EasyPip: Detect and Fix Dependency Problems in Python Dependency Declaration Files

Shuo Li
Institute of Software Chinese Academy of Sciences, University of Chinese Academy of Sciences
Beijing, China
lishuo19@otcaix.iscas.ac.cn

Jie Liu*
State Key Lab of Computer Sciences at ISCAS, University of Chinese Academy of Sciences University of Chinese Academy of Sciences,Nanjing
China
ljie@otcaix.iscas.ac.cn

HaoXiang Tian
Institute of Software Chinese Academy of Sciences, University of Chinese Academy of Sciences
Beijing, China
tianhaoxiang20@otcaix.iscas.ac.cn

Shuai Wang*
Institute of Software Chinese Academy of Sciences, University of Chinese Academy of Sciences
Beijing, China
wangshuai@otcaix.iscas.ac.cn

Wei Chen
State Key Lab of Computer Sciences at ISCAS, University of Chinese Academy of Sciences University of Chinese Academy of Sciences,Nanjing
wchen@otcaix.iscas.ac.cn

Liangyi Kang, Dan Ye
Institute of Software Chinese Academy of Sciences
Beijing, China
{kangliangyi15,yedan}@otcaix.iscas.ac.cn

Abstract

Environment configuration is the basis for software reuse, enabling developers to reuse specific functions. However, the lack of uniform practice in dependency declaration specifications of Python projects can cause problems for developers trying to install third-party libraries. Existing package management tools are often inadequate to help fix these problems. Fixing these errors requires expensive hours and domain knowledge for developers.

To help address related problems, some studies focus on well-maintained and popular Python projects about dependency conflict problems caused by PIP's installation rules. However, many projects in the wild are outside of this scope. We carefully investigate 110 issues in 110 projects in the wild. Based on the comprehensive study, we design and implement EasyPip to automatically detect and fix problems in Python dependency declaration files. Different from existing tools, EasyPip can locate conflicting dependencies without trying to install dependencies, and generate fixing solutions with the least modification to the original files. We evaluate EasyPip on the collected dataset which shows that EasyPip outperforms two state-of-the-art tools and can effectively detect problems in 91.04% Python dependency declaration files and generate feasible fixing solutions for 65.67% of them.

***Index Terms*—Software reuse, dependency declaration file, dependency error fix, Python**

I. Introduction

Python is one of the most popular programming languages, but its dependency declaration lacks uniform specifications, which may lead to unknown problems. We call the third-party library installation problems caused by the dependency declaration files instead of package management tools dependency declaration issues. The issues can be seen in real projects, such as Im2Vec [1] whose requirements.txt file specifies conflicting dependencies, leading to dependency declaration issues.

DOI reference number: 10.18293/SEKE2023-120

ISCAS are short for Institute of Software Chinese Academy of Sciences

* Both Jie Liu and Shuai Wang are the corresponding authors of this paper.

The dependency declaration file should contain all third-party libraries used in the project to ensure successful installation. PIP and Conda can export the dependency declarations of the installed third-party libraries in the environment of the project to help reuse Python projects. However, the compatibility between direct dependencies and transitive dependencies is ignored. Recent techniques like PyEgo [2] infer compatible environment dependencies' versions for Python code snippets, but couldn't identify potential conflicts that weren't in their knowledge database. And they ignore that the original dependency declaration file provided by the project developer is closer to the environment configuration that the developer expects. And developers will likely perform smaller updates to mitigate the impact of breaking changes. So, the fewer modification to the original dependency declaration file, the better to successfully build the project and achieve the expected results. Therefore, detecting and fixing dependency declaration issues with less modification is helpful for developers to reuse Python projects.

Wang et.al. [3], [4] investigate dependency conflict issues with the legacy and new dependency-resolving strategies adopted in PIP. However, the studies focus on well-maintained projects. There are many projects left in the wild not well-maintained or unpopular. Developers will face more challenges when reusing these projects.

