2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00175

Understanding and Remediating Open-Source License Incompatibilities in the PyPI Ecosystem

Weiwei Xu*, Hao He*, Kai Gao, Minghui Zhou†
School of Computer Science and School of Software & Microelectronics, Peking University, Beijing, China
Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
xuww@stu.pku.edu.cn, {heh, gaokai19, zhmh}@pku.edu.cn

***Abstract*—The reuse and distribution of open-source software must be in compliance with its accompanying open-source license. In modern packaging ecosystems, maintaining such compliance is challenging because a package may have a complex multi-layered *dependency graph* with many packages, any of which may have an incompatible license. Although prior research finds that license incompatibilities are prevalent, empirical evidence is still scarce in some modern packaging ecosystems (e.g., PyPI). It also remains unclear how developers remediate the license incompatibilities *in the dependency graphs* of their packages (including direct and transitive dependencies), let alone any automated approaches.**

To bridge this gap, we conduct a large-scale empirical study of license incompatibilities and their remediation practices in the PyPI ecosystem. We find that 7.27% of the PyPI package releases have license incompatibilities and 61.3% of them are caused by transitive dependencies, causing challenges in their remediation; for remediation, developers can apply one of the five strategies: migration, removal, pinning versions, changing their own licenses, and negotiation. Inspired by our findings, we propose SILENCE, an SMT-solver-based approach to recommend license incompatibility remediations with minimal costs in package dependency graph. Our evaluation shows that the remediations proposed by SILENCE can match 19 historical real-world cases (except for migrations not covered by an existing knowledge base) and have been accepted by five popular PyPI packages whose developers were previously unaware of their license incompatibilities.

I. INTRODUCTION

Open-source licenses dictate the terms and conditions regarding how a piece of open-source software (OSS) can be reused, modified, and redistributed [1]. As of April 2023, the Open Source Initiative (OSI) has approved 117 open-source licenses [2], ranging from highly restrictive ones (e.g., GPL 3.0 [3]) to highly permissive ones (e.g., MIT [4]). When developers incorporate OSS into their projects, it is critical to comply with all the terms and conditions declared in the license of the OSS. Failure to do so can result in ethical, legal, and monetary consequences [5], [6].

As OSS thrives, modern software development is increasingly dependent on the reuse of OSS packages from major packaging ecosystems (e.g., PyPI [7], Maven [8], npm [9]). On the other hand, the legal risks of reusing OSS packages from packaging ecosystems are high because packages form complex dependency networks in which one package can directly or transitively depend on hundreds of other packages [10]. Any of the dependent packages may have a very restrictive license, which can easily introduce license violations for any package or downstream project depending on them.

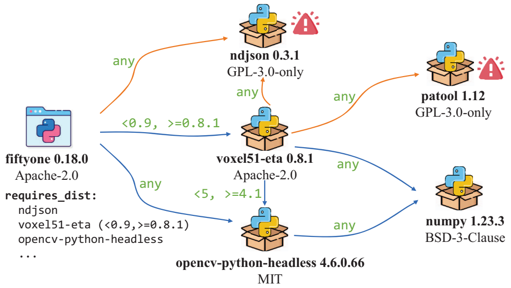


Fig. 1. License incompatibilities in `fiftyone 0.18.0` when it is released.

In this paper, we consider the *license incompatibility* issue occurring when an OSS package release[1] depends on another release whose license is incompatible with its own license. License incompatibilities can arise from both direct and transitive dependencies in a release's dependency graph [12], [13]. By *dependency graph* (sometimes also referred to as dependency tree [14]), we mean a directed graph with a release as the root node, releases that the root node directly or transitively depends on as other nodes, and direct dependency relationships between nodes as edges. A dependency graph represents all upstream dependencies of a release and is resolved using a dependency resolver such as `pip` [15] or `Poetry` [16].

For example, Figure 1 illustrates a part of the dependency graph for `fiftyone 0.18.0` when it is released on November 10th, 2022. We can observe that `fiftyone 0.18.0` depends on two GPL-3.0-licensed releases, i.e., `ndjson 0.3.1` and `patool 1.12`. However, `fiftyone 0.18.0` itself is licensed under Apache 2.0, which violates the requirement of GPL 3.0 that any of its dynamically linked derivative work should be also licensed under a GPL license (as interpreted by the Free Software Foundation [17]). Such license incompatibilities can happen for many reasons, including but not limited to: 1) developers may pay insufficient attention to OSS licensing or have insufficient knowledge about OSS licensing [18], [19]; 2) dependency graphs dynamically change over time [14] and packages may change licenses in new releases [20], [21]; 3) developers may only manage direct dependencies, overlooking

*Both authors contributed equally to this paper.
†Minghui Zhou is the corresponding author.

[1] In this paper, we align with the terminology of PyPA [11] and use the term *release* to refer to a specific version of a package. For example, `fiftyone 0.18.0` is one of the releases of `fiftyone` with version number `0.18.0`.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00175

or lacking enough control over transitive dependencies [22].

Past research has revealed the prevalence of license incompatibilities in npm and RubyGem [12], [13] and techniques are proposed to detect incompatibilities [1], [23]–[28]. An earlier study [25] provided guidance on reusing OSS components to avoid license incompatibilities. However, to the best of our knowledge, other packaging ecosystems are understudied and little is known about how developers remediate license incompatibilities *in the dependency graph*. Such knowledge is important for the design of tools to support this process.

To bridge the aforementioned gap, we begin with a large-scale empirical study in the PyPI ecosystem, one of the most thriving packaging ecosystems in recent years. To enable this study, we build an up-to-date dataset containing licensing and dependency information of 3,622,711 releases from 438,967 PyPI packages. Our study answers these research questions:

- **RQ1**: *What is the distribution of licenses and how does licensing evolve in the PyPI ecosystem?*
- **RQ2:** *What is the distribution of license incompatibilities in the dependency graphs of PyPI releases?*
- **RQ3:** *How do PyPI package developers respond to and remediate license incompatibilities in practice?*

Inspired by our findings, we propose SILENCE, an SMT-solver-based incompatibility remediator for licenses in the dependency graph. Given a release and its dependency graph with one or more license incompatibilities, SILENCE 1) finds alternative licenses that are compatible with the dependency graph, and 2) searches for alternative graphs with no license incompatibilities and minimal changes compared to the original graph (i.e., indicating minimal remediation costs). The results are aggregated as a report of recommended remediations (i.e., migrations, removals, version pinnings, or license changes) for developers to consider and choose. Our evaluation shows that the results of SILENCE can match the remediations proposed by developers in 19 historical real-world cases except when the migration is not covered by an existing knowledge base [29]. We further identify and report license incompatibilities that are still present in nine popular PyPI packages, five of which have been confirmed and remediated by package developers following one of the SILENCE's suggestions.

In summary, the contributions of this paper are as follows:

- We build an up-to-date dependency and licensing dataset for the PyPI ecosystem, laying the foundation for license incompatibility analysis and remediation.
- We conduct the first large-scale empirical study to confirm the prevalence of license incompatibilities in PyPI and reveal developers' remediation practices.
- We design and evaluate a novel SMT-solver-based approach, SILENCE, for recommending actions to remediate license incompatibilities in Python dependency graphs.

II. Related Work

OSS licenses and licensing are studied in both software engineering and information system research. We review related work in three main realms: license identification, license usage and evolution, and license incompatibility detection.

License Identification. The first step of any license-oriented research is the identification of licenses and/or license terms in OSS, which can be difficult in the absence of clean and curated data sources. Therefore, researchers have proposed various approaches to identify licenses, or some specific license terms, from source code, binary files, or text [30]–[34]. There are also open-source tools for this purpose, such as `ScanCode` [35] and `Licensee` [36]. To facilitate the automated processing of OSS licensing information, the Linux Foundation proposed the Software Package Data Exchange (SPDX) standard in which a list of standard license identifiers is defined [37].

License Usage and Evolution. Di Penta et al. [38] studied the licensing evolution of six OSS systems, concluding that they underwent frequent and substantial changes with variable patterns. Comino and Manenti [39] proposed a model to explain the commercial benefits of dual-licensed OSS. Vendome et al. [20], [21] conducted a large-scale mixed-method study on 16,221 Java projects; they discovered a clear trend toward the use of less restrictive licenses mainly for facilitating reuse. In the context of JavaScript projects, studies analyzed the use of non-approved OSI licenses [40] and multi-licensing [41].

License Incompatibility Detection. Perhaps the most important topic in OSS licensing is to check if some software is legally compliant with all the OSS it depends on, as violations can lead to legal, monetary, and ethical consequences [5], [6].

Licensing issues can manifest in many ways (see a comprehensive taxonomy in Vendome et al. [42]), but most research effort is focused on checking license incompatibilities between common, known OSS licenses. German et al. [25] developed a model for license incompatibility and performed case studies on how different software systems address incompatibilities. Further studies proposed approaches to understand and check license incompatibilities in the Fedora Linux distribution [43], Android apps [26], and Java applications [27], [28]. Kapitsaki et al. [44] proposed a general process based on SPDX. Wolter et al. [45] studied license inconsistencies within GitHub repositories, finding that many of the most popular ones do not fully declare all the licenses found in their source code.

