Characterizing and Detecting Python Version Incompatibilities Caused by Inconsistent Version Specifications

Shuo Li[a,b], Haocheng Gao[a,b], Wei Chen[d,e,*], Yi Li[c,*], Haoxiang Tian[a,b], Chengwei Liu[c], Dan Ye[a]

[a]*Institute of software, University of Chinese Academy of Sciences, Beijing, China*
[b]*University of Chinese Academy of Sciences, Beijing, China*
[c]*Nanyang Technological University, Singapore*
[d]*Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China*
[e]*Nanjing Institute of Software Technology, Nanjing, China*

Abstract

Python interpreter serves as the foundation for developing Python projects. Configuring compatible Python version is essential for packaging and reusing third-party libraries (TPLs) in the Python ecosystem. When the Python distribution version range constraint of a third-party library is incorrectly declared, it may lead to installation failures or runtime errors when the library is used by a client project. We call such errors Python Distribution Version Specification Error (PDSpecErr). PDSpecErrs are not studied adequately and there still lacks a method to guarantee the correctness of Python version specifications. To combat PDSpecErrs and mitigate their adverse impacts on downstream applications, we conducted the first exploratory study on 3,000 real-world TPLs to investigate PDSpecErrs about their prevalence, symptom categories, and diagnostic patterns. Based on the comprehensive study, we designed and implemented **PyChecker**, a tool to automatically

*Corresponding author
Email addresses: lishuo19@otcaix.iscas.ac.cn (Shuo Li), gaohaocheng21@otcaix.iscas.ac.cn (Haocheng Gao), chenwei@otcaix.iscas.ac.cn (Wei Chen), yi_li@ntu.edu.sg (Yi Li), tianhaoxiang20@otcaix.iscas.ac.cn (Haoxiang Tian), chengwei.liu@ntu.edu.sg (Chengwei Liu), yedan@otcaix.iscas.ac.cn (Dan Ye)

detect PDSpecErrs in TPLs, locate root causes, and generate fixing solutions. We evaluated PyChecker on 3,000 TPL releases and PyChecker detected 842 PDSpecErrs, reporting their root causes and recommending fixing solutions. We have reported 52 PDSpecErrs to the concerned developers. So far, 32 issues have been confirmed and fixed according to our suggestions.

Keywords: Python library, Python version incompatibility, Exploratory study

---

1. Introduction

The Python community is flourishing with millions of **T**hird-**P**arty **L**ibraries (TPL) indexed in the online central repository PyPI [1]. Contributors publish their projects to the repository, from which downstream users can download and reuse the TPLs. With the online repository and client-side tools, such as `pip` [2], the installation of TPLs can be fully automated. Notably, each Python third-party library (TPL) includes a set of configuration entries typically found in configuration files such as setup.py, pyproject.toml, or requirements.txt. These entries contain essential metadata about the project, including its description, dependencies, and environmental restrictions. For instance, in the setup.py file, a configuration entry *python_requires* might look like this: "`python_requires='>=3.6, <4.0'`". These metadata help downstream users understand how to use the TPL and facilitate many client-side tasks and tools.

TPL developers may set the configuration entries incorrectly, which can potentially result in version incompatibility between the configuration options. This, in turn, can lead to errors either during installation or run-time for downstream users. Taking Figure 1 as an example, the TPL `goatools 1.2.3` [3] forgets to declare that the compatible Python versions should be 3.0 or above (by setting "`python_requires`" entry). This incorrectly implies that the library is without any Python version restriction. As a result, an "installation failure" occurs when a downstream user tries to install the library with Python 2.7. This is due to the TPL using a Python 2-incompatible syntax *Exception Chaining* (PEP 3134 [4]) in `setup_helper.py`. The version incompatibility is caused by the inconsistent version specifications between "`python_requires`" and configuration options in `setup_helper.py`. Of course, users may avoid this problem by changing the local Python environment or downloading other alternative libraries. However, the same

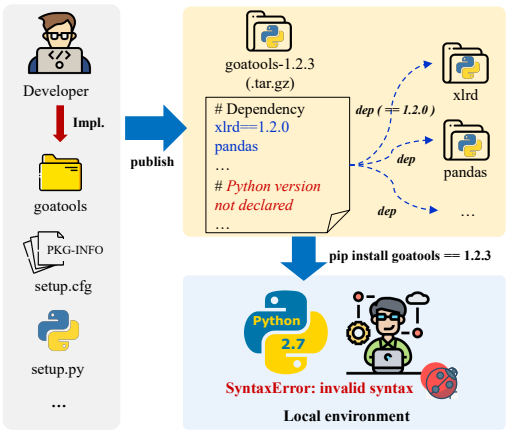


Figure 1: A configuration error example.

specification error may still cause other incompatibilities when the problematic library is used by other client projects. We call such incorrectly declared Python version ranges in a third-party library **P**ython **D**istribution Version **Spec**ification **Err**ors (PDSpecErrs), which often result in installation failures or runtime errors of client projects. We found that even widely used frameworks like TensorFlow with stars over 184K contains PDSpecErr. The PDSpecErr persists over two years and still confuse downstream users [5]. Unfortunately, it is difficult for downstream users to prevent and detect PDSpecErrs due to the complexity of the configurations and the target TPL source code.

However, few studies have focused on the problem. Existing works mainly focus on dependency conflicts [6, 7, 8] and dependency errors [9, 10, 11, 12]. These works address dependency-related issues from the perspective of downstream users based on the assumption that TPL configuration entries are correct. In contrast, we find the configuration entries in Python TPLs are not always reliable. Combined with an overreliance on human practices and a lack of collective awareness, these factors together result in the oversight of PDSpecErrs and the absence of a reliable approach to ensure the correctness of Python version specifications. To fill this gap, our work revolves around identifying TPLs' Python version incompatibilities caused by inconsistent version specifications in configuration entries.

To combat PDSpecErrs and mitigate their adverse impacts on down-

stream applications, we begin by tracking and summarizing the underlying automated procedure from publishing to installing a TPL. This helps us understand why Python version constraints in TPL configurations are error-prone. We then conduct an exploratory study on PDSpecErrs among the top 3,000 real-world TPLs in the Python community to investigate its prevalence, characters, manifestation patterns, and diagnostic patterns. Our exploration finds that, 17.8% (534 out of 3,000) of the collected TPLs experience installation failures or run-time errors, among which 84.46% (299/534) are relevant to PDSpecErrs. We retrieve from *Libraries.io* that these failed cases have the potential to cause disruptions on a vast scale on downstream projects with over 45K libraries and 182K repositories. Some characteristics of TPLs, e.g., the total number of version releases and average update time interval, exhibit significant differences between TPLs with and without PDSpecErrs. These characters can help provide useful insights for fostering a more reliable development ecosystem. Furthermore, these PDSpecErrs can occur in TPL installation and run-time with several distinct error manifestation patterns. Motivated by the manifestations, we analyze the common diagnostic patterns of PDSpecErrs relating to the "`python_requires`" configuration entry. Specifically, we identify three diagnostic patterns of PDSpecErrs, i.e., *setup script issue*, *incompatible feature*, and *Python version conflict*.

Based on our findings, we propose PyChecker, a tool to automatically detect PDSpecErrs. PyChecker thoroughly inspects a TPL's configuration file and source code to detect PDSpecErrs under three diagnostic patterns. We evaluate PyChecker on the top 3,000 TPLs and find 842 TPLs with PDSpecErrs. During our investigation, we find that a single TPL may be associated with multiple diagnostic patterns simultaneously, and PyChecker is able to detect them at the same time. Additionally, PyChecker also reveals obscured PDSpecErrs that hide behind other errors, which can only be detected under specific conditions. With limited time and manpower, we have reported 52 detected PDSpecErrs to the relevant developers. So far, 32 PDSpecErrs (61.54%) have been confirmed and fixed following our suggestions. Many developers conveyed their gratitude to us for reporting errors in their project configurations.

In summary, this work makes the following contributions.

- **Originality:** To the best of our knowledge, we conduct the first exploration study of TPL PDSpecErrs within the Python community. Our study can help better understand the characteristics of PDSpecErrs

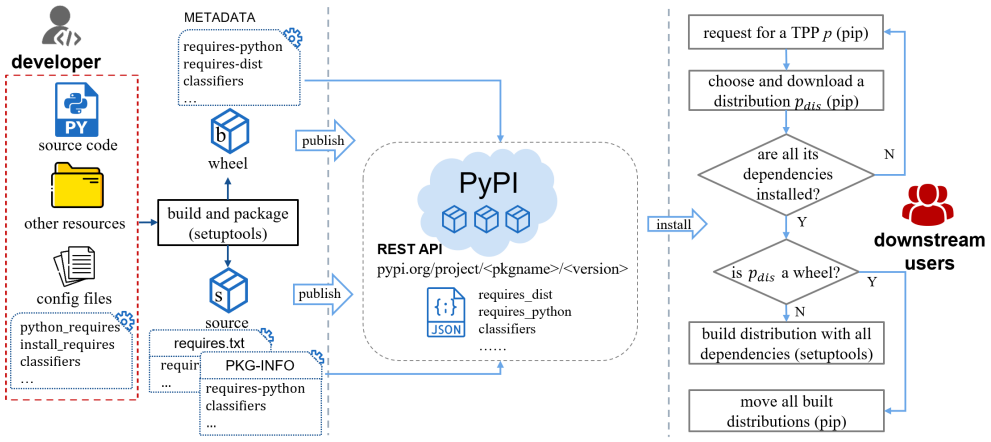


Figure 2: The procedure from packaging to installing a TPL.

and offers valuable insights for future studies in this domain.