To investigate the characteristics of dependency declaration issues of projects in the wild, we conduct an empirical study on dependency declaration files in Python projects from Github. We collect 1000 issues related to dependency declaration, and among them, 110 dependency declaration issues of 110 Python projects are identified. We thoroughly analyze these issues and conclude manifestation patterns from them. Note that different from the empirical study of Watchman [3], our study mainly focuses on the dependency conflicts caused by dependency declaration files instead of those caused by the installation strategy of PIP.

Based on our study, to address dependency declaration issues and help developers successfully install third-party libraries when reusing Python projects, we design and implement a tool, EasyPip, which can automatically detect and fix dependency declaration issues. Differently from the existing tools (e.g., PIP, SMARTPIP [4]), EasyPip can detect and locate dependency declaration issues without trying to install, and generate feasible fixing solutions with the least modification to the original files. Specifically, EasyPip formulates the detection and fixing of dependency declaration issues as a graph-search problem. To improve the efficiency of detection and location, we introduce the *equivalent node* to reduce search space. To generate the feasible fixing solutions with the least modification, EasyPip utilizes greedy search to find the compatible versions closest to the original dependency declaration file.

We evaluate EasyPip on the dataset of dependency declaration files with dependency declaration issues from open-source projects. The results show that, EasyPip can effectively detect 91% dependency declaration issues and report the root causes, and generate feasible fixing solutions for 72% of the detected issues. Comparing to the state-of-art techniques for detecting and fixing dependency declaration issues, on the same dataset, EasyPip detects 12% more dependency declaration issues and generates 34% more feasible fixing solutions. For the fixing solutions generated by EasyPip and the selected baselines, we calculated their changes in dependencies and versions to the corresponding original files. The results show that the fixing solutions generated by EasyPip are closer to the original dependency declaration files.

In summary, the main contributions of our work are as follows:

- We conduct the empirical study of dependency declaration issues in Python projects in the wild. Our findings can help further understand the characteristics of dependency installation problems.
- Based on our findings, we design and implement a tool, EasyPip, to help automatically detect and locate dependency declaration issues without installing, and generate feasible fixing solutions with the least modifications to the original dependency declaration file.
- We release the dataset used in our empirical study and evaluation, which can facilitate future research related to Python dependency declaration issues.

II. EMPIRICAL STUDY

A. Data Collection

To collect dependency declaration issues, we follow the data collection practice of empirical studies [5]–[8] on Python developer communities, and collect our data by the following steps: First, we go through issues that are related to dependency declarations of Python projects on GitHub. Next, we manually read the issue reports and comments, to identify whether it is a dependency declaration issue.

(1) Collecting Python projects: We search Python projects on GitHub with issues filtered by the keywords: ”requirement” ”dependency” or ”install”, and obtain 1000 issues. Then we filter out the projects without dependency declaration files (requirement.txt and setup.py). The duplicate issues are dropped with the first one left for these keywords can appear at the same time in one issue.

(2) Identifying dependency declaration issues: We then go through each issue report and its comments, and further identify dependency declaration issues by several criteria: a. the issue is caused by a dependency declaration file instead of the user's incorrect operation and PIP's installation, b. the corresponding release of the project is available so that the issue can be reproduced. Three of the authors independently check each issue, and then discuss it to reach an agreement. After filtering by the criteria above, there are 110 issues in 110 projects left.

B. Manifestation Patterns

We categorize dependency declaration issues into four categories (Pattern A to D) according to their occurrence stages and root causes.

Manifestation of Pattern A: The dependency declaration issues of Pattern A manifest as different dependencies required by the project cannot be satisfied at the same time(57/110). As conflicts come from different sources, Pattern A is divided into three sub-categories. *Pattern A.1:* conflicts between direct and transitive dependencies (47/57). *Pattern A.2:* conflicts between transitive dependencies (10/57).

Manifestation of Pattern B: The issues of Pattern B are caused by Python interpreters(10/110). Different dependencies in the dependency declaration file require different Python interpreters.