In the context of packaging ecosystems, Qiu et al. [12] find that 0.644% of npm packages have license incompatibilities and developers face difficulties in managing them. Considering more licenses and the entire ecosystem, Makari et al. [13] find that 7.3% of npm packages and 13.9% of RubyGem packages contain license incompatibilities. Pfeiffer [46] studied incompatibilities caused by the AGPL license in seven ecosystems, concluding that incompatibilities are present in all ecosystems, among which PyPI and Maven packages are most risky.

Other studies explored the possibility of using fine-grained analysis on license terms to find incompatibilities in arbitrary licenses, using argumentation system [47] or learning-based approaches [1]. For example, Xu et al. [1] proposed LIDETECTOR, an NLP-based method to interpret any OSS license and detect incompatibilities. Their analysis of 1,846 GitHub projects revealed that 72.91% of them have license incompatibilities, but they did not consider license incompatibilities *in the dependency graph*. Researchers also studied the developers' understanding of OSS licensing [18], [19], [48], proposed license recommendation tools [49]–[51], and investigated the impact of OSS licensing on different topics [6], [52]–[54].

To the best of our knowledge, none of the previous studies have investigated how packaging ecosystem developers remediate license incompatibilities in the dependency graph of a specific package. Such understanding is critical for the design of automated tools to address developers' demand in remediating such incompatibilities (as shown in Qiu et al. [12]). Among different packaging ecosystems, PyPI is understudied in OSS licensing (the only study on PyPI [46] investigated only the AGPL license) but highly popular (currently the 3rd largest packaging ecosystem with rapid growth [55]). This motivates us to instantiate our study in the PyPI ecosystem.

III. The PyPI Dependency & Licensing Dataset

To provide a foundation for license incompatibility analysis and remediation in the dependency graph, we build a dataset with the dependency and licensing information of the entire PyPI ecosystem as of November 2022. In this Section, we will describe the dataset and its construction process in detail.

A. PyPI Dependency Data

1) Data Collection: We begin with a complete PyPI distribution metadata dump obtained from the official dataset hosted on Google BigQuery [56] in November 2022. The dump contains 438,967 packages with 3,622,711 different releases, and each release may have multiple distributions (e.g., intended for different operating systems or Python versions). For each distribution, the metadata provides a `requires_dist` field specifying other packages required by this distribution, optionally with version constraints and extra markers (as defined by PEP 508 [57], see an example in Figure 1). We observe that for the same release, the `requires_dist` fields of different distributions are almost always consistent.[2] For convenience, we arbitrarily select the `requires_dist` from one distribution as the dependencies of a particular release.

2) Dependency Resolution: The `requires_dist` field only encodes a *specification* of direct dependencies which is a list of requirement strings [59]. Using a dependency solver (e.g., `pip` [15] or `Poetry` [16]), the specification can be solved into a *concrete* dependency graph, with all dependencies (direct and transitive) and their versions. Unfortunately, the relationship between dependency specifications and dependency graphs is loose: the same specification can result in different dependency graphs at different times due to new package releases, flexible version constraints, and changes in the dependency solver [14], [60], [61]. For the purpose of longitudinal analysis, we need to restore the dependency graph of a specific release at any past time of interest. Thus, we implement a custom dependency solver following the algorithm described in Wang et al. [62], which imitates the breadth-first search behavior of `pip` but ignores dependency conflicts and backtracking [61].

To evaluate the extent to which this dependency solver can imitate `pip`, we collect packages with ≥ 1 non-optional direct dependency from the top 5000 most downloaded PyPI packages [58], resulting in 825 packages. For each package p, we use our solver to solve a dependency graph G_{ours} at the current time for its latest release. Then, we run `pip install` p in a clean virtual environment to get a ground truth dependency graph G_{pip} solved by `pip` and compute precision & recall as:

$$Precision(p) = |G_{ours} \cap G_{pip}| \ / \ |G_{ours}|$$
$$Recall(p) = |G_{ours} \cap G_{pip}| \ / \ |G_{pip}|$$

Among the 825 packages, we obtain an average precision of 0.9715 and an average recall of 0.9390, indicating a very high degree of match between the results of the two solvers. The mismatches can happen for various reasons, such as the four-month lag between our dump and the experiment time, `pip`'s backtracking behaviors [61], etc. Still, our custom dependency solver is orders of magnitude faster than `pip` because it directly queries our metadata dump (instead of interacting with PyPI APIs and downloading a lot of release files). It also supports resolving dependency graphs at any historical time of interest, which is not possible using `pip`. Using this solver, we compute a historical dependency graph for each release *at its upload time*, which we will use for our empirical study.[3]

B. PyPI Licensing Data

1) Data Collection: The licensing information of a release can be found in three possible data sources:

- The `license` field in its distribution metadata. It has two notable limitations: 1) its value is left to the discretion of individual developers without a uniform format (e.g., an Apache 2.0 licensed package can have values like `"Apache v2"`, `"Apache Version 2"`, `"Apache 2"`, or even the complete license text; 2) 31.9% of the releases do not have this field in its distribution metadata.
- The `classifier` field in the metadata may contain predefined license identifiers that can be easily mapped into SPDX identifiers [37]. This data source is validated by PyPI and can serve as ground truth, but even fewer (13.8%) releases have license tag(s) in `classifier`.
- The wheel distribution files, which can include `LICENSE` and `README` files with licensing information. However, downloading all distribution files from PyPI would require excessive computation and network resources.

To address the limitations of these data sources, we design a multi-step cross-validation approach to get cleaned licensing information (as SPDX license identifiers) for as many releases as possible in our dataset. For this purpose, we build a mapping between `license` fields and SPDX license identifiers using all package versions with available classifier tags. Using this mapping, we build another mapping between SPDX license

[2] Specifically, among the top 5000 most downloaded PyPI packages [58] (which we will also use for the empirical study, Section IV-B), only 0.28% of their releases have inconsistent `requires_dists` in different distributions.

[3] Note that this approach ignores development dependencies and optional dependencies, computing only the dependency graph that is always distributed with the package (e.g., when a `pip install` is executed). This means that any license incompatibilities, especially those related to redistribution, would be highly problematic if present in this dependency graph.

identifiers and common keywords in the `license` fields, including name keywords, version keywords, "must-not-have" keywords, and "must-have" keywords. The two mappings are intended to "cross-validate" `license` fields using the ground truth available from the license classifier tags.

For each release, if it already contains a license identifier in the `classifier` field, we just convert it to the SPDX identifier. Then, for each of the remaining releases with a `license` field, we retrieve the most frequent SPDX license identifier corresponding to the value of this field using the first mapping. If the above retrieval does not work, we use the second keyword mapping (which is looser) to map the `license` field into one of the SPDX license identifiers. If the previous steps fail, or if the release does not have a `license` field, we download its distribution file and scan the `LICENSE` and `README` files using `ScanCode` [35], a widely used license detection tool. Finally, if all attempts fail to resolve into an SPDX identifier, we mark the license as `Unrecognizable`.

By applying the above approach to the 3,622,711 releases in our dataset, we get licensing information from classifier tags, the `license` field, and distribution files for 500,457 (13.8%), 2,465,863 (68.1%), 135,590 (3.7%) releases respectively, leaving 520,801 (14.4%) releases with `Unrecognizable` licensing.

To evaluate the effectiveness of this license identification approach, we randomly sample 385 releases from the population of 3,622,711 releases (95% confidence level, 5% confidence interval [63]). Then, we manually check whether the licenses identified by our approach can match different sources of information, including 1) GitHub repositories), 2) `LICENSE` files in the distribution, and 3) the `license` field. Among the 385 samples, our approach returns `Unrecognizable` for 51 of them (13.2%). Among the remaining 334 samples, 323 match other sources of information, resulting in an accuracy of 96.7% (323 / 334). For the 11 misidentified samples, six are due to users providing incorrect licensing information in the metadata, four are because users omit the versions of their license in the `license` field, and one is due to dual licensing. Among the 51 samples with `Unrecognizable` licensing, ten have been removed from PyPI at the time of inspections, 39 do not have licensing information in all sources, and five are early releases of a package (developers may only consider licensing until official release [21]). Two samples have custom licenses that are not covered by existing license identifiers. Finally, for one sample, there is no sufficient information in both the `license` field and the `LICENSE` file to determine the specific license for the release. To summarize, the evaluation results demonstrate that our approach is able to identify licensing information in the majority of cases except when the data sources are noisy or dual licensing is used, but both cases are rare. We believe that the resulting licensing information can provide a sound foundation for subsequent analyses.

2) Finding License Incompatibilities: Inspired by previous works [44], [50], we consider the *one-way combinative incompatibility* between licenses in this paper, defined as:

Definition 1: (*License Incompatibility*) License A is one-way incompatible with license B if and only if it is infeasible to distribute derivative works of A-licensed software under B.

For example, GPL 3.0 is one-way incompatible with Apache 2.0 because the derivative works of GPL-3.0-licensed software cannot be distributed under Apache 2.0 (but the reverse is feasible, and thus *one-way* incompatible). On the other hand, Apache 2.0 and GPL 2.0 are incompatible in both ways because they have conflicting terms about patents [64].

This definition fits well in the context of packaging ecosystems because a package can be considered the derivative work of its dependencies (according to the Free Software Foundation (FSF) but there are some controversies [65]–[67]).