- **Technique:** We categorize three common symptoms of PDSpecErrs and identify corresponding diagnostic patterns. Based on our findings, we designed PyChecker to detect PDSpecErrs automatically. Our evaluation reveals that PyChecker can effectively identify PDSpecErrs, including obscured ones. Additionally, developers' responses validate the usefulness of PyChecker in aiding developers in detecting PDSpecErrs.

- **Dataset:** PyChecker and the experiment dataset are publicly available at http://github.com/PyVCEchecker/PyChecker, which can facilitate future research.

2. Problem Analysis

This section first summarizes the underlying procedure of packaging, publishing, and installing a TPL and highlights the usage and transformation of typical configuration entries during the procedure. Then, we analyze why PDSpecErrs occur and which tasks in the procedure are error-prone.

2.1. Procedure Summarization

Figure 2 summarizes the procedure from TPL development to installation, including three stages: *packaging*, *publishing*, and *installing*.

Stage 1: Packaging a TPL. To package a project into a TPL, a developer should create a configuration file in addition to the source code and other necessary resources (e.g., data, tests, and documents). A configuration file usually contains *descriptive* and *directive* entries. Descriptive entries provide project meta-information, such as the release name and version. Directive entries specify the required dependency resources and their versions and dictate the build tool, usually `setuptools` [13], to automate the project's build and packaging task. Note that three kinds of configuration files, i.e.,`setup.py`, `setup.cfg`, `pyproject.toml`, are commonly used, and the entries of the same semantics within them can be named differently, such as "`python_requires`" in `setup.cfg` and `setup.py` [14] and "`requires-python`" in `pyproject.toml` [15]. These configurations files are typically located at the root of the TPL directory and may be present together or as standalone files within the project.

TPL packaging can generate two types of releases, including binary releases (also known as Wheel binary or built) and source code releases. The build tool creates metadata files by extracting information from the configuration file. Specifically, besides the compiled extensions, a wheel removes the original configuration file and generates a metadata file, `METADATA`. In contrast, a source release retains the original configuration file and creates some metadata files, e.g., `PKG-INFO` and `requirements.txt`, each containing a part of the original information and configuration options. Notably, some entries in the generated metadata files are renamed. In particular, the two restrictive entries are renamed to "`requires-python`" and "`requires-dist`", respectively.

Stage 2: Publishing a TPL. For an uploaded TPL, PyPI extracts information from metadata file(s) and presents it through the web page and REST APIs. This allows downstream users to quickly grasp essential details such as release version, type, and description. Users can also draw preliminary conclusions by examining more detailed entries like semantic tags ("`classifiers`"), required Python versions ("`requires-python`"), and dependencies ("`requires-dist`"). For instance, the entry "`requires-python`" helps users assess compatibility with their local environment.

Stage 3: Installing a TPL. Installing a TPL requires to recursively download and build the target TPL and its dependencies. Overall, this stage contains several steps. (1) `pip`, the most common client-side tool collaborating with

PyPI [16], requests a target TPL p from PyPI and receives an available release list of p. (2) `pip` selects and downloads the latest compatible release p_{dis} from the list, prioritizing wheel releases for faster installation. If a specific TPL version is provided, `pip` will download the corresponding version instead. For example, if the user wants to install version 1.2.3 of the `requests` library, they can specify the version in the pip command as "`pip install requests==1.2.3`". This command instructs pip to download and install version 1.2.3 of the requests library, bypassing any newer versions. (3) `pip` then gets all required dependencies and their restricted versions from the entry "`requires-dist`" in p_{dis}'s metadata file `METADATA` and checks whether all dependencies are installed. For source releases, `pip` generates `METADATA` using `setup.py`. (4) Specific to source releases, `pip` builds p_{dis} with a build tool after ensuring all dependencies are installed. (5) `pip` copies built files to a specific directory, such as `site-package`, and creates an entry point.

2.2. Misconfigured Version Specifications

Overall, the TPL configuration is error-prone due to its heavy reliance on the manual work of developers and the lack of effective inspection mechanisms.

For developers, *diverse entries* in TPL configuration are often renamed and reused across files and stages, which may lead to developers' misunderstanding, neglect, or arbitrary settings. Additionally, setting entries requires a *comprehensive project understanding*, yet developers may lack awareness, resulting in overlooked dependencies or relaxed restrictions.

For downstream users, installing a TPL is also prone to encountering misconfigured version specifications. The entry "`requires-dist`" may miss some dependencies. Further, incorrect "`requires-python`" may lead downstream users to configure their environments with inappropriate Python interpreters, causing compatibility problems.

Notably, to gain a comprehensive study and help Python developers and users combat these errors, we focus on errors related to the configuration entries "`python_requires`" and "`install_requires`" for their widespread usage and closely associated with PDSpecErrs. Specifically, we explored various errors and, based on development practices and the guidance provided by accepted Python Enhancement Proposals (PEPs) [17, 18], we found that most other errors were related to user-defined entries, which are less common. Other entries, such as "programming languages", may be missing in TPLs. In comparison, the errors caused by the two standard entries were

much more prevalent. Additionally, the analysis in procedure summarization also highlights the importance of these two entries and their attributions to PDSpecErrs. Therefore, we decided to focus on these two most common entries. Other configuration errors are left for future work.

3. Exploratory Study

We conducted an exploratory study to characterize the prevalence of PDSpecErrs, explore the potential influences of the specific inherent characteristics of TPLs on their occurrences, and identify error manifestation patterns. Subsequently, we identified and summarized three diagnostic patterns for detecting PDSpecErrs.

3.1. Data Collection

To the best of our knowledge, there does not exist any public dataset of TPL releases with PDSpecErrs. Additionally, as of December 2023, approximately 499,871 libraries, encompassing over 5,158,546 releases, have been published on the PyPI ecosystem [1]. Analyzing all libraries and their releases becomes a prohibitive task. Therefore, following previous studies [10, 19], we focused on the top 3,000 TPLs ranked in PyPI according to the *SourceRank* provided by *Libraries.io* [20]. Focusing on the top 3,000 libraries is due to that top libraries are widely adopted and significantly impact the ecosystem, making them highly representative of broader trends and practices. The threshold of 3,000 is derived from existing research; for instance, Huang et al., [21] investigated 3,000 real-world applications, while Watchman [6] examined the top 1,000 popular Python projects on PyPI. Thus, selecting 3,000 libraries allows for a comprehensive analysis that aligns with established studies.

To identify and collect TPLs that exhibit PDSpecErrs, we simulated the behavior of downstream users by installing and running each TPL with its latest version in multiple simulated local environments. During this process, we checked the compatibility of each TPL with its declared compatible Python versions, automated by scripts. For any error that occurred during the installation or run-time, we collected it and manually performed a thorough analysis of the error messages to verify whether it was caused by a PDSpecErr. The detailed steps of our data collection are as follows.

Step 1: Constructing local environments for TPLs. For each collected TPL, we created multiple container-based local environments with different Python versions according to its declared compatible Python versions in metadata. Notably, for the declared Python distributions, we selected version 2.7 and versions 3.5 to 3.10, as they are mature and widely used by Python developers and the community. Each local environment was constructed with the official Docker image containing the corresponding Python interpreter and the Debian 10 operating system.

For example, `docker-compose 1.29.2` [22] specifies the Python version requirement ≥ 3.4. Thus, its local environments are constructed by six containers, and each container is configured with a different Python version among 3.5 to 3.10. The TPLs that do not declare their Python version constraints are supposed to be compatible with all Python versions, and thus, their local environments were installed with all six Python versions.

Step 2: Installing and importing TPLs. We ran the command "`pip install <lib-name>==<version>`" to install a release *lib* in each constructed Docker container. *lib* possibly contains a PDSpecErr when the installation fails in one or more local environments. For example, `docker-compose 1.29.2` is successfully installed in all its local environments except the one with Python 3.5. To mitigate the influences brought by network disruptions, we retried each failed installation three times, and considered it a failure only if none of the retries succeeded. The stimulation process run for 7 days and 16 hours running on Ubuntu 18.04 LTS with 8-core 3.50 GHz CPU and 32 GB RAM.

Considering that PDSpecErrs may also occur at run-time besides installation, for the TPLs that can be successfully installed in all local environments, we further extracted their top-level modules from the metadata file `top-level.txt` and ran the command *`python -c ''importing <module>''`* to simulate importing a `module` in *lib* at the run time. Note that due to the diverse and complex nature of run-time errors, they may arise under specific triggering conditions. Therefore, for the sake of time and labor costs, we focused solely on errors related to importing modules in this context.

Step 3: Identifying TPLs with PDSpecErrs. After Step 2, among the tested libraries, 371 and 163 TPLs experienced installation failures and run-time errors, respectively. Notably, run-time errors are not limited to errors that occurred during importing the top-level modules of TPLs. As we have mentioned above in Step 2, due to our time and labor constraints, only top-level

import errors(as we mentioned before, it refers to the errors caused by importing a module at the highest level within the TPL hierarchy, such as "`import scipy`", instead of "`import scipy.linalg`") were considered in the context of run-time errors in the automated detection process.