Manifestation of Pattern C: The issues of Pattern B are caused by incompatibility with Operating System (13/100). The dependency in the dependency declaration file is incompatible with the OS of the developer who wants to reuse the project.

Manifestation of Pattern D: The dependencies of the project and that of its upstream/downstream project, cannot be satisfied at the same time (19/110).

The other issues (11/110) are caused by unavailable dependency (e.g., library not found or not released publicly) declared in the file.

From our study, it can be seen that most dependency declaration issues are caused by conflicts among declared dependencies, transitive dependencies, and Python interpreters.

III. EasyPip

To help automatically solve the dependency declaration issues without trying to install dependencies, we design and implement a tool, EasyPip, which can efficiently detect and locate the issues of the two most common patterns (Pattern A and B). EasyPip can also find feasible fixing solutions with minimal modifications to the original dependency declaration file as developers will likely perform smaller updates to mitigate the impact of breaking changes. The workflow of EasyPip is shown as Figure 1.

Given a Python dependency declaration file, EasyPip first resolves its dependencies and obtains all transitive dependencies. Then EasyPip constructs the dependency graph by the relations of Python interpreters and these dependencies. EasyPip formulates the detection and fixing of dependency declaration issues as a graph search problem. If there is no fully-connected path that satisfies all declared version constraints in the dependency declaration file, EasyPip will report a dependency declaration issue and conflicting dependencies. For the conflicting dependencies, EasyPip utilizes greedy search to find the closest version that can constitute a fully-connected path of the dependency graph.

Dependency Graph Construction The input of EasyPip is a dependency declaration file. Firstly, EasyPip parses it to get the third-party libraries and their version constraints. Then EasyPip queries their transitive dependencies from the dependency knowledge database. Based on these dependencies and their relations, EasyPip constructs the dependency graph of these dependencies with Python interpreters.

In the dependency graph, each node represents a Python interpreter or a third-party library with a specific version. The edges contain two types: seq_edge (the undirected edge) and dep_edge (the directed edge). The nodes of different versions in the same third-party library or interpreter are connected by seq_edges. The nodes of different libraries or interpreters with dependency relations are connected by dep_edges. As the latest versions are preferred in Python project development, the nodes of the same library or interpreter are ordered in descending of versions.

For example, there are two dependency declarations in the file: $B == 5, C$. The dependency relations of B and C are shown as Table I, and the constructed dependency graph is shown as Figure 2. (Note that to intuitively display the example, the Python interpreters are not shown in the figure). In the dependency graph, for each library, EasyPip denotes the nodes with maximal version and minimal version of corresponding declared version constraints (If there is no declared version constraint on the dependency, the nodes from the oldest version to the newest version of the dependency will all be denoted).

TABLE I. Dependency relation

Package	Version	Dependent package	Dependent version
B	V1-V2	A	V4
B	V3-V4	A	V6
B	V5	A	>V7
C	V1	A	V5
C	V2-V3	A	V6

To reduce the time-consuming of dependency graph construction, the queried metadata of third-party libraries via PyPi is stored locally as a dependency knowledge database. Each item in the database is represented by a 2-tuple $< p, v >$. p is the name of a third-party library. v is a