We compute all one-way incompatible license pairs using the license compatibility matrix proposed by Xu et al. [50], in which they analyzed the compatibility between licenses along 19 dimensions of terms such as copyleft, trademark grants, and patent grants. We choose this matrix for three reasons. First, it is the largest available license compatibility data to the best of our knowledge, compromising compatibility relationships between 63 licenses. Second, to ensure popularity and representativeness, all the licenses are: 1) certified by FSF or OSI [2]; 2) not obsolete (e.g., Apache-1.1); 3) not restricted to specific domains, software, or authors (e.g., IPA is a font license). Third, the 63 licenses can cover 99.4% of releases of which the license information has been obtained in our dataset.

Using these incompatible license pairs, we identify incompatible dependencies for each release based on the dependency graphs computed in Section III-A2.

C. Dataset Overview

To summarize, our dataset contains 438,967 PyPI packages and 3,622,711 releases from the entire PyPI ecosystem as of November 2022. For each release, the dataset offers 1) an SPDX license identifier, 2) a list of direct dependencies and their version constraints, 3) a dependency graph at its upload time, and 4) a list of incompatible dependencies. The dataset is stored as a MongoDB collection occupying 3.45GB of storage space with built-in compression. As most of the dataset construction process is automated (except building the keyword mapping in Section III-B1), the dataset can be easily updated using the latest PyPI BigQuery dataset. To the best of our knowledge, this is the *first* dataset of dependency and licensing information in the entire PyPI ecosystem. We will discuss the limitations of this dataset in Section VI-B.

IV. Empirical Study

A. Research Questions

The goal of this empirical study is to provide evidence about license incompatibilities and their remediation practices in the PyPI ecosystem. Such evidence can help the design of automated tools supporting remediation in dependency graphs. Toward this goal, we ask the following research questions:

- **RQ1:** *What is the distribution of licenses and how does licensing evolve in the PyPI ecosystem?*
  Rationale. This **RQ** aims to provide an overview of licenses and licensing evolution in the PyPI ecosystem. We are especially interested in the prevalence and evolution

of restrictive licenses as they are most likely to introduce license incompatibilities. Although the same question has been answered in other contexts [13], [20], it has not been answered in PyPI yet, motivating us to ask this **RQ**.

- **RQ2:** *What is the distribution of license incompatibilities in the dependency graphs of PyPI releases?*
  Rationale. Due to the prevalence of license incompatibilities in npm and RubyGem [12], [13], this **RQ** intends to confirm, in PyPI, the prevalence of license incompatibilities. We are also interested in their positions in the dependency graph (direct or transitive), and their degree of connectivity with other nodes in the dependency graph, which may indicate possible difficulties in remediation.
- **RQ3:** *How do PyPI package developers respond to and remediate license incompatibilities in practice?*
  Rationale. The goal of this **RQ** is to uncover the challenges that developers face when attempting to remediate license incompatibilities and to explore common remediation strategies discussed by developers. Such understanding is vital for the design of supportive tools, especially in the design of potential solution spaces.

B. Study Subjects

For **RQ1** & **RQ2**, we consider two groups of PyPI packages:

- TOP: The top 5000 most downloaded PyPI packages [58]. This group represents widely-used Python packages for which license incompatibilities can have a huge impact;
- ALL: All the 438,967 PyPI packages in our dataset.

We expect a comparison to reveal the differences between popular packages and the global population in terms of their license preferences and licensing practices. To avoid bias from packages with a large number of releases, we only select the latest release of each package in each year for all subsequent analyses (except for within-package evolution in **RQ1**).

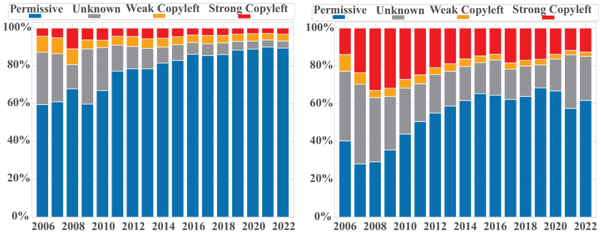
For **RQ3**, we only focus on the TOP group as they are more likely to have mature development practices and transparent development activities (e.g., extensively using issue trackers), without which the answering of **RQ3** would be impossible.

C. Methods and Results

1) RQ1: License Distribution & Evolution: Following prior work [44], [45], [49], we classify licenses into four different categories ordered by their level of permissiveness:

- **Permissive**: Software that changes or uses existing software can be licensed under a different license (e.g., MIT);
- **Weak Copyleft**: Software that changes existing software must be licensed under the same license, but software that uses existing software (e.g., by calling APIs) does not have to (e.g., LGPL 3.0).
- **Strong Copyleft**: Software that changes or uses existing software must be licensed under the same license unless an exception is specified (e.g., GPL 3.0 and AGPL 3.0);
- **Unknown**: The license is `Unrecognized` (Section III-B).

Overall, widely-used PyPI packages tend to be permissive: in the TOP group, 85.82% have a permissive license, 4.07% have a weak copyleft one, and 3.72% have a strong copyleft one, leaving 6.39% as unknown. However, the global population is more restrictive and less recognizable: in the ALL group, the ratio of packages with a permissive, weak left, strong copyleft license is 62.14%, 2.80%, and 14.67% respectively, leaving a large proportion of 20.39% as unknown.



(a) TOP (b) ALL

Fig. 2. The yearly distribution of licensing categories in the two groups.

We plot the yearly distribution of licensing categories in Figure 2(a) and 2(b). We can observe that permissive licenses are not only the most common but also increasingly popular over the years in both groups. However, as of 2022, packages with strong copyleft licenses in ALL still constitute a significant portion (12.63%) and 4.0x higher than that of TOP (3.17%). What's more, the proportion of the unknown category in TOP is lower than that in ALL and is decreasing over the years. This indicates that widely-used packages have devoted efforts to providing accurate and complete licensing information but less popular ones have not done so.

Similar to Vendome et al. [20], we investigate how licensing evolves *within packages*. We confirm that licensing changes are not uncommon in PyPI packages (just as other OSS [20], [38]): in the TOP group, 425 (9.10%) packages have undergone one licensing change, and 87 (1.86%) packages have undergone two or more changes. This is significantly higher than that in ALL (3.04%). Most licensing changes are between licenses in the same level of permissiveness (63.74% in TOP and 56.20% in ALL). In TOP, there is a tendency toward using more permissive licenses (27.66%) but changing toward less permissive ones is less frequent (8.60%). In ALL, licensing changes in both directions are common (26.15% toward more permissive and 17.65% toward less permissive).

Answers for RQ1: In the PyPI ecosystem, 85.82% of the TOP packages have a permissive license, but strong copyleft licenses are also present (3.72% among TOP and 14.67% among ALL). 10.96% of the TOP packages and 3.04% of ALL have undergone at least one licensing change. Although many licensing changes are within the same level of permissiveness, a non-negligible portion is toward more restrictive ones (17.65% among ALL).

Implications: The risk of license incompatibilities could be high in PyPI due to the presence of strong copyleft licenses. We also confirm that licensing changes are common in PyPI packages, among which changing toward more restrictive licenses could be especially problematic for the downstream packages. To take licensing changes into consideration, a precise and versioned dependency graph is necessary for license incompatibility analysis.

TABLE I
THE LICENSE COMPATIBILITY STATUS OF PYPI RELEASES

	Top (10,282 releases)		All (271,811 releases)	
Compatibility Label	Count	Percentage	Count	Percentage
`Compatible`	5,731	55.64%	114,135	41.99%
`Incompatible`	202	1.96%	19,772	7.27%
`Unknown`	4,349	42.30%	137,904	50.74%

TABLE II
THE CUMULATIVE DISTRIBUTION OF DEPENDENCY GRAPH METRICS FOR ALL INCOMPATIBLE DEPENDENCIES

		$= 0$	≤ 1	≤ 2	≤ 3	≤ 4	≤ 5
`Depth`	Top	-	74.0%	96.2%	100%	-	-
	All	-	38.7%	71.8%	89.1%	95.2%	97.6%
`In-degree`	Top	-	95.9%	100%	-	-	-
	All	-	75.4%	87.7%	91.9%	94.2%	95.8%
`Out-degree`	Top	60.8%	68.7%	81.9%	90.2%	97.4%	97.7%
	All	45.6%	57.9%	63.7%	68.6%	74.1%	79.0%

2) RQ2: License Incompatibility Distribution: In Section III-C, we have resolved a dependency graph for each release at its upload time and checked whether the licenses of all its dependencies in the graph are compatible with the license of this release. If any incompatibility is detected, we label the release as `Incompatible`; if all dependencies have compatible licenses, we label the release as `Compatible`; otherwise, (i.e., there is at least one dependency with `Unrecognizable` license), we label the release as `Unknown`.

As we study license incompatibilities introduced by dependencies, we exclude releases without dependencies, leaving 10,282 releases in the Top group (3,068 packages) and 271,811 releases in the All group (176,955 packages). We summarize their license compatibility status in Table I. We can observe that license incompatibilities are less common among Top, with only 202 (1.96%) of the releases being `Incompatible` (92 packages). However, this proportion is significantly higher among All, with 19,772 (7.27%) being `Incompatible`. This indicates that license incompatibilities are not uncommon in PyPI ecosystem and much more common (3.7x) in less popular packages than widely-used packages.