According to the error messages and the stages where the identified errors occur, three co-authors manually checked if the errors are PDSpecErrs. Three authors all have at least eight years of programming experience in Python. It took about two weeks to label and analyze the 534 failed TPLs.

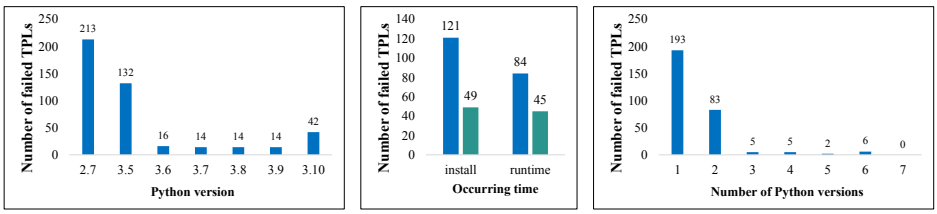
Specifically, we maintained a shared table with six columns, including library, version, error message, failed Python versions, error stage (install or import), and PDSpecErr (True or False). At each round, each author was randomly assigned 100 failed TPLs with the corresponding error messages. After about five rounds of crossed-validations, all three co-authors independently checked all the error messages of each failed TPL. The Cohen's kappa rate reaches 99.19%. Each inconsistency was discussed among the authors until they reached a consensus.

Finally, we obtained 299 TPLs containing PDSpecErrs, out of which 170 experienced installation failures and 129 experienced run-time errors. There were 135 removed releases, which failed for several other reasons, such as TPLs being removed or unavailable, C language-related errors, and so on. Note that pip, as the Python community's de facto TPL manager, executes commands sequentially and only reports the first encountered failure, and thus, the other following errors would not be exhibited. Therefore, we can only categorize a TPL error into one type of PDSpecErrs, which may miss cases in which a TPL has multiple PDSpecErrs.

3.2. Observation 1 (Prevalence)

For investigating the prevalence of PDSpecErrs, we conducted the statistics of TPLs containing PDSpecErrs in terms of: (1) the number of errors associated with each Python version, (2) the number of errors occurring in each stage, and (3) the number of Python versions with which each TPL fails. The results are shown in Figure 3.

Figure 3 (a) shows that most TPLs failed in Python 2.7, 3.5, and 3.10. This is understandable since Python 2.7 and 3.5 have reached their end-of-life (EOL), and many newly published TPLs tend to stop supporting them. Other failures mostly happen on TPLs that release updates less frequently, failing to keep up with the pace of Python version upgrades.



(a) The number of failed TPLs with each Python version

(b) Statistics of failed TPLs in terms of package type and occurring time

(c) The statistics of TPLs in terms of how many Python versions they simultaneously failed with

Figure 3: Statistics of TPLs containing PDSpecErrs.

Figure 3 (b) shows that the majority of PDSpecErrs occurred during the installation stage. We also count the library types (binary and source-code releases) that failed in each stage. In both installation and run-time stages, the majority of the TPLs are released as source-code releases. In contrast, the binary releases are less error-prone as they do not need to be built locally.

Figure 3 (c) shows that most problematic TPLs (refer to the TPL with PDSpecErrs) failed with one or two Python versions, and only a few releases failed in more than three.

We also identified libraries and repositories that might be affected by these 299 TPLs on Libraries.io [23]. Our findings reveal that these failed TPLs may cause significant disruptions across more than 45k libraries and 182k repositories. Notably, 266 out of the 299 erroneous TPLs do not specify Python requirements with "`python_requires`", even if some of them are incompatible with certain Python distributions.

> **Finding 1**: *PDSpecErrs appear often (299/3000) among popular Python libraries. They may cause installation failures and run-time errors, mostly manifest with one or two Python versions.*

Notably, our primary aim is to investigate the prevalence of PDSpecErrs, even if some PDSpecErrs could not be discovered during the exploration due to some reasons. Despite its incompleteness, we've already observed a significant prevalence of the issue. We believe that with further in-depth research, more related issues will be discovered.

3.3. Observation 2 (Characteristics)

To further characterize the problematic TPLs (299 TPLs), we analyzed problematic TPLs and the normal TPLs (it refers to the TPLs without

PDSpecErrs, 2701 TPLs) and compared them in multiple aspects, aiming to identify some distinct characteristics of the problematic TPLs to provide insights for fostering a more reliable development ecosystem. While these characteristics may not directly help to detect PDSpecErrs, they offer useful insights in deriving practical suggestions and guidelines for developers.

For each subject TPL, we retrieved its license (stored in the core metadata specifications), number of version releases, and the release time of the earliest and the latest versions from Libraries.io [23], and retrieved the other metadata information and the topic information from the project web page in PyPI [1]. Many characteristics were investigated, but only those showing significant differences are discussed. Among these, the number of forks and contributor statistics are used to evaluate how maturity affects PDSpecErr occurrences. The rationale is that a mature library tends to attract more attention, leading to more stars, forks, and contributors. Version release statistics and average update frequency reflect the activity level of the project, while the count of Python files offers insights into its structure. Additionally, license type and project topic are considered as metrics that influence how a TPL is used.

(1) Version Releases Statistics. As figure 4 (1) shows, the interquartile ranges (IQR) for both groups demonstrate that, on average, the version releases of the problematic TPLs are more than that of the normal TPLs.

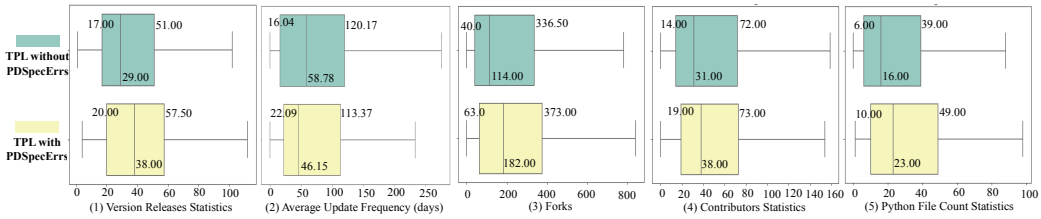


Figure 4: Comparing statistics of TPLs with and without PDSpecErrs.

(2) Average Update Frequency. This character of a TPL is defined as the total time elapsed between its earliest and latest releases, divided by its number of releases. As figure 4 (2) shows, the problematic TPLs exhibit slightly lower quartile values (Q3: 113.37, and median: 46.15), implying that problematic TPLs update more frequently than normal ones. We suspect that these problematic TPLs evolve more rapidly and are not always thoroughly tested, especially on the compatibility of Python versions.

The characteristics observed above suggest that the problematic TPLs tend to evolve faster with continuous updates, which may have contributed to their instability in terms of Python compatibility. It is noteworthy that the problematic TPLs are among the top 3,000 ranked libraries on Libraries.io, indicating a high degree of popularity and maturity. Yet, despite their mature status, our observation further accentuates the subtlety of PDSpecErrs, highlighting the need for designing detection tools to help identify them.

(3) Number of Forks. As figure 4 (3) shows, problematic TPLs tend to be forked more frequently, indicating that more TPL developers are interested in modifying and contributing to them.

(4) Number of Contributors. As figure 4 (4) shows, the first quartile and median (Q1: 19.00, median: 38.00) of problematic TPLs are higher than that of normal ones (Q1: 14.00, median: 31.00), suggesting a potentially larger and more active contributor community.

(5) Number of Python Files. The number of Python Files is calculated by the total count of Python files present within a given TPL. As figure 4 (5) shows, problematic TPLs tend to have a larger number of Python files, indicating potentially more complex project structures. We suspect that more complex project structures may increase the possibility of using incompatible Python version features.

The data suggests a counter-intuitive fact that accepting contributions from larger developer community may not help in identifying and fixing PDSpecErrs in TPLs. In fact, this may further contribute to their instability, which is amplified by the project complexity.

Finding 2: *PDSpecErr issues may persist in highly popular, actively updated, and widely contributed TPLs. Frequent upgrades and complex project structures could potentially contribute to the instability in Python version compatibility. This finding highlights the challenge and importance of detecting PDSpecErrs in TPLs.*

To understand if there is any obvious pattern for problematic TPLs, we calculated the error ratios across project domains and license types.

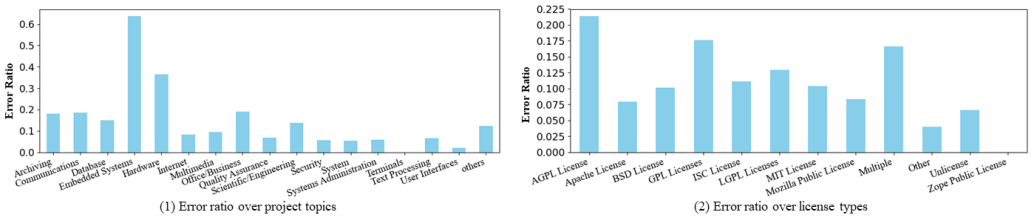


Figure 5: Error Ratio of TPLs for various Project Topics and Licenses.