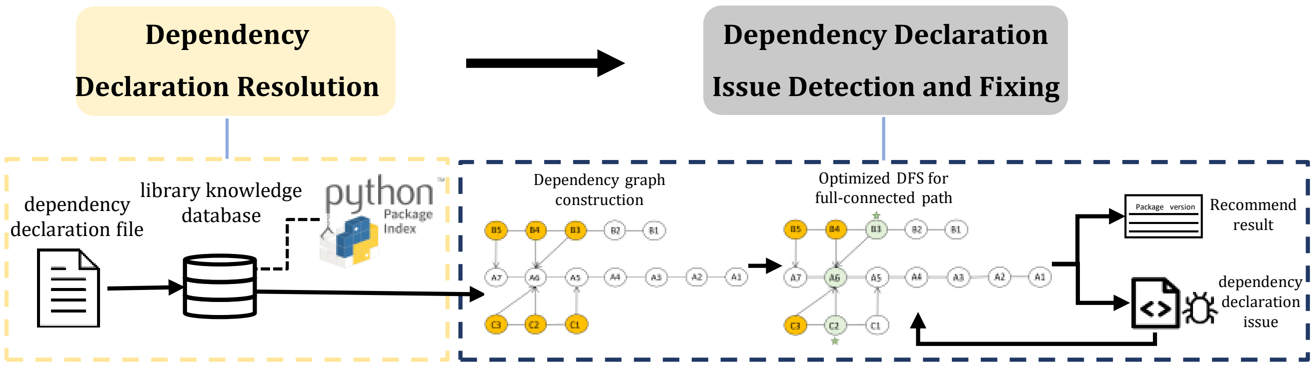
Dependency Declaration Resolution
Dependency Declaration Issue Detection and Fixing
dependency declaration file
library knowledge database
python Package Index
Dependency graph construction
Optimized DFS for full-connected path
Recommend result
dependency declaration issue

Fig. 1. The workflow of EasyPip.

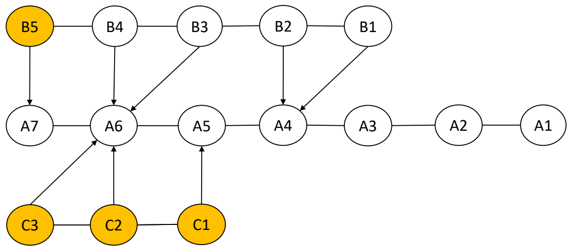

Fig. 2. An example of the dependency graph.

2-tuple represented as $< vid, td >$, where vid is a version of p, and td is a set of third-party libraries with specific versions that $p.vid$ depends on. Before querying PyPi, EasyPip queries the database. If the database contains the information of the third-party library, EasyPip will not query PyPi. Considering that in the actual development, for a developer, there may be many common third-party libraries used in projects. Therefore, with EasyPip used by the developer more times, the local knowledge database can reduce time costs. EasyPip will update the information in the local knowledge database periodically.

Dependency Declaration Issue Detection Based on the constructed dependency graph, EasyPip detects dependency declaration issues by searching a fully-connected path that satisfies all denoted version constraints in the graph. Note that the interpreter and each library can only have one node on the fully-connected path. The fully-connected path represents the feasible solution satisfying all declared constraints of the dependency declaration file. Therefore, if there is no fully-connected path that can satisfy all denoted version constraints in the dependency graph, EasyPip will report a dependency declaration issue and the conflicting dependencies that cannot find compatible versions within declared version constraints.

To determine the search order that can improve search efficiency and align with the PIP installation policy, EasyPip applies topological sorting to the dependency graph. To do this, the nodes of the same library are regarded as a whole in the topological sorting. For two libraries i and j, if there exists any node of i that connects to the node of j by a dep_edge, the in-degree of library j is added 1. For example, in the Figure 2, the in-degree of library A is 2. EasyPip calculates the degree of each library. The higher the degree of the library, the more constraints from other libraries (we call them pre-constraints) on the library. For the library, EasyPip should search for compatible versions after its pre-constraints are satisfied, which can reduce the number of backward searches. Therefore, EasyPip applies topological sorting to the dependency graph in ascending order of libraries' in-degree. In this example, the topological order is B, C, and A.

Based on the sorted graph, EasyPip searches for the fully-connected path that satisfies all denoted version constraints. It travels from the starting library or interpreter to the tail library. For the nodes of the same library or interpreter, EasyPip searches from the node with a newer version to an older version within the denoted node range.

To reduce the search space, we introduce the *Equivalent nodes*, which refers to different nodes of the same library or interpreter that have the same dependency relations with the nodes of other third-party libraries. Taking an example in Figure 2, for library B, B4 and B3 are equivalent nodes. For one node, if there is no fully-connected path containing the node, there is also no fully-connected path containing its equivalent nodes. Therefore, for each library, instead of traversing all nodes, after finishing the search of one node, EasyPip will skip its equivalent nodes.