In the dependency graph of a release, license incompatibility can be caused by both direct dependencies and transitive dependencies. The latter is more difficult to remediate because: 1) transitive dependencies are required by other dependencies and developers have limited control over them; 2) their remediation can trigger a ripple effect due to edges in the graph. Therefore, to gain a better understanding of this problem, we are interested in the *location* of license incompatibilities in dependency graphs. For each license incompatibility, we compute the following metrics in the dependency graph:

- **Depth:** The shortest distance between the incompatible dependency and the root node. Direct dependencies have a depth of one. A high depth means a long dependency chain needs to be addressed during remediation.
- **In-degree:** The number of packages in the dependency graph directly depending on the incompatible dependency, which needs to meet the version constraints for all of them. In-degree characterizes the number of constraints that need to be considered during remediation.
- **Out-degree:** The number of packages that the incompatible dependency directly depends on. Out-degree characterizes the number of dependencies that could be impacted when remediating the compatibility issue.

Table II shows the cumulative distribution of these metrics for incompatible dependencies in the dependency graph. In total, there are 265 and 46,237 incompatible dependencies in the Top and All groups, respectively (a release may have multiple incompatible dependencies). We find that incompatible dependencies are more likely to be in a complex position among All compared with Top. Among Top, 26.0% of them come from transitive dependencies (i.e., depth $\geq$2) while the percentage rises to 61.3% among All. 5,032 (10.9%) of them in All have a depth of at least four in the dependency graph, and 5,681 (12.3%) of them have an in-degree greater than or equal to three. However, among Top, all cases of license incompatibilities caused by transitive dependencies are limited to the second or third layer of the dependency graph, with an in-degree of either one or two. Moreover, among All, the mean of the out-degree for incompatible dependencies in the dependency graph is 3.93, whereas in Top, it is only 1.06.

In other words, license incompatibilities are sophisticated for many releases in the PyPI ecosystem. They may be caused by incompatible transitive dependencies, some of which are deeply nested with many dependencies and dependents. This means that the remediation of these incompatibilities requires addressing many other interrelated dependencies, necessitating a method that can identify feasible solutions from a global perspective considering the entire dependency graph.

Answers for RQ2: In the entire PyPI ecosystem, a significant proportion of releases (7.27%) have license incompatibilities. Although most incompatible dependencies (74.0%) are direct dependencies in dependency graphs of Top packages, 61.3% of them in that of All are transitive dependencies that may reside in deep and sophisticated dependency graph positions.

Implications: License incompatibilities form a significant problem in the PyPI ecosystem. Remediating license incompatibilities in transitive dependencies requires searching for a feasible solution from a global perspective in the entire dependency graph.

3) RQ3: License Incompatibility Remediation in Practice: To answer **RQ3**, we analyze the GitHub issue trackers of the 92 packages with license incompatibilities from Top. For each package, we manually find their GitHub repository and search the issue tracker using three different keywords: 1) `license`; 2) the name of incompatible license (e.g., `GPL`); 3) the name of the incompatible package (e.g., `unidecode`). Then, we manually identify relevant issues, pull requests (PRs), and discussions from the search results, resulting in 25 issues and eight PRs from 17 repositories. For each repository, we find the developers' discussions and categorize the remediations (or proposed remediations) using an open-coding procedure [68]. To ensure reliability and avoid bias, two authors of this paper, both with over five years of software development experience,

TABLE III
LICENSE INCOMPATIBILITIES AND THEIR REMEDIATIONS. ★ MARKS THE FINAL REMEDIATION TAKEN BY DEVELOPERS.

	Package	License	Incompatible Dependency	License	Issue(s) & PR(s)	Proposed Remediation(s)
Identified in RQ3	`ansible-lint`	MIT	`ansible`	GPL 3.0	#1188, #1882	★Change Own License
	`apache-airflow`	Apache 2.0	`mysql-connector-python`	GPL 3.0	#9898, #10667	Migration, ★No Remediation
	`cvxpy`	Apache 2.0	`ecos`	GPL 3.0	#313	Migration, ★No Remediation
	`dvc`	Apache 2.0	`grandalf`	GPL 2.0	#1115	★No Remediation
	`fbprophet`	3-Clause BSD	`lunardate`	GPL 3.0	#1069, #1091	★Migration, Removal
	`fbprophet`	3-Clause BSD	`pystan`	GPL 3.0	#1045, #1221	★Migration
	`fiftyone`	Apache 2.0	`ndjson`	GPL 3.0	#2864, `eta`#590	★Migration
	`fiftyone`	Apache 2.0	`patool`	GPL 3.0	#2864, `eta`#590	★Migration
	`halo`	MIT	`cursor`	GPL 3.0	#118, #147	Pin Version, Migration, ★Removal
	`jiwer`	Apache 2.0	`levenshtein`	GPL 3.0	#69, #71	★Migration
	`mitmproxy`	MIT	`html2text`	GPL 3.0	#2572, #2573	★Removal
	`netcdf4`	MIT	`cftime`	GPL 3.0	#1000, #1073	★Negotiation, Pin Version
	`orbit-ml`	Apache 2.0	`pystan`	GPL 3.0	#435	Migration
	`pulp`	3-Clause BSD	`amply`	EPL 1.0	#394	★Negotiation, Removal
	`pytest-pylint`	MIT	`pylint`	GPL 2.0+	#178	No Remediation
	`textacy`	Apache 2.0	`fuzzywuzzy`	GPL 2.0	#62, #63	★Removal
	`textacy`	Apache 2.0	`unidecode`	GPL 2.0+	#203	Migration, ★Removal
	`textacy`	Apache 2.0	`python-levenshtein`	GPL 3.0	#203	Migration
	`wemake-python`	MIT	`flake8-isort`	GPL 2.0	#2481	Negotiation, ★Migration
	`workalendar`	MIT	`lunardate`	GPL 3.0	#346, #536, #709	Change Own License, ★Migration, Removal
	`yt-dlp`	Unlicense	`mutagen`	GPL 2.0+	#348, #2345	Change Own License, Removal
Reported by SILENCE	`amundsen`	Apache 2.0	`unidecode`	GPL 2.0+	#2148, #2168	Chg. Own Lic., ★Migration, Removal, Pin Ver.
	`cibuildwheel`	2-Clause BSD	`bashlex`	GPL 3.0	#1484	Change Own License, Removal
	`glean-parser`	MPL 2.0	`yamllint`	GPL 3.0	#1830049, #578	Change Own License, ★Removal
	`metaflow`	Apache 2.0	`pylint`	GPL 2.0+	#1377, #1378	Change Own License, Migration, ★Removal
	`music-assistant`	Apache 2.0	`unidecode`	GPL 2.0+	#1220	Change Own License, Migration, Removal
	`optbinning`	Apache 2.0	`ecos`	GPL 3.0+	#242	Change Own License, Removal
	`pylint-gitlab`	MIT	`pylint`	GPL 2.0+	#15, #20	★Change Own License, Migration, Removal
	`sphinx-autoapi`	MIT	`unidecode`	GPL 2.0+	#382, 0a557fc	Chg. Own Lic., ★Migration, Removal, Pin Ver.
	`zha-quirks`	Apache 2.0	`zigpy`	GPL 3.0	#3256	Change Own License, Removal

independently performed the above steps; they later discussed and merged the results into a consensus.

The upper half of Table III summarizes the 21 license incompatibilities we found and the remediations proposed or taken by developers. We have two immediate observations:

a) License incompatibilities happen because OSS developers lack knowledge or pay little attention to OSS licensing. For example, a developer commented: *I don't get into licensing much and hence MIT everything, thus don't know the implications of this. I will investigate this and get back.* (`halo`#118).

b) License incompatibilities frequently cause confusion and controversies, even among experienced OSS developers, after they are raised in an issue. Many issues in Table III triggered lengthy discussions about whether the incompatibility really exists and whether it really matters for their projects (e.g., `pulp`#394). For example, a common argument is that having a GPL-3.0-licensed dependency does not result in the package becoming a "derivative work" of that dependency. However, this contradicts the interpretation of FSF [66] and is disagreed by many other developers. The situation is more controversial and sophisticated in some cases, such as with the presence of optional dependencies (e.g., `apache-airflow`#9898).

In 17 of the 21 cases, developers acknowledged the relevance of license incompatibilities and the necessity of remediation. However, it can be non-trivial to find an appropriate remediation method and developers often need to evaluate multiple possibilities (as can be observed in Table III). Specifically, they considered the following remediation methods:

a) Migration (13 Incompatibilities): The most common remediation is to migrate the incompatible dependency to an alternative package with similar functionalities. For example, `lunardate` can be replaced with `LunarCalendar` and `unidecode` can be replaced with `text-unidecode`. This observation echoes prior research showing that developers migrate packages due to licensing issues [69], [70].

b) Removal (8 Incompatibilities): If the incompatible dependency is not used extensively, developers choose to remove the dependency and replace it with their own implementations of the desired functionality. For example, the developers of `halo` eventually decided, after lengthy discussions, to remove `cursor` and re-implement based on a Stack Overflow snippet.

c) Change Own License (3 Incompatibilities): Some developers proposed changing their package's own license to comply with the licensing requirement of its dependency. This remediation was finally taken by `ansible-lint` as it is closely integrated with its GPL-licensed dependency, `ansible`.

d) Negotiation (3 Incompatibilities): Another feasible option is to ask upstream developers (i.e., developers of the incompatible dependency) to change the licenses of their packages toward more permissive ones. For example, `cftime` decided to remove GPL-related code and relicense itself under MIT after a request from `netcdf4` developers (`cftime`#116).

e) Pin Version (2 Incompatibilities): In the case of `cursor` and `cftime`, the two packages were initially released under a permissive license but changed their license in a new release. To remediate this, developers of `halo` and `netcdf4` proposed to pin their versions to the version before the license change.