(6) Error Ratio of TPLs in various Project Domains. "Topic" is used to provide additional information about the subject matter or domain of the TPL. This error ratio is calculated by dividing the total number of problematic TPLs appearing under a topic by the total number of TPLs of the same topic. As shown in figure 5 (1), the ratio of TPLs with PDSpecErrs on hardware related topics is high, accounting for 36% (hardware) and 63% (embedded system), while the ratios on other topics are much lower. We further calculated the average number of repositories that depend on the TPLs of each topic (also retrieved via [23]).

Conversely, topics with lower error ratios have a higher number of dependent repositories, with a minimum of 791. This seems to suggest that TPLs that are referenced by more repositories are less prone to PDSpecErrs, and hardware-related TPLs do not belong to this category. Another plausible explanation is that TPLs associated with topics that do not prioritize coding standards and focus solely on practicality, may neglect Python version compatibility issues. This oversight increases the likelihood of encountering PDSpecErrs in the TPLs.

(7) Error Ratio of TPLs for various Licenses. The license information is crucial for users and developers to understand the permissions and restrictions associated with using and distributing a particular software package, which may influence the stability and error rates of TPLs. For instance, more permissive licenses might attract a broader pool of contributors, potentially leading to more frequent updates but also a higher chance of introducing PDSpecErrs. This error ratio is calculated by dividing the total number of problematic TPLs appearing under a type of license by the overall frequency of occurrences of the same license type. As shown in figure 5 (2), the GPL family appears to have a higher error ratio (AGPL:21.4%, GPL:17.6%, LGPL:12.9%) than the rest. We speculate this phenomenon arises due to the fact that the GPL license family restricts commercial redistribution, which

may limit participation of large software companies. On the other hand, Apache and similar licenses are widely adopted by commercial entities, potentially resulting in higher quality standards for TPLs and, consequently, a reduced likelihood of encountering PDSpecErrs.

Finding 3: *PDSpecErrs is more prevalent in hardware-related areas and in TPLs with GPL family licenses.*

3.4. Observation 3 (Manifestation Patterns)

Besides the inherent characteristics of TPLs with PDSpecErrs, we also analyzed PDSpecErrs' manifestation patterns based on their error messages. According to the stages where PDSpecErrs occur, we grouped PDSpecErrs into two types: *installation failures* and *run-time errors*. Specifically, we employed a heuristic algorithm to iteratively classify error messages of TPLs. Initially, we randomly selected a set of error messages and categorized them by referring to [24] where two authors independently reviewed and identified the issues, and any discrepancies between their assessments were resolved with the involvement of a third author. After each iteration, the remaining unclassified error messages are further categorized. We added a new category if the error message cannot be classified into the previous categories. Two authors primarily carried out the categorization. In cases of disagreement, a third author joined the discussion until a consensus was reached. Finally, we concluded four distinct manifestation patterns of installation failures and one of run-time errors.

(1) Installation failures. Based on the manually categorized statistical results, 170 TPLs encountered installation failures, 105 of which exhibited *dependency-choosing* failures, and 65 encountered build failures. Dependency-choosing failures refer to that `pip` fails to find any suitable TPL version for installation to proceed, which occurs at the initial stage of downloading libraries. They typically prompt an error message such as "`No matching release found for <lib-name>`" when attempting to choose and download a suitable release using `pip`. Among 105 failing TPLs, 56 are source-code releases, and the remaining 49 are binary releases. Such failures are relevant to both "`python_requires`" and "`install_requires`" because `pip` cannot find any TPL release specified in "`install_requires`" that is also compatible with the local Python interpreter as specified by "`python_requires`".

In such a case, at least one of the two entries may be misconfigured. For example, installing `flower 1.0.0` [25] with Python 3.5 would fail as one of its required dependencies `celery` (>=5.0.5) restricts Python version >=3.6.

The remaining 65 failures are referred to as *build failures*. These failures happen between library downloading and importing. Specifically, misconfiguring the entry, "`python_requires`", can lead downstream users to employ an unsuitable Python interpreter, resulting in build failures when `pip` has downloaded the TPL and executes the setup script (i.e., `setup.py`) to build a source-code release. The error log usually starts with "`Command errored out with exit status 1`." Although they constitute a small amount, we find that their symptoms are more diverse and appear in run-time errors as well. We further classify them into three types:

- *Missing required modules.* The following error messages usually contain "`ModuleNotFoundError: No module named <mod> ...`", or `ImportError: cannot import name <sub_mod> from <mod> ...`", denoting the setup script fails to find a (sub)module within the local Python environment. There are 34 TPL releases prompting such an additional build error message. Notably, 7 of the 34 releases fail due to missing necessary dependency TPLs, which are used in the setup scripts but not declared in the metadata entry "`requires-dist`" derived from the configuration entry "`install_requires`". The remaining 27 releases failed due to missing Python standard modules.

- *Invalid syntax features.* The following error message usually contains "`SyntaxError: invalid syntax...`", denoting `setup.py` uses a syntax feature not supported by the local Python environment. There are 19 TPL releases prompting such error messages.

- *Specific error messages.* Some additional error messages are more specific, with customized error messages in `setup.py`, providing detailed information, such as "`ERROR: You need Python 3.6 or greater to install <dis>`". There are 12 TPL releases prompting such a build error message.

(2) Run-time errors. There are 129 TPLs with PDSpecErrs which were successfully installed but experienced errors at run-time. The scope of the analysis on the 129 TPLs mainly focuses on the run-time errors caused by top-level

import errors. "`python_requires`" with incorrect values can result in *missing required modules* and *invalid syntax features*. However, the error messages are prompted when importing TPLs instead of building TPLs.

We think that the severity of errors may indeed vary depending on the stage of the pipeline. Among the 299 PDSpecErrs, TPP runtime errors may have the most severe consequences for they potentially lead to downstream application failures, financial loss, or other negative impacts. These errors are implicit, and downstream users may not notice them until the corresponding runtime exceptions happen. In comparison, installation failures have less influence for they occur in the early stage.

Finding 4: *PDSpecErrs that occur in the installation stage can be summarized into four patterns. The run-time failures caused by PDSpecErrs are more subtle and cannot be easily captured by specific patterns. Most PDSpecErrs in both stages are attributed to the incorrect configurations of "`python_requires`".*

3.5. Error Diagnostic Pattern Analysis

Based on Observation 3, we analyzed the manifestation patterns of the PDSpecErrs related to misconfigured entries and summarized three diagnostic patterns: *setup script issues*, *using incompatible features*, and *Python version conflicts*. The details are as follows.

Diagnostic pattern 1. Setup script issues. This diagnostic pattern manifest as inconsistencies in Python version specifications between the developers hardcoded checks and configuration entries in setup script, e.g., `setup.py`. The specific details of a library are defined as keyword arguments of a global function `setup()`, including the entry "`python_requires`". However, this entry is optional, and TPL developers can either make mistakes or not follow the common practice of specifying restricted Python distributions. In this case, a PDSpecErr is raised by a *script issue*, which can prevent a build tool from generating appropriate metadata "`requires-python`" [14].

Some developers purposely hard-code Python version checking instead of configuring the entry "`python_requires`". Their intentions cannot be reflected in this way. For example, the code snippet below, defined in `setup.py` of `typed_ast 1.4.1` [26], is responsible for checking whether the local Python version is below 3.3. Due to the absence of "`python_requires`",

the build tool cannot generate the appropriate metadata entry that reflects the true Python version restriction. In consequence, `pip` cannot make a correct filtering on the TPLs and may choose a release incompatible with a user's local environment. Finally, the user would assume that the TPL is without Python version restriction until he gets an installation failure message prompted by the customized version checking code. Our study uncovered 12 TPL releases that failed due to PDSpecErrs of this pattern.

```
1  if sys.version_info[0]<3 or sys.version_info[1]<3:
2     sys.exit('Error: typed_ast only runs on Python 3.3 and
     above.')
```

Diagnostic pattern 2. Using incompatible features. The diagnostic pattern manifest as developers might misconfigure the "`python_requires`" entry unintentionally by using syntax or importing standard modules specific to certain Python distributions, leading to version constraints they are not aware of. These unsupported features, termed as *'incompatible features',* can be presented in the setup script or the source code, causing build failures or run-time errors. Our study found 175 releases with such PDSpecErrs, consisting of 91 source and 84 binary releases. Examples are as follows:

(1) Incompatible features in the setup script can lead to TPL build failures. For example, the code snippet below belongs to `setup.py` of `discord 1.7.3` [27], where the keyword argument "`encoding`" in the function `open()` has been introduced since Python 3. As a result, the TPL would encounter an installation failure with an error message like "`TypeError: 'encoding' is an invalid keyword argument for this function`" with Python 2, but the entry "`python_requires`" does not specify the Python version restriction. Finally, the client-side tool `pip` cannot choose a proper release when the local environment is Python 2.

```
1  with open(path.join(this_directory, 'README.md'),
     encoding='utf-8') as f:
2           long_description = f.read()
```

(2) Incompatible features in a TPL's source code usually lead to TPL run-time errors, manifesting either import or syntax errors. PDSpecErrs under this pattern can lie in both binary and source releases. Take the code snippet below as an example, in `__init__.py`, `pyxel 1.4.3` [28] imports the class `collections.MutableSequence`. However, from Python 3.10, the class `collections.MutableSequence` is removed. As a result, `pyxel 1.4.3`

can be successfully installed with Python 3.10, but an error "`ImportError: cannot import name 'MutableSequence' from 'collections'`" will be thrown when importing the package.

```
1  import sys
2  import traceback
3  from collections import MutableSequence
```

Diagnostic pattern 3. Python version conflicts. This pattern is derived from the dependency conflict (DC) [6] between a TPL p and one of its dependencies. The dependency is incompatible with at least one Python version declared in p's configuration entry. `Pip` can sometimes resolve failures of unavailable dependencies by backtracking and finding a compatible earlier version, but certain situations remain unresolved, leading to *dependency-choosing failures* in some TPLs. 105 TPLs with PDSpecErrs are of this pattern. For example, `sklearn-pandas 2.2.0` does not set the configuration entry and thus is assumed compatible with all Python distributions. However, its dependency `pandas>=1.1.4` requires Python >=3.6.1. Therefore, `pip` fails to find a compatible version of `pandas` when installing `sklearn-pandas 2.2.0` with any Python <3.6.1.