Dependency Declaration Issue Fixing When a dependency declaration issue and conflicting dependencies are reported, EasyPip will fix it by a greedy search of nodes in conflicting libraries. Considering the developers tend to make smaller updates to minimize the impact of any breaking changes, the metric function is the distance from the original declared version range. To fix the issue with the least modification, for the conflicting libraries, EasyPip will find the compatible nodes closest to the denoted version range. Specifically, EasyPip traverses the nodes of the conflicting library from the upper boundary of denoted node range to the node with the latest version. If EasyPip fails to find the fully-connected path, it will traverse the nodes from the lower boundary of denoted node range to the node with the oldest version. The first fully-connected path found by EasyPip is the closest solution to the declared constraints in the original dependency declaration file, and EasyPip will recommend it as the fixing solution.

Taking Figure 2 as an example, as the dependency declaration requires $B == 5, C$, there is no fully-connected path that satisfies the declared version constraints of B. EasyPip fixes the issue by greedy search from B4 to B1 to find a fully-connected path (B3 will be skipped as it is an equivalent node of B4). $B4$ is the first node of library B that can constitute a fully-connected with A and C, and its version is closest to the declared version range in the original dependency declaration file. EasyPip will recommend $B4, C3, A6$ as the fixing solution.

IV. EVALUATION

To demonstrate the effectiveness and efficiency of EasyPip, we evaluate it on the dataset of dependency declaration files with dependency declaration issues. The dataset contains 67 dependency declaration files, which are from the issues of Pattern A and Pattern B in our empirical study. We answer the following questions:

RQ1: How effective is EasyPip in detecting and fixing dependency declaration issues?

RQ2: How effective and efficient is EasyPip to detect and fix dependency declaration issues compared to other state-of-art techniques? Whether EasyPip can generate feasible fixing solutions with less modification to original dependency declaration files?

A. Experiment Design

For RQ1, we run EasyPip to detect and fix dependency declaration issues in our dataset.

For RQ2, we evaluate EasyPip in comparison to two recently state-of-art techniques for detecting and fixing dependency declaration problems, SMARTPIP [4] and PyEgo [2]. We select the baselines following the criteria: (1)the tools are available, (2) they should be align with the goals of our research. SMARTPIP [4] is an open-source technique to search for the dependencies' compatible versions within declared version constraints, which translates concerned libraries' version constraints collected in a pre-built knowledge base into SMT expressions for dependency resolution, resulting in superior performance compared to PIP no matter pre or post V20.3. If SMART-PIP fails to find compatible versions of dependencies within all declared version constraints, it will report a conflicting issue. We run EasyPip and SMARTPIP to detect dependency declaration issues in the same dataset, and compare their effectiveness and efficiency from the following aspects:

1) How many dependency declaration issues are detected?
2) How long does it take to detect the issue?

While SmartPip cannot provide fixing solutions for dependency declaration issues, to compare the performance of fixing dependency declaration issues, we select PyEgo, a tool to generate feasible solutions for versions of dependencies used in projects according to code snippets. We run EasyPip and PyEgo to fix the same dependency declaration issues, and compare them from the following aspects:

1) How many dependency declaration issues can be fixed successfully?
2) How many modifications are made to the original dependency declaration files?

B. Answer to RQ1

In the 67 dependency declaration files with dependency declaration issues, EasyPip successfully detects 61 of them and reports the root causes. It takes EasyPip 7.5 seconds on average to detect the issue in one dependency declaration file. For the detected dependency declaration issues, EasyPip generates 44 feasible fixing solutions.

We reviewed the 6 dependency declaration files that EasyPip failed to detect issues and found that they were due to some dependencies whose requires_dist on PyPI are missing. To verify our analysis, we manually obtain the corresponding information of them and add it to the library knowledge database. Then EasyPip can successfully detect and report the correct root cause of the 6 dependency declaration issues.