In three cases, developers conclude that remediation is not necessary because the incompatible dependency is optional (`apache-airflow`, `cvxpy`) or the dependency provides a dual-licensing option (`dvc`). In the case of `pytest-pylint`, developers questioned the necessity of remediation, but the issue is still open and unresolved at the time of writing.

Answers for RQ3: PyPI package developers show unfamiliarity and raise controversies with OSS licensing when they discover a license incompatibility. They remediate license incompatibilities by 1) migrating, removing, or pinning a version of the incompatible dependency; 2) changing their own licenses; or 3) asking upstream developers to change the licensing of their package.

Implications: Automated approaches can be helpful in making developers aware of license incompatibilities and recommending remediations. The practices taken by developers can serve as the solution space to be explored by automated approaches.

V. The Silence Approach

Inspired by the results from the empirical study, we propose Silence, an SMT-solver-based incompatibility remediator for licenses in the dependency graph. In this section, we describe the design, implementation, and evaluation of Silence.

A. Data and Notations

Recall in Section III-C that our dataset contains 438,967 packages and 3,622,711 releases from the entire PyPI ecosystem. To simplify the presentation of Silence, we provide a formal notation of this dataset. We denote the set of package names as $\mathcal{P}$, the set of version strings as $\mathcal{V}$, and the releases in our dataset (i.e., the entire PyPI ecosystem) as $\mathcal{E} \subseteq \mathcal{P} \times \mathcal{V}$ ($|\mathcal{E}| = 3,622,711$). Each $\langle p, v \rangle \in \mathcal{E}$ contains:

- An SPDX license identifier $l(p, v)$.
- Direct dependencies and version constraints $deps(p, v) \subseteq \mathcal{P} \times \mathcal{C}$ ($\mathcal{C} \subseteq \mathcal{V}^*$ denotes the set of version constraints).
- A dependency graph $\mathcal{G}(p, v) ::= \langle N(p, v), D(p, v) \rangle$, s.t. $\langle p, v \rangle \in N(p, v) \subseteq \mathcal{E}$, $D(p, v) \in N(p, v) \mapsto N^*(p, v)$.
- A list of incompatible dependencies $incomp(p, v) \subseteq N(p, v)$ in the dependency graph, such that $\langle p', v' \rangle \in incomp(p, v) \Rightarrow \langle l(p', v'), l(p, v) \rangle \in \mathcal{I}$ (here $\mathcal{I}$ denotes the set of one-way incompatible license pairs).

We denote the set of 63 licenses in the compatibility matrix as $\mathcal{L}$. To support finding migrations, we use the Python package migration dataset by Gu et al. [29] containing 640 migration rules between Python packages, denoted as $\mathcal{M} \subseteq \mathcal{P} \times \mathcal{P}$.

B. Problem Formulation

According to our **RQ3**, developers may take one of the following approaches to remediate license incompatibilities: migration, removal, pinning version, changing their own license, and negotiating with upstream packages. The results inspire us with the idea of using an automated approach to generate and recommend possible remediations to developers when a license incompatibility is detected (the detection can be easily automated using our PyPI dependency and licensing dataset in Section III). Such an automated approach can be implemented as a GitHub CI/CD Action or a bot deployed to notify and help developers remediate licensing incompatibilities. As developers frequently discuss several remediations in their issues and choose one of them eventually, this automated approach should be able to recommend multiple reasonable remediations for developers to consider and choose.

For the possible remediations, negotiations fall out of scope for an automation tool, and determining which license(s) can be changed is trivial as it only requires an enumeration of all alternative licenses while assessing their compatibility with the package dependency graph. However, finding migration, removal, and version-pinning solutions is more challenging because incompatible dependencies may reside in a sophisticated dependency graph position and any change can have a ripple effect over the entire graph. On the other hand, developers generally want to minimize changes to their dependency graph because larger changes would often result in more remediation effort. What's more, finding viable migration targets itself is challenging and has been explored in prior research [69], [71].

Considering the above rationales, we define the license incompatibility remediation problem as follows:

1) **Input:** a release $\langle p, v \rangle$, its dependency graph $\mathcal{G}(p, v)$, and the PyPI dataset (Section V-A);
2) **Output:** N alternative dependency graphs $\mathcal{G}'_1, ..., \mathcal{G}'_N$, all of which have no license incompatibility and minimal changes to $\mathcal{G}(p, v)$, and M alternative licenses $l_1, ..., l_M$ with which $\langle p, v \rangle$ would have no license incompatibility in $\mathcal{G}(p, v)$.

We observe that this definition is similar to the dependency resolution problem studied in prior work [72]–[74] with some important differences. The alternative dependency graphs can ignore dependencies (for removals), violate version constraints (for pinning versions), and add new direct dependencies (for migrations). Nonetheless, any deviations from the original graph need to be minimized. Just like the dependency resolution problem, such alternative dependency graphs can be found using a Max-SMT solver with a carefully designed objective function. The exact remediations can be generated by comparing the alternative graph and the original graph.

C. Approach Overview

```
Algorithm 1: The Silence Approach
   Input: $\langle p, v \rangle$, $\mathcal{G}(p, v)$, and the PyPI dataset (Section V-A)
   Output: $\mathbf{G} = \{\mathcal{G}'_1, ..., \mathcal{G}'_N\}, \mathbf{L} = \{l_1, ..., l_M\}$
1  $\mathbf{G} \leftarrow \emptyset$, $\mathbf{L} \leftarrow \emptyset$
2  foreach $l \in \mathcal{L}$ do  # find compatible licenses
3      if $\langle l(p', v'), l \rangle \notin \mathcal{I}$ for all $\langle p', v' \rangle \in N(p, v) \setminus \langle p, v \rangle$ then
4          $\mathbf{L} \leftarrow \mathbf{L} \cup \{l\}$
5  Keep only top-$M$ licenses in $\mathbf{L}$ ordered by their popularity
6  $vars \leftarrow \{p$, plus all packages reachable from $deps(p, v)\}$
7  $clauses \leftarrow build\_constraints(p, v, vars)$
8  while $\mathcal{G}' \leftarrow find\_solution(vars, clauses, objective)$ do
9      if $|\mathbf{G}| \geq N$ or $\mathcal{G}' = unsat$ then break
10     $\mathbf{G} \leftarrow \mathbf{G} \cup \{\mathcal{G}'\}$
11     Add new constraints to exclude solutions similar to $\mathcal{G}'$
12 return $\mathbf{G}, \mathbf{L}$
```

Algorithm 1 summarizes the Silence approach. In line 2-5, it finds M compatible licenses. In line 6-10, it finds N

alternative dependency graphs without license incompatibilities. The key idea is to find all packages that may be present in the alternative graph (line 6), build version constraint clauses for each package (line 7), and find top-N solutions under the $objective$ function using a Max-SMT solver (line 8-11). We will describe the underlying details in Section V-D.

D. SMT-Solver-Based License Incompatibility Remediation

To create a constraint SMT problem over a finite domain, the first step is to initialize a set of finite domain variables for all packages that may be present in the alternative graphs (i.e., $vars$ in line 6). To find these packages, we utilize a breadth-first search (BFS) beginning from the root package p and all possible migration targets p_m that may replace one of the dependencies of p (i.e., $\exists\langle p_d, C\rangle \in deps(p, v)$, s.t. $\langle p_d, p_m\rangle \in \mathcal{M}$). For each package p' in the BFS queue, we encode all its versions $v_1, ..., v_k$ in a finite integer domain, ordered by semantic versioning [75], from $-k$ to -1 (i.e., oldest to latest). We use the special value $p' = 0$ to indicate p' is not included in the graph. All packages that p' may depend on (i.e., packages in $\bigcup_{i=1,...,k} deps(p', v_i)$) will be added to the BFS queue. The search stops once saturation is reached (i.e., no more new packages could be added to $vars$).

With a set of finite domain variables $vars$, the next step is to encode their dependency relationships and version constraints as $clauses$ (line 7). For each $p' \in vars$, excluding the root package p, we encode a logical implication as follows:

$$(p' = v') \implies \bigwedge_{\langle p_d, C\rangle \in deps(p', v')} \left(\bigvee_{v_d \in C} (p_d = v_d) \right)$$

Here we use $p = v$ as a convenience notation meaning that the corresponding finite domain variable of p in $vars$ takes the concrete integer value corresponding to v.

For root package p, we need to encode possible remediations (i.e., migrations, removals, and version pinning) into its clause, all of which can result in violations of $dep(p, v)$. To consider this, we add all possible migration targets without version constraints (i.e., $\{\langle p_m, \mathcal{V}\rangle\}$) to $deps(p, v)$, allow each p_d in $deps(p, v)$ to be removed (i.e., $p_d = 0$), and allow the version constraints to be violated, forming the following clause:

$$\bigwedge_{\langle p_d, C\rangle \in deps(p,v) \cup \{\langle p_m, \mathcal{V}\rangle\}} \left((p_d = 0) \vee \bigvee_{\langle p_d, v_d\rangle \in \mathcal{E}} (p_d = v_d) \right)$$

Of course, p must be of its original version (i.e., $p = v$).