3.6. Implications

PDSpecErrs are relatively common even in top-ranked TPLs and can result in downstream application failures and degrading user experiences. Thus, resolving PDSpecErrs is of great significance for the reliability of TPLs. Based on our observations, there are some effective steps that different TPL stakeholders can take to avoid PDSpecErrs.

TPL developers. Developers should try to avoid complex Python project structures, including having too many Python files in a TPL, as this could elevate the risk of PDSpecErrs. When collaborating with external contributors, coding standards and contribution guidelines should be in place to ensure compatibility with different Python versions (Finding 2). Developers should follow the common practice of explicitly configuring Python version specifications in the setup scripts. Developers should thoroughly examine setup scripts to avert human errors and closely monitor specifications to ensure consistency. To avoid incorrect configurations, developers should be aware of the version-specific Python features used and which Python distributions support them. TPL developers should keep Python version specifications up-to-date for its dependencies in order to prevent PDSpecErrs.

Downstream users. Downstream users should prioritize stable and more recent Python distributions, especially when working with TPLs that impose no restrictions. High popularity and frequent releases do not guarantee the stability of a TPL (Finding 2). Blindly trusting developers' specifications may lead to errors, as the information could become outdated. Users need to pay attention to configurations, and to be proactive in preventing problems. When PDSpecErrs occur, downstream users can assess error symptoms and inspect the corresponding configuration entry (Finding 4) to identify and address potential configuration errors. By adopting appropriate strategies based on these symptoms, users can mitigate subsequent issues.

Python community. The relevant entries for Python version specifications are only configured based on developers' best effort. The Python library managers should move towards a better enforcement for these important entries and provide a mechanism to verify their correctness and consistency.

PDSpecErrs detection. Existing TPL error detection approaches [6, 8, 29, 19] mainly focus on detecting dependency errors and conflicts. They assume that the configuration entries of TPLs are correct. However, our study shows that a large portion of TPLs do not set their configuration entries correctly, resulting in PDSpecErrs (Finding 1). Existing approaches cannot handle these PDSpecErrs properly. Our study investigates the diagnostic patterns of PDSpecErrs, which can guide us in detecting them effectively.

4. Detecting PDSpecErrs

Based on our study, we propose PyChecker, a lightweight tool to detect PDSpecErrs and assist developers in improving configuration quality before releasing a TPL. For a given TPL, PyChecker detects whether any PDSpecErrs are present. If identified, PyChecker generates a detailed bug report providing essential information about the PDSpecErrs. The details of PyChecker are as follows.

4.1. Detecting Setup Script Issues

Based on diagnostic pattern 1, PyChecker detects *setup script issues* by checking the consistency between the Python version specifications in commonly used configuration entries and the hard-coded Python version specifications. We have identified two root causes of the setup script for detecting PDSpecErrs: (1) the variable "`sys.version_info`" is (in)directly used in a

conditional statement, and (2) the configuration entry "`python_requires`" is missing in any configuration files. If this pattern matches, a potential PDSpecErr is detected.

4.2. Detecting Incompatible Features

To detect *incompatible features* (diagnostic pattern 2), we extract comprehensive knowledge of Python versions. Based on that, PyChecker analyzes the TPL source code to infer its compatible Python version range. If the declared Python versions in the TPL are not align with the inferred range, a PDSpecErr is found.

The knowledge of Python interpreter syntax features is extracted following the prior work [30, 19]. PyChecker derives the mapping relationships between Python versions and syntax features, as well as standard modules, expressed as $PY(v) = S, M_s$, where $PY(v)$ is a Python interpreter at version v, and S and M_s denote its supported syntax features represented as regular expressions and standard modules, respectively. Then PyChecker analyzes TPL's source code by employing the generated abstract syntax tree (AST) to extract imported standard modules M_{lib} and identify syntax features S_{lib}. Using this information, PyChecker infers a set of Python distributions PY_{lib}, which support the syntax features in S_{lib} and contain the standard modules in M_{lib}. Similarly to 4.1, PyChecker derives the declared Python versions PY_{dec} from the TPL's configuration entries. If $PY_{dec} - PY_{lib} \neq \emptyset$, a PDSpecErr is detected.

4.3. Detecting Version Conflicts

PyChecker detects *version conflicts* (diagnostic pattern 3) by checking the inconsistent Python version specifications between a TPL and its dependencies. For a given TPL, the straightforward way to do this is traversing all restrictive versions of its dependencies (including both the direct and transitive dependencies). To reduce the time cost, PyChecker employs an optimized detection method. The method is feasible based on two prerequisites.

(1) *Newer versions of a TPL tend to support newer Python distribution interpreters.* For example, `numpy 1.20.1` and `1.22.2` are compatible with Python 3.7-3.10 and 3.8-3.10, respectively. The oldest compatible Python version (3.8) of `numpy 1.22.2` is newer than that (Python 3.7) of `numpy 1.20.1`.

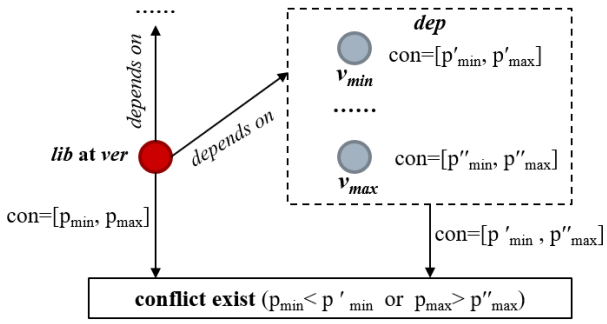


Figure 6: Inspecting Python version conflict.

(2) *The compatible Python versions for a set of successive TPL versions are also successive.* For example, `tensorflow 1.0.0` is compatible with Python 3.5, and `tensorflow 2.8.0` is compatible with Python 3.9. It can be inferred that the versions of `tensorflow` between 1.0.0 and 2.8.0 are compatible with one or more Python interpreters between 3.5 and 3.9, e.g., `tensorflow 2.1.3` and `2.2.3` are compatible with Python 3.6 and 3.8, respectively.

The rationale behind the optimized method is depicted in Figure 6. To compare the supported Python distributions between the target TPL's *lib* (version *ver*) and each dependency library *dep*, it first identifies the earliest and latest versions of *dep*, i.e., v_{min} and v_{max}. Then, it extracts the compatible Python version range of both, i.e., $[p'_{min}, p'_{max}]$ and $[p''_{min}, p''_{max}]$. Next, it constructs a compatible Python version range for *dep* with p'_{min} and p''_{max} as its lower- and upper-bound. Subsequently, PyChecker considers *dep* and *lib* at *ver* are conflicting in their declared compatible Python distributions if $p_{min} < p'_{min}$ or $p_{max} > p''_{max}$.

We also preliminarily investigated the applicability of the optimized method. We randomly sampled 500 TPLs in the top 3,000 and investigated how many TPLs satisfy the two prerequisites. The investigation reveals that 99.4% (497/500) and 99.8% (499/500) TPLs satisfy the first and the second prerequisites, respectively, indicating that the optimized method is applicable in most cases.

4.4. Bug Report

For a TPL release, PyChecker sequentially detects setup script issues, incompatible features, and version conflicts. After finishing the detection, if

one or more PDSpecErrs are found, PyChecker will generate a bug report for them. The report contains the key information of found PDSpecErrs, including the diagnostic pattern and the recommended version range of Python distributions. Below shows an example:

```
1 -- root cause 2: using incompatible feature--
2 cause : using incompatible feature
3 current python requires : >=3.6
4 ---------------------------------------
5 error file : ./src/astral/location.py
6 specific cause : module: <dataclasses> do not support python
      ['3.6']
7 ---------------------------------------
8 suggest python requires : >=3.7
```

5. Evaluation

To evaluate the effectiveness and usefulness of PyChecker, we utilize PyChecker to detect PDSpecErrs in TPLs and answer the following research questions.

- **RQ1 (Effectiveness)**: How effective is PyChecker in detecting PDSpecErrs?
- **RQ2 (Usefulness)**: Can PyChecker help library developers locate the root causes of PDSpecErrs?
- **RQ3 (Overhead)**: What is the computational overhead of PyChecker?

