Python. However, the total number of projects we were able to evaluate to establish our results was 183. There is a three-part explanation for the reduced number of projects. First, our core requirement for analysis depended on using Safety to inspect vulnerabilities. Safety requires a requirements.txt file to query it against its database, therefore, we dropped all the projects that did not maintain a requirements.txt file. Second, we noticed that there can be a few discrepancies between the commit history on WoC and the commit history on the actual GitHub. Since we depended on commits to the projects available on WoC, we dropped all the projects if we did not get a single commit against requirements.txt in those projects. Lastly, in a requirements.txt file you can specify dependency with or without specifying the specific version. In case you do not specify the version, at the time of installation, Python's

Preferred Installer Program (PIP) installs the latest version. Safety requires packages specified with a version so that it can query its database to see if it is vulnerable with respect to a CVE or not. Therefore, projects that had requirements.txt with all packages specified without a version had to be dropped as well. We believe that the packages with no versions attached to them in the requirements.txt will be downloaded as the latest version and hence can be considered safe. Therefore, they might not affect the scope of our study.

Inspecting packages against a wide range of CVEs manually would be extremely tedious. To solve this, we looked for software solutions that are free to use and can be automated. Safety was the only software that met our working demands completely. We rely on Safety to detect vulnerable Python packages and thus any vulnerable packages not listed in the Safety DB will not be detected. However, Safety is a well-maintained and up-to-date project and thus we do not expect any significant false negatives. However, Safety is synced only once per month, hence any zero-day vulnerabilities might have been missed.

We have only analyzed a subset of Python projects available in the WoC corpus due to time and computation constraints. We extracted projects which we believe cover multiple domains by using the Awesome Python project. However, our insights may not generalize as they may change with changing the population. An improvement will be to run this on all the projects available in WoC which would require a large-scale study.

We carried out a quantitative analysis with the data that we had at our disposal. This data might not be complete to answer our RQ3, there may be several external factors that affect how a project updates its dependencies. In future work, we can couple it up with a qualitative study to understand what might be the external factors.

9 CONCLUSION

With this study, we aimed to see how open-source software responds to evolving dependencies. In this project, we carry out a small-scale study to investigate the prevalence of vulnerabilities through insecure packages in Python projects. We use WoC [17] to get our corpus of projects and use Safety [18] to detect the vulnerabilities of the dependencies in open-source software. We then carry out a quantitative analysis of our data and find out that around half of these projects are currently vulnerable. Furthermore, only one-third of the projects adopt to secure versions of the packages while the rest are negligent. In the future, this project can be expanded to find out the reasons why dependencies are not kept up-to-date and the insights can be made more generalized on a large corpus of projects.

REFERENCES

[1] 2013. CVE-2013-2132. https://nvd.nist.gov/vuln/detail/CVE-2013-2132
[2] 2013. CVE-2013-7489. https://nvd.nist.gov/vuln/detail/CVE-2013-7489
[3] 2016. CVE-2016-9243. https://nvd.nist.gov/vuln/detail/CVE-2016-9243
[4] 2017. CVE-2017-3735. https://nvd.nist.gov/vuln/detail/CVE-2017-3735
[5] 2017. CVE-2017-3736. https://nvd.nist.gov/vuln/detail/CVE-2017-3736
[6] 2018. CVE-2018-18074. https://nvd.nist.gov/vuln/detail/CVE-2018-18074
[7] 2020. CVE-2020-11022. https://nvd.nist.gov/vuln/detail/CVE-2020-11022
[8] 2020. CVE-2020-11023. https://nvd.nist.gov/vuln/detail/CVE-2020-11023
[9] 2020. CVE-2020-28493. https://nvd.nist.gov/vuln/detail/CVE-2020-28493
[10] 2022. CVE-2022-40898. https://nvd.nist.gov/vuln/detail/CVE-2022-40898
[11] 2022. CVE-2022-42969. https://nvd.nist.gov/vuln/detail/CVE-2022-42969
[12] Mahmoud Alfadel, Diego Costa, and Emad Shihab. 2021. Empirical Analysis of Security Vulnerabilities in Python Packages. https://doi.org/10.1109/SANER50967.2021.00048
[13] Kemal Altinkemer, Jackie Rees, and Sanjay Sridhar Krannert. 2005. Vulnerabilities and Patches of Open Source Software : An Empirical Study.
[14] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 181–191. https://doi.org/10.1145/3196398.3196401
[15] Theodoros Giannakopoulos. 2016. Python Audio Analysis Library: Feature extraction, classification, segmentation and applications. https://github.com/tyiannak/pyAudioAnalysis
[16] Feras Hanandeh, Ahmad Saifan, Mohammed Akour, Noor Alhussein, and Khadijah Shatnawi. 2017. Evaluating Maintainability of Open Source Software: A Case Study. *International Journal of Open Source Software and Processes* 8 (01 2017), 1–20. https://doi.org/10.4018/IJOSSP.2017010101
[17] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. 2020. World of Code: Enabling a Research Workflow for Mining and Analyzing the Universe of Open Source VCS data. arXiv:2010.16196 [cs.SE]
[18] Pyupio. 2017. Safety Cli - Security for your python dependencies. https://pyup.io/safety/
[19] Pyupio. 2017. SafetyDB: A curated database of insecure Python packages. https://github.com/pyupio/safety-db
[20] Project Repository. 2023. Vulnerable Python Packages Study. https://github.com/abdulmonum/vulnerable-python-packages-study
[21] Jukka Ruohonen, Kalle Hjerppe, and Kalle Rindell. 2021. A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI. In *2021 18th International Conference on Privacy, Security and Trust (PST)*. IEEE. https://doi.org/10.1109/pst52912.2021.9647791
[22] Ssc-Oscar. 2018. oscar.py: Python interface for Oscar Data. https://github.com/ssc-oscar/oscar.py
[23] Vinta. 2015. Awesome-python: A curated list of awesome python frameworks, libraries, software and resources. https://github.com/vinta/awesome-python/
[24] WTForms. 2010. Flask-WTF. https://flask-wtf.readthedocs.io/en/1.0.x/