Finally, to remediate license incompatibilities, we add the following logical implications for all packages in $vars$:

$$\bigwedge_{p' \in vars} (\langle l(p', v'), l(p, v)\rangle \in \mathcal{I} \implies p' \neq v')$$

All the above $clauses$ form the constraints of this problem. For the packages in $vars$, any set of concrete integer values satisfying all the constraints forms a valid solution. However, the constraints here are loose with many possible solutions. To find solutions (i.e., an alternative graph $\mathcal{G}'$) with minimal differences compared with the original graph $\mathcal{G}(p, v)$, we define the optimization $objective$ (in line 8) as follows:

$$\min_{\mathcal{G}'} \sum_{\langle p_{\text{old}}, p_{\text{new}}\rangle \in \text{diff}(\mathcal{G}, \mathcal{G}')} \begin{cases} c_{\text{migration}}, & \langle p_{\text{old}}, p_{\text{new}}\rangle \in \mathcal{M} \\ c_{\text{removal}}, & p_{\text{new}} = 0 \\ |p_{\text{new}} - p_{\text{old}}|, & \text{otherwise} \end{cases}$$

Specifically, this objective function attempts to find a $\mathcal{G}'$ that minimizes the total cost of all changed packages by comparing $\mathcal{G}'$ with $\mathcal{G}(p, v)$. For each changed package, the cost depends on what has been changed: if there is a migration between two packages, we add a constant cost $c_{\text{migration}}$; if a package is removed, we add a larger constant cost c_{removal}; if the version is changed within the same package, we add a cost equal to the distance between the changed versions (i.e., $|p_{new} - p_{old}|$). The two constant costs can be adjusted in practice. Using this objective function, line 8-11 finds the top-N solutions (ordered by the cost determined by $objective$) as the alternative graphs, all of which do not contain license incompatibilities. To avoid generating redundant solutions, we add a new constraint to $clauses$ to exclude all solutions similar to the current solution:

$$\bigvee_{\langle p_{\text{old}}, p_{\text{new}}\rangle \in \text{diff}(\mathcal{G}, \mathcal{G}')} \begin{cases} p_{\text{new}} = 0, & p_{\texttt{new}} \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

This means the new solution must not include all the changed packages in the previous solution. The algorithm stops if the solver returns $unsat$ or it has found N viable solutions.

E. Implementation

We implement SILENCE in Python using the Python binding of Z3 [76], the state-of-the-art SMT solver. To find the versions satisfying version constraints, we simply use the Python standard library `packaging` which implements PEP 440 [77]. We also implement an additional post-processing step to convert the results of Algorithm 1 into a remediation report like:

```
Possible Remediations for [package] [version]:
1. Change project license to $l_1$, $l_2$, ..., or $l_M$;
2. ($\mathcal{G}'_1$) Migrate [package] to [package];
3. ($\mathcal{G}'_2$) Remove [package];
4. ($\mathcal{G}'_3$) Pin [package] to [version];
```

In current implementation, we heuristically set $N = 5$, $M = 3$, $c_{\text{migration}} = 10$, and $c_{\text{removal}} = 100$, which we find to produce satisfactory results (see Section V-F). We tested SILENCE on the 202 incompatible releases in TOP (Table I). SILENCE can generate results for all of them with a median running time of 14.9 seconds (max = 295 seconds), which is satisfactory in practical application scenarios (e.g., as a CI/CD workflow).

F. Evaluation

We evaluate the effectiveness of SILENCE by observing to what extent can the remediations provided by SILENCE match those *proposed* by developers in the upper half of Table III. We do not compare against the final remediations because SILENCE is intended to provide recommendations and support the decision-making process. We observe that the final remediation is contingent upon multiple factors from the specific project context (e.g., the development cost of each remediation), so the decision should be left to project developers.

Of the 21 cases in upper Table III, developers proposed at least one remediation in 19 cases, except for `pytest-pylint` and `dvc`. We find that the results returned by Silence can cover all the proposed removals, version-pinnings, and license change remediations in these cases. However, due to the incompleteness of the Python migration dataset [29], Silence can only cover two out of the 13 migration proposals in these cases (`mysqlclient` to `PyMySQL` and `unidecode` to `text-unidecode`). For the remaining 11 migration proposals, Silence simply proposes to remove the incompatible dependency, leaving developers to find migrations themselves. This limitation can be easily overcome by adding more migration rules to $\mathcal{M}$ once they are discovered. Based on this evaluation, we conclude that Silence performs relatively well in the remediation of license incompatibilities for Python packages.

G. Example

In this section, we use the example of `fiftyone 0.18.0` in Figure 1 to illustrate how Silence can be applied to practice. For `fiftyone 0.18.0`, Silence provides the following remediation report for the existing license incompatibilities:

```
Possible Remediations for fiftyone 0.18.0:
1. Change project license to GPL-3.0-only,
    GPL-3.0-or-later, or AGPL-3.0-only;
2. Or make the following dependency changes:
    a) Remove ndjson;
    b) Pin voxel51-eta to 0.1.9;
    c) Pin pillow to 6.2.2;
    d) Pin imageio to 2.9.0;
    e) Pin h11 to 0.11.0.
3. Or make the following dependency changes:
    a) Remove voxel51-eta;
    b) Remove ndjson;
    c) Pin h11 to 0.11.0.
```

This report includes changes to `pillow` and `imageio` due to the ripple effect of pinning `voxel51-eta`. The change to `h11` is included to fix dependency conflicts in the previously resolved dependency tree, a positive side effect similar to SMT-solver-based dependency resolution like SmartPip [60].

As shown in Table III, the developers of `fiftyone` finally migrate `ndjson` to `jsonlines`. As mentioned in Section V-F, this migration is not covered by an existing dataset [29]. By adding $\langle \texttt{ndjson}, \texttt{jsonlines} \rangle$ to $\mathcal{M}$, Silence returns:

```
Possible Remediations for fiftyone 0.18.0:
1. Change project license to GPL-3.0-only,
    GPL-3.0-or-later, or AGPL-3.0-only;
2. Or make the following dependency changes:
    a) Migrate ndjson to jsonlines;
    b) Pin voxel51-eta version to 0.1.9;
    c) Pin pillow to 6.2.2;
    d) Pin h11 to 0.11.0;
    e) Pin imageio to 2.9.0.
3. Or make the following dependency changes:
    a) Migrate ndjson to jsonlines;
    b) Remove voxel51-eta;
    c) Pin h11 to 0.11.0.
```

With the above report, developers may conclude that `ndjson` should be migrated to `jsonlines`. Although the report points out that removal or downgrading `voxel51-eta` is necessary for remediating `patool`, developers may find such remediation undesirable because `voxel51-eta` is tightly integrated with `fiftyone`. In fact, they are developed under the same GitHub organization `voxel51`. In such cases, *the dependency changes must be made upstream*. The developers of `fiftyone` may then begin to negotiate with the developers of `voxel51-eta`, who can use Silence to produce a report for themselves:

```
Possible Remediations for voxel51-eta 0.8.1:
1. Change project license to GPL-3.0-only,
    GPL-3.0-or-later, or AGPL-3.0-only;
2. Or make the following dependency changes:
   a) Migrate ndjson to jsonlines;
   b) Migrate patool to py7zr.
3. Or make the following dependency changes:
   a) Migrate ndjson to jsonlines;
   b) Migrate patool to rarfile.
... (omitted due to space limitations)
```

H. Preliminary User Study

To evaluate how developers perceive the usefulness of Silence, we carefully select packages from Top that: 1) have incompatible releases in our dataset; 2) still have incompatibilities in their latest releases; 3) actively use an issue tracker; 4) have no previous issues about licensing. This results in ten packages. After manual inspection, we exclude one false positive, `dvc`, which is not actually incompatible with its GPL-licensed dependency `pygit2` due to its explicit statement of link exception [78]. We then open nine issues with the report by Silence, summarized in the lower half of Table III.

At the time of writing (August 2023), we received responses in seven issues, among which five packages have completely or partially adopted one of the remediations suggested by Silence. Notably, `glean-parser` subsequently implemented license checking in its CI/CD workflow (#578), indicating the need for and usefulness of integrating tools like Silence into the development process. `sphinx-autoapi` accepted the migration suggestion but migrated to another package not recommended by Silence. The remaining two packages, however, closed our issue. One package responded that although they acknowledge this incompatibility, they will only fix it if it actually causes issues to end users (which they believe is unlikely because their package is a CI tool, not a library).

In conclusion, five of the seven responded packages adopted one of the suggestions provided by Silence. The high adoption rate signifies the relevance of license incompatibilities to PyPI developers, their positive attitude towards Silence, and the effectiveness of Silence in addressing incompatibilities.

VI. Discussion

A. Implications

In this section, we discuss the implications of our results for developers, package distribution platforms, and researchers.

1) Developers: The results of **RQ1** show the prevalence of packages without accurate or complete licensing information in the PyPI ecosystem. However, if a package lacks licensing information, it is not really open-source [79], posing difficulties for others to legally use this package. Hence, developers should pay meticulous attention to the licensing of their dependencies and provide precise licensing information for their own packages to the best of their abilities. Additionally, 10.96% of the Top packages have undergone at least one licensing change as revealed in **RQ1**, which may impact numerous downstream projects and lead to incompatibilities. Therefore, developers of popular and influential packages should exercise more caution than those of common projects when making decisions regarding licensing changes. Finally, **RQ2** reveals that most of the license incompatibilities in the PyPI ecosystem are caused by direct dependencies (74.0%). These incompatibilities can be easily detected by parsing dependency manifest files and license checking can be integrated into CI/CD workflow, as evidenced by our preliminary user study. However, an accurate dependency graph like in this paper is needed to thoroughly detect license incompatibilities.