To answer **RQ1**, we used the 3,000 TPL releases collected in Section 3.1 as our evaluation dataset and applied PyChecker to detect PDSpecErrs. The dataset is divided into two parts. (1) TPLs with Confirmed PDSpecErrs (TCon). These instances account for 299 TPL releases with PDSpecErrs identified by our exploratory study. (2) TPLs with Potential PDSpecErrs (TPoten). No PDSpecErrs were identified for these 2,701 TPLs in our exploratory study. However, as described in Section 3.1, our exploratory study only focuses on the PDSpecErrs that occurred during the installation and importing stages. Considering that some more subtle PDSpecErrs may also occur at run-time, additional PDSpecErrs can still possibly be revealed with a deeper analysis. Notably, PyEGo [19] is a state-of-the-art related work addressing constraints regarding compatible Python versions of dependencies.

However, PyEGo assumes TPL configuration entries are accurate. Additionally, PyEGo can only generate a single Python version and cannot help pinpoint the root cause of an incompatible Python version because it relies on solving constraints. So, we can not use it for comparison.

To answer **RQ2**, we randomly selected 10% (80) of the 842 TPLs detected by PyChecker and manually verified them. Subsequently, we submitted bug reports to the corresponding TPL developers and maintainers. These reports explained how the detected PDSpecErrs may arise and the recommended suitable Python versions. The responses from developers and maintainers can serve as evidences for the practical usefulness of PyChecker.

To answer **RQ3**, we applied PyChecker to detect the 3,000 TPL releases collected in 3.1 to collect its time overhead.

We carry out the experiment on a computer running `Ubuntu 18.04 LTS` with 8-core 3.50 GHz CPU and 32 GB RAM.

5.1. Effectiveness

PyChecker successfully detected PDSpecErrs in 212 TPLs within TCon and 630 within TPoten, respectively. The results analysis are as follows.

TCon Detection Analysis. PyChecker detected PDSpecErrs of 212 TPL releases within TCon. Among them, 80 TPL releases encompass two diagnostic patterns, and four TPL releases encompass three diagnostic patterns. For example, `zappa 0.54.0` [31] exhibited a PDSpecErr involving diagnostic patterns 1 and 3. The experimental results prove that PyChecker can diagnose multiple PDSpecErrs stemming from different diagnostic patterns within a single TPL release. Furthermore, we investigated the remaining 87 TPL releases not detected by PyChecker and summarized two reasons.

(1) *Incomplete domain knowledge.* PyChecker, like other state-of-the-art tools [19, 30], struggles to fully model diverse and flexible syntax in Python. Certain features, such as contextual dependencies or runtime-evaluated constructs, remain unmodelled. For instance, the Postponed Evaluation of Annotations introduced in Python 3.7 cannot be statically modeled. As a result, PyChecker is unable to detect all PDSpecErrs under diagnostic pattern 2. 47 false positive are due to this reason.

(2) *Influence of pip's backtracking mechanism.* `Pip` for Python 3 has the backtracking mechanism since `20.3` [32], which improves the TPL installation success rate. This strategy referrs that pip makes initial assumptions about package versions and, if a conflict arises, backtracking to test alternative

versions until compatibility is ensured. However, `pip` for Python 2 does not support this mechanism. Some TPL releases experience *dependency-choosing failures* (diagnostic pattern 3) when a suitable dependency cannot be resolved with older `pip`. But PyChecker misses such PDSpecErrs since it assumes a newer `pip`. There are 40 false positives due to this reason. For example, `pip` failed to install `djangocms-text-ckeditor 4.0.0` [33] on Python 2.7 since a transitive dependency `Django` cannot be resolved. However, PyChecker finds a solution using newer `pip`, which comes with the direct dependency "`django-cms`==3.6" and the transitive dependency "`Django`<2.2, >=1.11".

TPoten Detection Analysis. PyChecker found PDSpecErrs in 630 TPL releases within TPoten. With limited time and human efforts, we randomly sampled 10% (63) of them for further analysis. We found that the preliminary method used in our exploration study in Section 3 missed some PDSpecErrs hidden deeply, and there are 53 TPLs that have been manually verified to contain PDSpecErrs, with the detailed reasons outlined below.

(1) *PDSpecErrs suppressed by other errors.* In our exploratory study (Section 3.1), some TPL installations failed and the error messages show non-PDSpec errors. Therefore, we missed some PDSpecErrs in our manual inspection because they are suppressed by other errors occurring before them. There are 37 TPLs of such cases. For example, installing `responder 2.0.5` [34] within Python 3.10 exhibits a C language-related error. However, PyChecker can detect a suppressed PDSpecErr associated with diagnostic pattern 3 as a transitive dependency `graphene` [35] requires Python <3.5.

(2) *PDSpecErrs caused by incompatible submodules.* The TPLs containing such PDSpecErrs do not encounter any error during their download, installation, and top-level module importing. The PDSpecErrs only occur when using the TPLs' specific submodules since the submodules use features incompatible with some specific Python versions. There are 16 TPLs of such cases. For example, during the installation and import of `apkid` on V2.1.1, everything proceeds smoothly. However, a SyntaxError occurs when we execute "`import apkid.rules`" on Python 2.7. PyChecker reported the detailed information "`specific cause: module: <typing> do not support python ['2.7']`".

The above analysis demonstrates that PyChecker can effectively detect deeper PDSpecErrs, which cannot be easily discovered through simple manual inspections.

Additionally, we reviewed the false positives by our tool (accounts for

about 15.8% in our sampled TPLs) and we summarize their causes below.

(1) *Imprecise matching rules.* TPL developers often define a considerable number of variable names that might unintentionally clash with Python's built-in keywords. This may confuse the matching rules used by PyChecker for inferring Python versions of a Python file. There are 6 of such cases. For instance, PyChecker reported a PDSpecErr because Python 2.7 does not support the "nonlocal" keyword, which appears in the `unparser.py` file of the `astunparse 1.6.3` library. However, "nonlocal" is only used as a part of a function name, "`def _Nonlocal(self, t): self.fill(''nonlocal'')`", which is accidentally matched by the PyChecker's rules.

(2) *Overlooked conditional dependencies.* Conditional dependencies refer to declaring and using different modules in a program regarding the Python version being used [19]. TPL developers may declare conditional dependencies for better backward compatibility. However, PyChecker currently does not recognize conditional dependencies precisely enough and may report spurious PDSpecErrs of diagnostic patterns 1 and 2. There are 4 of such cases. For example, in the case of `astral 2.2`, it declares a Python version constraint "`python_requires='>=3.6'`" and depends on "`dataclasses`" when using Python==3.6. PyChecker failed to recognize the conditional dependency on `dataclasses` and reported that "`module: <dataclasses> do not support in python ['3.6']`", because the module `dataclasses` only becomes a built-in module in Python 3.7 and above.

5.2. Usefulness

We randomly sampled about 10% (80) reported PDSpecErrs for further analysis. We manually examined these issues; if they were not genuine PDSpecErrs, we excluded them and randomly sampled new ones. During this process, we excluded five issues for they are false positives after investigation. Among them, 28 PDSpecErrs are fixed at their follow-up versions before our commit. Therefore, we reported the remaining 52 PDSpecErrs to the concerned developers on GitHub with bug reports generated by PyChecker. Each TPL received a consolidated bug report containing all PDSpecErrs categorized based on their corresponding root causes. For example, if one TPL contains two PDSpecErrs under two different causes, we will send one bug report, containing the two errors, to its corresponding GitHub repository. We categorize the developers' responses into four groups. "**Fixed**" refers to the issues that have been acknowledged and resolved by the developers. "**Ignored**" indicates that the developers have responded to the problems,

stating that they do not require fixing and providing a reason. "**Pending**" means that the developers have not responded or reacted to the reported issues until now. "**Unmaintained**" means the absence of any feedback or activity from the associated projects (such as commits, issue responses, or releases) since we submitted our issues, with inactivity persisting over a year, referring to previous work [36, 37].

As of April 1, 2024, the developers have replied to 35 of the issue reports, fixed 32 PDSpecErrs under our reports, and three were ignored. The first issue was submitted on October 28, 2022. For the ignored issue [38] associated with diagnosing pattern 1, the developer explained that `breathe` is an extension of `sphinx`, whose customized setup script intended to check the compatibility between `sphinx` and the local Python interpreter. However, the contributor also said that "*I guess this proposal wouldn't hurt anything if implemented.*" For the other ignored issue [39], we suggested setting the metadata entry with ">= 3.6" for the dependent library `wcmatch` at 8.1.2 or above requires Python >=3.6. However, developers wanted to find a version of `wcmatch` that supported lower Python distributions or a replacement for `wcmatch`, which indirectly suggests a PDSpecErr in this project.

Nine issues were pending by developers, and eight repositories were no longer updated after we submitted the issues. Among the nine pending issues, we found that one developer did not want their projects to be compatible with Py 2.7. He claimed that [40] "*Python 2.7 has reached its EOL for all security releases and, therefore, is not supported by this package.*" The developer hoped we would not use his library on Python 2.7 instead of constraining the Python version in the setup file. Nevertheless, the Python community still prefers to declare the Python version specifications clearly and precisely.

Thirty-two PDSpecErrs have been fixed, and their developers replied to us that they referenced our bug reports and successfully fixed the PDSpecErrs. For example, `LightGBM 3.3.2` is a popular and well-maintained TPL with 15.3k stars in GitHub. However, PyChecker still attracts the developers' attention. The developer approved and fixed it, and replied "*Thanks very much for the report, ..., I'd support a pull request that adds 'python_requires= >=3.5* ". The above results and feedbacks from developers indicate that PyChecker can find PDSpecErrs and the detailed bug report is useful for TPL developers to locate and fix PDSpecErrs in practice.