As for the fixing solutions generated by EasyPip, we need to verify their correctness. To do this, for each fixing solution, we installed and built them in the corresponding projects, to check whether it can be successfully installed and built without errors. The 44 fixing files generated by EasyPip are successfully installed and built. For the 17 dependency declaration issues that EasyPip failed to generate fixing solutions, we reviewed the running log of EasyPip and found that they were due to ”timed out”. We manually reproduce the fixing of these issues. In the process of searching, some libraries have too many versions which results in a too long time of searching.

TABLE II. Comparison result with PyEgo

		min	avg	max
The number of dependency changes	PyEgo	0	+1.10	+7
	EasyPip	0	0	0
	PyEgo	-3	-19.2	-55
	EasyPip	0	0	0
The number of dependency version changes	PyEgo	0	2.90	12
	EasyPip	1	1.43	3
The version distance	PyEgo	0	6.54	2
	EasyPip	0.04	1.06	11

C. Answer to RQ2

Compare to SMARTPIP: In 67 dependency declaration issues, SMARTPIP detects 53 of them but does not report the root cause. It takes 25 seconds on average to detect the dependency declaration issue in one file. We reviewed and analyzed the 14 issues that SMARTPIP failed to detect and found that they were due to incompatible versions of Python interpreters. From the comparison results, it can be seen that EasyPip detects dependency declaration issues more efficiently than SMARTPIP. Furthermore, compared with it, EasyPip has the ability to locate the root causes of dependency declaration issues and detect the issues caused by incompatible versions of Python interpreters.

Compare to PyEgo: PyEgo fixes 21 out of 67 dependency declaration issues, and the 21 fixing dependency declaration files are successfully installed and built in corresponding projects. Compared with PyEgo, EasyPip generates more feasible fixing solutions for the same dataset of dependency declaration issues. EasyPip also fixes all 21 issues. To fairly compare EasyPip and PyEgo, we evaluate both tools on the 21 common issues.

To compare the modification to the original dependency declaration file, we define three indicators:

(1) The number of dependency changes: measures the change in the number of dependencies from the original to the fixing file. For example, if there are two dependencies in the original file, and the number of dependencies in the corresponding fixing solution is three. The number of dependency changes is 1. **(2) The number of dependency version changes:** counts the number of dependencies with version constraints that differ between the original and fixing files. For example, if there is a dependency d that the version constraint of d in the fixing file is different from its version constraint in the original file, the number of dependency version changes is added 1. **(3) The version distance:** quantifies the degree of version changes from the original to the fixing file. For example, the original dependency declaration file is *"A==1, 1 <=C <=3"* and the fixing file is *"A==2, C==4"*, the version distance between the original file and fixing file is 2 (the version distance of A is 1, and the version distance of C is 1).

By comparing the three indicators above of EasyPip and PyEgo on the same dependency declaration files, from Table II, we can conclude that the feasible fixing solutions generated by EasyPip make less modification to the original dependency declaration files.

V. Threats to Validity

Similar to other bug-related studies [3], [9], the collected data and researchers involved, and keyword search can include irrelevant issues. And manual analysis can introduce bias. To reduce these threats, the researchers carefully reviewed the relevant information and discussed the issues until reaching an agreement. Another threat comes from the collected knowledge. The collected dependencies relationship between third-party libraries and the system libraries may introduce false positives. To alleviate this, we acquire the knowledge following the practice of [2], [10]. And our dataset and analysis results are publicly available. It can help other researchers for further analysis and validate our study results.

VI. RELATED WORK

Dependency Inference. Techniques like DockerizeMe [11], V2 [12] PyCRE [13], and PyEgo [2] can build an environment specification for the code snippet as a Dockerfile. PyEgo conduct static analysis and infer dependencies by solving constraints with Z3. The dependencies they infer must be in their knowledge database, so there is potential conflict among the dependency the developer want but are not listed in the Dockerfile. Different from them, EasyPip only takes the dependency declaration file into consideration for the original dependency declaration file is most close to the environment configuration that the developer expects.