2) Package Distribution Platforms: In Section III-B, we find that the license information of a large number of PyPI packages on the PyPI platform is missing and the `license` field in the metadata does not have a uniform format, leading to the difficulty of identifying package's license. Therefore, package management platforms can enhance their management in this aspect by providing standardized options and requiring developers to provide accurate license information when uploading packages. Moreover, the platform can also perform license compatibility checks periodically, e.g., during the package uploading process, to ensure that the uploaded packages are compliant with licensing requirements.

3) Researchers: Our study sheds light on further research regarding license incompatibility. First, migration is the most common license incompatibility remediation practice (**RQ3**). Therefore, researchers can explore more accurate package migration recommendation techniques and build more comprehensive package migration datasets to help developers make more informed decisions. Second, we find that the licensing information declared by package developers is noisy. Therefore, better license detection techniques can be developed to capture these packages' licensing information in the future. Finally, our study also lays a foundation for further research on the license incompatibility remediation practices and automated solutions in other packaging ecosystems like NPM.

B. Limitations

We discuss some notable limitations of our dataset, the empirical study, and the SILENCE approach, as follows.

In terms of the PyPI dependency dataset, its main limitation is that the resolved dependency graphs at time t may differ from the actual dependency graphs resolved by popular tools like `pip` or `Poetry` at the same time. However, since each of them uses different resolution algorithms and may change its algorithms in new versions (e.g., `pip` implements backtracking since version 20.3 [61]), we believe accurate historical replication is impossible. Compared with using `pip install`, our custom solver is orders of magnitude faster and able to resolve dependency graphs at arbitrary time points. Despite possible deviations, we believe this approach is the most suitable for such large-scale studies (e.g., a similar approach is also used by Liu et al. [14] for studying security vulnerabilities in npm).

Several limitations pertain to the PyPI licensing data. First, this dataset does not consider dual licensing, multi-licensing, or license exceptions used by some OSS [39], [80]. Although our manual evaluation shows that they are rare in PyPI (Section III-B1), they may occasionally introduce false incompatibilities in the dataset. Future work is needed to take these corner cases into consideration. Second, our study ignores in-code licenses, which may also have incompatibilities with the package-wide license [45]. However, studying such incompatibilities would require a different methodology and is out of the scope of our study. Finally, none of the authors are law professionals and the dataset may contain inaccurate license incompatibilities. To alleviate this threat, we have tried our best to base our study on some sort of "joint consensus" among OSS developers, as reflected by reliable sources of information (e.g., OSI, FSF, and prior research). Even if some of the data are proven to be incorrect, we believe the methodology and the SILENCE approach presented in this paper are general and can be easily adapted to any new compatibility criterion.

In terms of external validity, the dataset and its construction process are largely unique and designed for PyPI, a flourishing packaging ecosystem of great importance in many application domains (e.g., AI). However, future work is needed for other packaging ecosystems, as they have different dependency resolution behaviors [81] and licensing data format. The remediation practices in **RQ3** are identified from a small number of popular Python packages, but we believe the general pattern should be applicable to proprietary Python projects and even projects in other ecosystems (future work is needed to validate our belief). The SILENCE approach is also general and can be extended to other packaging ecosystems by taking their unique dependency resolution behaviors into consideration [73].

VII. CONCLUSION

In this paper, we contribute 1) a PyPI dependency & licensing dataset, 2) a large-scale study of license incompatibilities and their remediation practices in the PyPI, and 3) an SMT-solver-based remediation approach, SILENCE. As packaging ecosystems are likely to grow more complex [10], we believe our contributions form a valuable reference for those willing to improve the state of OSS licensing compliance in modern packaging ecosystems. In the future, we plan to integrate our license incompatibility detection and remediation tool into CI/CD tools, e.g., GitHub workflow.

VIII. DATA AVAILABILITY

We provide a replication package at:

https://github.com/osslab-pku/SILENCE

ACKNOWLEDGMENT

This work is sponsored by the National Natural Science Foundation of China 61825201 and 62332001. We would like to extend our appreciation to Chao Wang and Xin Wu for their invaluable insights on license compliance.

References

[1] S. Xu, Y. Gao, L. Fan, Z. Liu, Y. Liu, and H. Ji, "LiDetector: License incompatibility detection for open source software," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 22:1–22:28, 2023.

[2] (2023, April) OSI approved licenses. [Online]. Available: https://opensource.org/licenses/

[3] (2023, April) GNU general public license version 3. [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.html

[4] (2023, April) The MIT license. [Online]. Available: https://opensource.org/license/mit/

[5] L. Rosen, "Open source licensing," *Software Freedom and Intellectual Property Law*, 2005.

[6] M. Sojer, O. Alexy, S. Kleinknecht, and J. Henkel, "Understanding the drivers of unethical programming behavior: The inappropriate reuse of internet-accessible code," *J. Manag. Inf. Syst.*, vol. 31, no. 3, pp. 287–325, 2014.

[7] (2022, Augest) PyPI. [Online]. Available: https://pypi.org/

[8] (2022, Augest) Maven. [Online]. Available: https://mvnrepository.com/

[9] (2022, Augest) npm. [Online]. Available: https://www.npmjs.com/

[10] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empir. Softw. Eng.*, vol. 24, no. 1, pp. 381–416, 2019. [Online]. Available: https://doi.org/10.1007/s10664-017-9589-y

[11] (2023, May) Glossary — Python packaging user guide. [Online]. Available: https://packaging.python.org/en/latest/glossary/

[12] S. Qiu, D. M. Germán, and K. Inoue, "Empirical study on dependency-related license violation in the JavaScript package ecosystem," *J. Inf. Process.*, vol. 29, pp. 296–304, 2021. [Online]. Available: https://doi.org/10.2197/ipsjjip.29.296

[13] I. S. Makari, A. Zerouali, and C. D. Roover, "Prevalence and evolution of license violations in npm and RubyGems dependency networks," in *Reuse and Software Quality - 20th International Conference on Software and Systems Reuse, ICSR 2022, Montpellier, France, June 15-17, 2022, Proceedings*, ser. Lecture Notes in Computer Science, vol. 13297. Springer, 2022, pp. 85–100.

[14] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 672–684.

[15] (2023, April) pip. [Online]. Available: https://pip.pypa.io/en/stable/

[16] (2023, April) Poetry. [Online]. Available: https://python-poetry.org/

[17] (2023, April) Frequently asked questions about the GNU licenses. [Online]. Available: https://www.gnu.org/licenses/gpl-faq.html

[18] M. Sojer and J. Henkel, "License risks from ad hoc reuse of code from the internet," *Commun. ACM*, vol. 54, no. 12, pp. 74–81, 2011. [Online]. Available: https://doi.org/10.1145/2043174.2043193

[19] D. A. Almeida, G. C. Murphy, G. Wilson, and M. Hoye, "Investigating whether and how software developers understand open source software licensing," *Empir. Softw. Eng.*, vol. 24, no. 1, pp. 211–239, 2019.

[20] C. Vendome, M. L. Vásquez, G. Bavota, M. D. Penta, D. M. Germán, and D. Poshyvanyk, "License usage and changes: A large-scale study of Java projects on GitHub," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*. IEEE Computer Society, 2015, pp. 218–228.

[21] C. Vendome, M. Linares-Vásquez, G. Bavota, M. Di Penta, D. M. German, and D. Poshyvanyk, "When and why developers adopt and change software licenses," in *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*. IEEE Computer Society, 2015, pp. 31–40. [Online]. Available: https://doi.org/10.1109/ICSM.2015.7332449

[22] I. Pashchenko, D. L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM, 2020, pp. 1513–1531. [Online]. Available: https://doi.org/10.1145/3372297.3417232

[23] (2023, April) LicenseCheck. [Online]. Available: https://github.com/FHPythonUtils/LicenseCheck

[24] (2023, April) LicenseFinder. [Online]. Available: https://github.com/pivotal/LicenseFinder

[25] D. M. Germán and A. E. Hassan, "License integration patterns: Addressing license mismatches in component-based development," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, pp. 188–198. [Online]. Available: https://doi.org/10.1109/ICSE.2009.5070520

[26] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2017, pp. 2169–2185.