5.3. Overhead

PyChecker detects all the TPL releases on averagely 7.32 seconds, with maximum 282.28 seconds, minimum 2.169 seconds. To the best of our knowledge, no tools or methods available for runtime checking of PDSpecErrs. The closest approach is simulating downstream user behavior by installing and running the TPL in multiple simulated local environments. Since this would require several Docker environments, the time overhead would be at least in the range of minutes.

In terms of resource consumption, PyChecker involves parsing and checking the code, requiring relatively low computational resources. Runtime checking involves actually running each TPL repeatedly in multiple simulated environments which not only has significant time overhead but also consumes substantial computational resources, such as memory, CPU, and storage as the Docker containers are involved. Thus, the difference in resource consumption could also be a reason why no comparison was made, as runtime checking demands far more resources than static analysis.

6. Discussion

Internal threats to validity. The first threat comes from potential inaccuracies in our manual collection and analysis of PDSpecErrs. To mitigate this, we first find the installation and run-time failures in an automated way. Then three co-authors independently investigated the collected Python libraries and their errors, and finally reached a consensus through cross-validations and discussions.

The second threat is that the rapid evolution of the Python library management tool-chains may affect our study results. We used the latest `pip`, equipped with a backtracking strategy, to reduce its influence on dependency choosing. In addition, we focused on the recent versions of each TPL when launching our work, and we tracked the TPL's newly released version during this work to inspect whether the detected PDSpecErr has been fixed.

External threats to validity. The third validity concerns generality. We conducted our work with hundreds of TPLs at the top rank, which are popular and representative by following work [19, 10], ensuring that our study has a substantial influence on the Python community.

The fourth threat comes from the data collection process. Considering that some PDSpecErrs may also occur at run-time, our study cannot reveal

all potential PDSpecErrs without exercising any functionality. Thus, some PDSpecErrs can still be found among these TPLs. To mitigate the impact of this issue, we analyzed the remaining 2,701 TPLs using PyChecker in the experimental process, and sampled 10% of the analysis results to confirm the detection outcomes.

Limitations. First, our detection tool, PyChecker, mainly focuses on detecting PDSpecErrs caused by the two most commonly used configuration entries. Furthermore, although PyChecker provides the root causes of PDSpecErrs in TPLs and the recommended Python versions, it cannot fix the detected PDSpecErrs automatically. We plan to improve the ability of PyChecker to detect and fix more PDSpecErrs in TPLs.

Second, during the exploratory study, some PDSpecErrs are missed, but we believe they did not affect the results of the observations. Generally, the TPLs ranked lower are less mature than the top ones, and with an increase size in TPLs with PDSpecErrs, they should further support our observations. This underscores the prevalence and subtlety of the PDSpecErrs.

Third, we only submitted a subset of the detected PDSpecErrs due to resource limits. We will continue our efforts on this line to help improve the Python version compatibility of the ecosystem.

7. Related Work

Compatibility Analysis. Compatibility issues have garnered significant attention across various programming language ecosystems, including Python [6, 7], Java [8, 29], and JavaScript [41]. These concerns primarily center on incompatible dependencies among libraries, commonly referred as dependency conflict issues.

Several techniques [10, 11, 12, 19, 30] aim to infer compatible dependencies for code snippets or Python programs. PyDFix [9] specializes in detecting and resolving build unreproducibility in Python builds caused by third-party library version errors.

Some studies focus on identifying and rectifying backward compatibility issues arising from library upgrades [42, 43, 44, 45, 46, 47, 48, 49]. In particular, the Android community exhibits more intricate and diverse behaviors with compatibility, consequently heightening concerns regarding API compatibilities [50, 51, 52, 53, 54].

Unlike them, we concentrate on the Python version incompatibilities caused by inconsistent version specifications. Additionally, these works ad-

dress dependency errors from the perspective of downstream users based on the assumption that TPL configuration entries are correct. In contrast, PyChecker stands in the perspective of TPL developers, and it can detect PDSpecErrs and warn developers to correct TPL configurations.

Misconfigurations. A closely related topic is the detection of software misconfigurations [55, 56, 57, 58]. These misconfigurations are primarily attributed to the evolution of systems [59, 60, 61], where existing test cases may play a crucial role in identifying the issues. In the absence of test cases, Huang et al. [62] proposed to encode common issue patterns to automatically extract detection rules for configuration compatibility issues in Android apps. A few other works [63, 64, 65, 66, 67, 68] automatically diagnose misconfigurations by injecting configuration errors into the systems and evaluating them. For example, ConfVD [66] generates configuration errors by violating the options' specifications.

These research works focus on the misconfiguration issues on the client side. Their application scenario differs from that of PyChecker, which detects PDSpecErrs from the perspective of TPL developers to identify incompatibilities caused by inconsistent version specifications.

Studies on open-source package-management systems. Current empirical studies mainly focus on dependency errors and security issues in open-source library management systems. Abdalkareem et al. [69] studied the impact of using trivial libraries with two large platforms npm and PyPI. [70] focuses on third-party libraries' usages, updates, and risks in open-source Java projects. Zimmermann et al. [71] studied the security risks of npm by analyzing dependencies between libraries, the maintainers, and security issues. [72] found that PyPI is an attractive target for attackers to trick developers into using malicious libraries. Some exploratory studies focus on the structure and characteristics of deep learning supply chains [73] and dependency bugs in deep learning technology stack [74].

To our knowledge, this is the first work regarding Python version incompatibilities in the Python open-source ecosystem.

8. Conclusion

PDSpecErrs are common and can cause TPL installation failures and runtime errors, but they have received little attention. This work conducted the

first exploratory study on PDSpecErrs regarding their prevalence, characteristics, manifestation patterns, and diagnostic patterns. Motivated by our exploratory study, we designed PyChecker, which can automatically detect PDSpecErrs associated with the three diagnostic patterns. Evaluation results show that PyChecker can effectively detect PDSpecErrs, and the generated bug report can help TPL developers fix PDSpecErrs. In the future, we plan to test PyChecker against a different dataset and enhance the detection capability of PyChecker and extend our technique for implementation on PyPI servers or a dedicated web service. This would allow for simpler checks and reporting of any such problems.

Acknowledgments

This research was partially funded by the Major Project of Institute of software, University of Chinese Academy of Science under Grant No. ISCAS-ZD-202302.

References

[1] Pypi, https://pypi.org/ (2024).

[2] pip documentation v22.3, https://pip.pypa.io/en/stable/ (2019).

[3] goatools 1.2.3, https://pypi.org/project/goatools/ (2022).

[4] K.-P. Yee, Exception chaining and embedded tracebacks (2005). URL https://www.python.org/dev/peps/pep-3134/#abstract

[5] Tensorflow issues, "https://github.com/tensorflow/tensorflow/issues/62189" (2023).

[6] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, Z. Zhu, Watchman: monitoring dependency conflicts for python library ecosystem, in: Proceedings of International Conference on Software Engineering (ICSE), 2020, pp. 125–135.

[7] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, S.-C. Cheung, Do the dependency conflicts in my project matter?, in: Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2018, pp. 319–330.

[8] Y. Wang, M. Wen, R. Wu, Z. Liu, S. H. Tan, Z. Zhu, H. Yu, S.-C. Cheung, Could i have a stack trace to examine the dependency conflict issue?, in: Proceedings of International Conference on Software Engineering (ICSE), 2019, pp. 572–583.

[9] S. Mukherjee, A. Almanza, C. Rubio-González, Fixing dependency errors for python build reproducibility, in: Proceedings of SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2021, pp. 439–451.

[10] E. Horton, C. Parnin, Dockerizeme: Automatic inference of environment dependencies for python code snippets, in: Proceedings of International Conference on Software Engineering (ICSE), 2019, pp. 328–338.

[11] E. Horton, C. Parnin, V2: fast detection of configuration drift in python, in: Proceedings of International Conference on Automated Software Engineering (ASE), 2019, pp. 477–488.

[12] J. Wang, L. Li, A. Zeller, Restoring execution environments of jupyter notebooks, in: Proceedings of International Conference on Software Engineering (ICSE), 2021, pp. 1622–1633.

[13] setuptools, "https://pypi.org/project/setuptools/" (2022).

[14] Packaging and distributing projects, "https://packaging.python.org/en/latest/guides/distributing-packages-using-setuptools/" (2022).

[15] Configuring setuptools using pyproject.toml files, "https://setuptools.pypa.io/en/latest/userguide/pyproject_config.html" (2022).

[16] Python packaging authority, "https://www.pypa.io/en/latest/" (2021).

[17] Python specifications core-metadata, "https://packaging.python.org/en/latest/specifications/core-metadata/" (2023).

[18] Pep 301, "https://peps.python.org/pep-0301" (2002).

[19] H. Ye, W. Chen, W. Dou, G. Wu, J. Wei, Knowledge-based environment dependency inference for python programs, in: Proceedings of International Conference on Software Engineering (ICSE), 2022.
URL https://github.com/PyEGo/PyEGo

[20] Libraries.io, https://libraries.io/ (2024).