Dependency conflict Dependency conflict(DC) issues hinder the reusability of open-source projects. Researchers have developed various approaches to detect and analyze DC issues [14]–[17]. Watchman [3] characterize DC issues caused by PIP's installation rule in popular Python projects. Besides them, SMARTPIP is the closest to our work. However, it aims to improve the dependency constraints-solving strategy of PIP. It cannot provide useful information to help locate the root cause of DC issues. And they don't take the Python interpreter into consideration during dependency resolution. PyDFix [18] concentrated on detecting and fixing unreproducibility in Python builds caused by third-party library errors. They take the old and current build logs as input and try to fix the errors by iteratively trying the other version of the same library. They indicate that it's efficient to fix dependency declaration issues by adjusting the dependency constraints. To the best of our knowledge, there is no previous work focusing both on detecting and fixing issues at the same time in the dependency declaration files in the Python world.

VII. Conclusion

In this paper, we design and implement a technique, EasyPip, to automatically detect and fix dependency declaration issues. We evaluate EasyPip with existing state-of-art techniques on both the dependency resolution task and dependency declaration issues fixing on the benchmark and collected dependency declaration issues in real-world Python projects. The results show that EasyPip can efficiently conduct dependency resolution, and detect and fix more dependency declaration issues with fewer modifications to the original dependency declaration files released by developers. In the future, we plan to further improve the detection and fixing capability of EasyPip on more patterns of dependency declaration issues.

VIII. ACKNOWLEDGMENT

We thank Keqin Xu, Zhiyang Zhou and Zhaolin Shi for their contributions. This work was partially supported by National Key R&D Program of China Grant (2017YFA0700603), National Natural Science Foundation of China (61972386).

References

[1] issue#22 of Im2Vec. (2022). [Online]. Available: https://github.com/preddy5/Im2Vec/issues/22

[2] H. Ye, W. Chen, W. Dou, G. Wu, and J. Wei, "Knowledge-based environment dependency inference for python programs," pp. 1245–1256, 2022. [Online]. Available: https://doi.org/10.1145/3510003.3510127

[3] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, "Watchman: Monitoring dependency conflicts for python library ecosystem," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 125–135.

[4] 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*, 2022, pp. 1–12.

[5] S. Lewis, "Qualitative inquiry and research design: Choosing among five approaches," *Health promotion practice*, vol. 16, no. 4, pp. 473–475, 2015.

[6] J. Hu, L. Wei, Y. Liu, S.-C. Cheung, and H. Huang, "A tale of two cities: How webview induces bugs to android applications," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 702–713.

[7] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1013–1024.

[8] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 226–237.

[9] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications." New York, NY, USA: Association for Computing Machinery, 2014.

[10] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for python build reproducibility." New York, NY, USA: Association for Computing Machinery, 2021.

[11] E. Horton and C. Parnin, "Dockerizeme: automatic inference of environment dependencies for python code snippets," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 328–338.

[12] ——, "V2: Fast detection of configuration drift in python," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 477–488.

[13] W. Cheng, X. Zhu, and W. Hu, "Conflict-aware inference of python compatible runtime environments with domain knowledge graph," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 451–461.

[14] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 319–330. [Online]. Available: https://doi.org/10.1145/3236024.3236056

[15] Y. Wang, M. Wen, R. Wu, Z. Liu, S. H. Tan, Z. Zhu, H. Yu, and S. C. Cheung, "Could i have a stack trace to examine the dependency conflict issue?" *ICSE*, pp. 572–583, 2019.

[16] J. Patra, P. N. Dixit, and M. Pradel, "Conflictjs: finding and understanding conflicts between javascript libraries," pp. 741–751, 2018.

[17] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S. C. Cheung, H. Yu, C. Xu, and Z. Zhu, "Will dependency conflicts affect my program's semantics?" *IEEE Transactions on Software Engineering*, vol. 48, pp. 2295–2316, 2022.

[18] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for python build reproducibility," in *ISSTA*, 2021, pp. 439–451.