[27] S. van der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. Germán, and A. Hemel, "Tracing software build processes to uncover license compliance inconsistencies," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. ACM, 2014, pp. 731–742. [Online]. Available: https://doi.org/10.1145/2642937.2643013

[28] D. M. Germán and M. D. Penta, "A method for open source license compliance of Java applications," *IEEE Softw.*, vol. 29, no. 3, pp. 58–63, 2012. [Online]. Available: https://doi.org/10.1109/MS.2012.50

[29] H. Gu, H. He, and M. Zhou, "Self-admitted library migrations in Java, JavaScript, and Python packaging ecosystems: A comparative study," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023*. IEEE, 2023, pp. 627–638. [Online]. Available: https://doi.org/10.1109/SANER56733.2023.00064

[30] R. Gobeille, "The FOSSology project," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings*. ACM, 2008, pp. 47–50. [Online]. Available: https://doi.org/10.1145/1370750.1370763

[31] T. Tuunanen, J. Koskinen, and T. Kärkkäinen, "Automated software license analysis," *Autom. Softw. Eng.*, vol. 16, no. 3-4, pp. 455–490, 2009. [Online]. Available: https://doi.org/10.1007/s10515-009-0054-z

[32] D. M. Germán, Y. Manabe, and K. Inoue, "A sentence-matching method for automatic license identification of source code files," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. ACM, 2010, pp. 437–446. [Online]. Available: https://doi.org/10.1145/1858996.1859088

[33] M. D. Penta, D. M. Germán, and G. Antoniol, "Identifying licensing of jar archives using a code-search approach," in *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*. IEEE Computer Society, 2010, pp. 151–160. [Online]. Available: https://doi.org/10.1109/MSR.2010.5463282

[34] X. Liu, L. Huang, J. Ge, and V. Ng, "Predicting licenses for changed source code," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 686–697.

[35] (2023, April) ScanCode. [Online]. Available: https://github.com/nexB/scancode-toolkit

[36] (2023, April) Licensee. [Online]. Available: https://github.com/licensee/licensee

[37] (2023, April) SPDX license list. [Online]. Available: https://spdx.org/licenses/

[38] M. D. Penta, D. M. Germán, Y. Guéhéneuc, and G. Antoniol, "An exploratory study of the evolution of software licensing," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 2010, pp. 145–154. [Online]. Available: https://doi.org/10.1145/1806799.1806824

[39] S. Comino and F. M. Manenti, "Dual licensing in open source software markets," *Inf. Econ. Policy*, vol. 23, no. 3-4, pp. 234–242, 2011. [Online]. Available: https://doi.org/10.1016/j.infoecopol.2011.07.001

[40] R. M. Meloca, G. Pinto, L. Baiser, M. Mattos, I. Polato, I. S. Wiese, and D. M. Germán, "Understanding the usage, impact, and adoption of non-OSI approved licenses," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 2018, pp. 270–280.

[41] J. P. Moraes, I. Polato, I. Wiese, F. Saraiva, and G. Pinto, "From one to hundreds: Multi-licensing in the javascript ecosystem," *Empir. Softw. Eng.*, vol. 26, no. 3, p. 39, 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09936-2

[42] C. Vendome, D. M. Germán, M. D. Penta, G. Bavota, M. L. Vásquez, and D. Poshyvanyk, "To distribute or not to distribute?: Why licensing bugs matter," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 2018, pp. 268–279.

[43] D. M. Germán, M. D. Penta, and J. Davies, "Understanding and auditing the licensing of open source software distributions," in *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010*. IEEE Computer Society, 2010, pp. 84–93. [Online]. Available: https://doi.org/10.1109/ICPC.2010.48

[44] G. M. Kapitsaki, F. Kramer, and N. D. Tselikas, "Automating the license compatibility process in open source software with SPDX," *J. Syst. Softw.*, vol. 131, pp. 386–401, 2017.

[45] T. Wolter, A. Barcomb, D. Riehle, and N. Harutyunyan, "Open source license inconsistencies on GitHub," *ACM Trans. Softw. Eng. Methodol.*, dec 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3571852

[46] R. Pfeiffer, "License incompatibilities in software ecosystems," *CoRR*, vol. abs/2203.01634, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2203.01634

[47] T. F. Gordon, "Analyzing open source license compatibility issues with Carneades," in *The 13th International Conference on Artificial Intelligence and Law, Proceedings of the Conference, June 6-10, 2011, Pittsburgh, PA, USA*. ACM, 2011, pp. 51–55. [Online]. Available: https://doi.org/10.1145/2018358.2018364

[48] M. Papoutsoglou, G. M. Kapitsaki, D. M. Germán, and L. Angelis, "An analysis of open source software licensing questions in stack exchange sites," *J. Syst. Softw.*, vol. 183, p. 111113, 2022.

[49] G. M. Kapitsaki and G. Charalambous, "Modeling and recommending open source licenses with findOSSLicense," *IEEE Trans. Software Eng.*, vol. 47, no. 5, pp. 919–935, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2909021

[50] W. Xu, X. Wu, R. He, and M. Zhou, "LicenseRec: Knowledge based open source license recommendation for OSS projects," in *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 180–183. [Online]. Available: https://doi.org/10.1109/ICSE-Companion58688.2023.00050

[51] (2023, April) Choose an open-source license. [Online]. Available: https://choosealicense.com/

[52] K. J. Stewart, A. P. Ammeter, and L. M. Maruping, "Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects," *Inf. Syst. Res.*, vol. 17, no. 2, pp. 126–144, 2006. [Online]. Available: https://doi.org/10.1287/isre.1060.0082

[53] R. Sen, C. Subramaniam, and M. L. Nelson, "Determinants of the choice of open source software license," *J. Manag. Inf. Syst.*, vol. 25, no. 3, pp. 207–240, 2009. [Online]. Available: https://doi.org/10.2753/mis0742-1222250306

[54] M. Sojer and J. Henkel, "Code reuse in open source software development: Quantitative evidence, drivers, and impediments," *J. Assoc. Inf. Syst.*, vol. 11, no. 12, p. 2, 2010. [Online]. Available: https://doi.org/10.17705/1jais.00248

[55] (2023, April) Module counts. [Online]. Available: http://www.modulecounts.com/

[56] (2023, April) PyPI BigQuery dataset. [Online]. Available: https://warehouse.pypa.io/api-reference/bigquery-datasets.html

[57] (2023, April) PEP 508 - Dependency specification for Python software packages. [Online]. Available: https://peps.python.org/pep-0508/

[58] (2023, April) Top PyPI packages. [Online]. Available: https://hugovk.github.io/top-pypi-packages/

[59] (2023, May) Core metadata specifications — Python packaging user guide. [Online]. Available: https://packaging.python.org/en/latest/specifications/core-metadata/#requires-dist-multiple-use

[60] C. Wang, R. Wu, H. Song, J. Shu, and G. Li, "smartPip: A smart approach to resolving Python dependency conflict issues," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 93:1–93:12. [Online]. Available: https://doi.org/10.1145/3551349.3560437

[61] (2023, April) Backtracking in dependency resolution - pip documentation v23.1.1. [Online]. Available: https://pip.pypa.io/en/stable/topics/dependency-resolution/#backtracking

[62] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S. Cheung, C. Xu, and Z. Zhu, "Watchman: Monitoring dependency conflicts for Python library ecosystem," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 125–135. [Online]. Available: https://doi.org/10.1145/3377811.3380426

[63] (2023, April) Sample size calculator. [Online]. Available: https://www.calculator.net/sample-size-calculator.html

[64] (2023, April) Various licenses and comments about them. [Online]. Available: https://directory.fsf.org/wiki/License:Apache2.0

[65] (2023, April) What is derivative work? [Online]. Available: https://opensource.stackexchange.com/questions/6427/

[66] (2023, April) What are the arguments for considering dynamic links to constitute derivative works? [Online]. Available: https://opensource.stackexchange.com/questions/1187/

[67] (2023, April) What are the arguments for considering dynamic links not to constitute derivative works? [Online]. Available: https://opensource.stackexchange.com/questions/1188/

[68] S. H. Khandkar, "Open coding," *University of Calgary*, vol. 23, p. 2009, 2009.

[69] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*. IEEE, 2021, pp. 72–83.

[70] H. He, R. He, H. Gu, and M. Zhou, "A large-scale empirical study on java library migrations: prevalence, trends, and rationales," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 2021, pp. 478–490. [Online]. Available: https://doi.org/10.1145/3468264.3468571

[71] H. He, Y. Xu, X. Cheng, G. Liang, and M. Zhou, "MigrationAdvisor: Recommending library migrations from large-scale open-source data," in *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 9–12.

[72] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the complexity of large free and open source package-based software distributions," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*. IEEE Computer Society, 2006, pp. 199–208. [Online]. Available: https://doi.org/10.1109/ASE.2006.49

[73] D. Pinckney, A. Guha, M. Culpo, and T. Gamblin, "Flexible and optimal dependency management via Max-SMT," *CoRR*, vol. abs/2203.13737, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2203.13737

[74] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, L. Zhao, J. Wu, and Y. Liu, "Compatible remediation on vulnerabilities from third-party libraries for Java projects," *CoRR*, vol. abs/2301.08434, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2301.08434

[75] (2023, April) Semantic versioning. [Online]. Available: https://semver.org/

[76] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.

[77] (2023, April) PEP 440 – Version identification and dependency specification. [Online]. Available: https://peps.python.org/pep-0440/

[78] (2023, April) GPLv2 with linking exception. [Online]. Available: https://www.pygit2.org/#license-gplv2-with-linking-exception

[79] (2023, April) OSI the open source definition. [Online]. Available: https://opensource.org/osd/

[80] C. Vendome, M. L. Vásquez, G. Bavota, M. D. Penta, D. M. Germán, and D. Poshyvanyk, "Machine learning-based detection of open source license exceptions," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 2017, pp. 118–129.

[81] P. Abate, R. D. Cosmo, G. Gousios, and S. Zacchiroli, "Dependency solving is still hard, but we are getting better at it," in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. IEEE, 2020, pp. 547–551.