[21] J. Huang, N. Borges, S. Bugiel, M. Backes, Up-to-crash: Evaluating third-party library updatability on android, in: 2019 IEEE European Symposium on Security and Privacy (EuroS&P), IEEE, 2019, pp. 15–30.

[22] docker-compose 1.29.2, "https://pypi.org/project/docker-compose/" (2021).

[23] goatools 1.2.3, https://libraries.io/api (2023).

[24] J. Wang, G. Xiao, S. Zhang, H. Lei, Y. Liu, Y. Sui, Compatibility issues in deep learning systems: Problems and opportunities, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 476–488.

[25] flower 1.0.0, https://pypi.org/project/flower/1.0.0/ (2021).

[26] typed-ast 1.4.1, "https://pypi.org/project/typed-ast/1.4.1/" (2020).

[27] discord 1.7.3 (2022).
URL https://pypi.org/project/discord/

[28] pyxel 1.4.3 (2020).
URL https://pypi.org/project/pyxel/1.4.3/

[29] K. Huang, B. Chen, B. Shi, Y. Wang, C. Xu, X. Peng, Interactive, effort-aware library version harmonization, in: Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2020, pp. 518–529.

[30] W. Cheng, X. Zhu, W. Hu, Conflict-aware inference of python compatible runtime environments with domain knowledge graph, in: Proceedings of the 44th International Conference on Software Engineering, ICSE

'22, Association for Computing Machinery, New York, NY, USA, 2022, p. 451–461. doi:10.1145/3510003.3510078.
URL https://doi.org/10.1145/3510003.3510078

[31] zappa 0.54.0, https://pypi.org/project/zappa/0.54.0/ (2021).

[32] Changes to the pip dependency resolver in 20.3 (2020), https://pip.pypa.io/en/stable/user_guide/#resolver-changes-2020 (2020).

[33] djangocms-text-ckeditor 4.0.0, "https://pypi.org/project/djangocms-text-ckeditor/4.0.0/" (2020).

[34] responder 2.0.5, https://pypi.org/project/responder/2.0.5/ (2019).

[35] httptools 3.1.1, https://pypi.org/project/graphene/ (2022).

[36] J. Coelho, M. T. Valente, Why modern open source projects fail, in: Proceedings of the 2017 11th Joint meeting on foundations of software engineering, 2017, pp. 186–196.

[37] J. Coelho, M. T. Valente, L. L. Silva, E. Shihab, Identifying unmaintained projects in github, in: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2018, pp. 1–10.

[38] Issue 810. breathe, "https://github.com/breathe-doc/breathe/issues/810" (2022).

[39] Issue 246. pycasbin, "https://github.com/casbin/pycasbin/issues/246" (2022).

[40] Issue 281. django-stdimage, "https://github.com/codingjoe/django-stdimage/issues/281" (2022).

[41] J. Patra, P. N. Dixit, M. Pradel, Conflictjs: finding and understanding conflicts between javascript libraries, in: Proceedings of International Conference on Software Engineering (ICSE), 2018, pp. 741–751.

[42] S. Mostafa, R. Rodriguez, X. Wang, A study on behavioral backward incompatibility bugs in java software libraries, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017, pp. 127–129. doi:10.1109/ICSE-C.2017.101.

[43] D. Foo, H. Chua, J. Yeo, M. Y. Ang, A. Sharma, Efficient static checking of library updates, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 791–796. doi:10.1145/3236024.3275535.
URL https://doi.org/10.1145/3236024.3275535

[44] G. Mezzetti, A. Möller, M. T. Torp, Type Regression Testing to Detect Breaking Changes in Node.js Libraries (Artifact), Dagstuhl Artifacts Series 4 (3) (2018) 8:1–8:2. doi:10.4230/DARTS.4.3.8.
URL http://drops.dagstuhl.de/opus/volltexte/2018/9239

[45] M. A. Saied, H. Sahraoui, E. Batot, M. Famelis, P.-O. Talbot, Towards the automated recovery of complex temporal api-usage patterns, GECCO '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 1435–1442. doi:10.1145/3205455.3205622.
URL https://doi.org/10.1145/3205455.3205622

[46] A. Møller, M. T. Torp, Model-based testing of breaking changes in node.js libraries, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 409–419. doi:10.1145/3338906.3338940.
URL https://doi.org/10.1145/3338906.3338940

[47] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, Y. Xiong, How do python framework apis evolve? an exploratory study, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 81–92. doi:10.1109/SANER48275.2020.9054800.

[48] L. Chen, F. Hassan, X. Wang, L. Zhang, Taming behavioral backward incompatibilities via cross-project testing and analysis, ICSE '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 112–124. doi:10.1145/3377811.3380436.
URL https://doi.org/10.1145/3377811.3380436

[49] C. Zhu, M. Zhang, X. Wu, X. Xu, Y. Li, Client-specific upgrade compatibility checking via knowledge-guided discovery, ACM Trans. Softw.

Eng. Methodol. 32 (4) (may 2023). doi:10.1145/3582569.
URL https://doi.org/10.1145/3582569

[50] H. Huang, L. Wei, Y. Liu, S.-C. Cheung, Understanding and detecting callback compatibility issues for android applications, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 532–542. doi:10.1145/3238147.3238181.
URL https://doi.org/10.1145/3238147.3238181

[51] L. Li, T. F. Bissyandé, H. Wang, J. Klein, Cid: Automating the detection of api-related compatibility issues in android apps, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 153–163. doi:10.1145/3213846.3213857.
URL https://doi.org/10.1145/3213846.3213857

[52] L. Wei, Y. Liu, S. C. Cheung, Pivot: Learning api-device correlations to facilitate android compatibility issue detection, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (2019) 878–888.

[53] P. Liu, L. Li, Y. Yan, M. Fazzini, J. Grundy, Identifying and characterizing silently-evolved methods in the android api, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2021, pp. 308–317. doi:10.1109/ICSE-SEIP52600.2021.00040.

[54] Y. Zhao, L. Li, K. Liu, J. Grundy, Towards automatically repairing compatibility issues in published android apps, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 2142–2153. doi:10.1145/3510003.3510128.
URL https://doi.org/10.1145/3510003.3510128

[55] A. Rabkin, R. Katz, Static extraction of program configuration options, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 131–140.

[56] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, S. Pasupathy, Early detection of configuration errors to reduce failure damage, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 619–634.

[57] Q. Chen, T. Wang, O. Legunsen, S. Li, T. Xu, Understanding and discovering software configuration dependencies in cloud and datacenter systems, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 362–374.

[58] J. Toman, D. Grossman, Staccato: A bug finder for dynamic configuration updates (artifact)., Dagstuhl Artifacts Ser. 2 (1) (2016) 14–1.

[59] F. Behrang, M. B. Cohen, A. Orso, Users beware: Preference inconsistencies ahead, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 295–306.

[60] S. Zhang, M. D. Ernst, Which configuration option should i change?, in: Proceedings of the 36th international conference on software engineering, 2014, pp. 152–163.

[61] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, T. Xu, An evolutionary study of configuration design and implementation in cloud systems, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 188–200.

[62] H. Huang, M. Wen, L. Wei, Y. Liu, S.-C. Cheung, Characterizing and detecting configuration compatibility issues in android apps, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2021, pp. 517–528.

[63] M. Attariyan, M. Chow, J. Flinn, X-ray: automating {Root-Cause} diagnosis of performance anomalies in production software, in: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), 2012, pp. 307–320.

[64] S. Zhang, M. D. Ernst, Proactive detection of inadequate diagnostic messages for software configuration errors, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015, pp. 12–23.

[65] W. Li, Z. Jia, S. Li, Y. Zhang, T. Wang, E. Xu, J. Wang, X. Liao, Challenges and opportunities: an in-depth empirical study on configuration error injection testing, in: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021, pp. 478–490.

[66] S. Li, W. Li, X. Liao, S. Peng, S. Zhou, Z. Jia, T. Wang, Confvd: System reactions analysis and evaluation through misconfiguration injection, IEEE Transactions on Reliability 67 (4) (2018) 1393–1405.

[67] W. Li, S. Li, X. Liao, X. Xu, S. Zhou, Z. Jia, Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation, in: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, 2017, pp. 88–97.

[68] F. A. Arshad, R. J. Krause, S. Bagchi, Characterizing configuration problems in java ee application servers: An empirical study with glassfish and jboss, in: 2013 IEEE 24th international symposium on software reliability engineering (ISSRE), IEEE, 2013, pp. 198–207.

[69] R. Abdalkareem, V. Oda, S. Mujahid, E. Shihab, On the impact of using trivial packages: An empirical case study on npm and pypi, Empirical Software Engineering 25 (2) (2020) 1168–1204.

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

[71] M. Zimmermann, C.-A. Staicu, C. Tenny, M. Pradel, Small world with high risks: A study of security threats in the npm ecosystem, in: Proceedings of {USENIX} Security Symposium ({USENIX} Security), 2019, pp. 995–1010.

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

[73] X. Tan, K. Gao, M. Zhou, L. Zhang, An exploratory study of deep learning supply chain, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 86–98.

[74] K. Huang, B. Chen, S. Wu, J. Cao, L. Ma, X. Peng, Demystifying dependency bugs in deep learning stack, arXiv preprint arXiv:2207.10347 (2022).