How to Pet a Two-Headed Snake? Solving Cross-Repository Compatibility Issues with Hera

Yifan Xie
National University of Defense Technology
Changsha, China
xieyifan@nudt.edu.cn

Zhouyang Jia*
National University of Defense Technology
Changsha, China
jiazhouyang@nudt.edu.cn

Shanshan Li†
National University of Defense Technology
Changsha, China
shanshanli@nudt.edu.cn

Ying Wang
Northeastern University
Shenyang, China
wangying@swc.neu.edu.cn

Jun Ma
National University of Defense Technology
Changsha, China
majun@nudt.edu.cn

Xiaoling Li
National University of Defense Technology
Changsha, China
lixiaoling@nudt.edu.cn

Haoran Liu
National University of Defense Technology
Changsha, China
liuhaoran@nudt.edu.cn

Ying Fu
National University of Defense Technology
Changsha, China
fuying@nudt.edu.cn

Xiangke Liao
National University of Defense Technology
Changsha, China
xkliao@nudt.edu.cn

ABSTRACT

Many programming languages and operating system communities maintain software repositories to build their own ecosystems. The repositories often provide management tools to help users using the packages. The tools are often, if not all the times, well-designed to handle intra-repository dependencies without considering inter-repository dependencies. The users, however, often need packages from different repositories, and thus may suffer from compatibility issues. We refer to these issues as ***C**ross-repository **C**ompatibility (CC) issues*. Existing works typically focus on a single software repository and are insufficient to detect CC issues.

To fill this gap, we use both Python and Ubuntu repositories as representatives to study the root cause of CC issues, then summarize their triggering patterns and failure symptoms. Guided by the above analysis, we design Hera, an automatic tool to solve CC issues. Hera first builds a cross-repository compatibility database offline, and then online predicts, detects and fixes CC issues in the user's system environment. In our evaluation, we construct a dataset of 1,692 real-world CC issues, and Hera can detect 3,689 issues with the precision of 90.5% and the recall of 93.7%. We also collected 27 real-world CC issues from GitHub and Stack Overflow, and reproduced 26 of them. Hera can detect all the 26 cases, and provide accurate reasons as well as fixing advice.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**.

KEYWORDS

Python Repository, Ubuntu Repository, Compatibility Issue

ACM Reference Format:
Yifan Xie, Zhouyang Jia, Shanshan Li, Ying Wang, Jun Ma, Xiaoling Li, Haoran Liu, Ying Fu, and Xiangke Liao. 2024. How to Pet a Two-Headed Snake? Solving Cross-Repository Compatibility Issues with Hera. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695064

1 INTRODUCTION

Package management tools are widely used to manage software dependency and automate the process of package installation. Many programming language communities provide their own management tools. For example, *pip* is responsible for managing nearly half a million Python packages [15], while *npm* handles over three million JavaScript packages [11]. Besides, operating system communities also provide management tools like *apt* and *dnf*. These tools manage packages for operating system distributions instead of specific programming languages. For instance, in distributions like Debian/Ubuntu, *apt* is capable of managing *deb* packages including but not limited to C/C++/Python/JavaScript [1]. Similarly, *dnf* can handle *rpm* packages of different programming languages in distributions like RHEL/Fedora [5].

A package management tool itself is often, if not all the times, well-designed to handle the dependencies of packages in its own

*Zhouyang Jia is the co-first author.
†Shanshan Li is the corresponding author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695064

```
1. $ apt install python3-amp
2. Successfully install python3-amp 0.6.1 python3-ase 3.19.0-1
                    python3-scipy1.3.3 python3-numpy 1.17.4
3. $ python3
4. >>> import amp
5. $ pip install pandas
6. Successfully install pandas 1.5.3 numpy 1.24.3
7. $ python3
8. >>> import amp
9. Traceback (most recent call last):
10.  ⋯
11.  File "/home/xxx/.local/lib/python3.8/sitepackages/numpy/__init__.py", ⋯
12.    raise AttributeError("module {!r} has no attribute "
13. AttributeError: module 'numpy' has no attribute 'typeDict'
```

Figure 1: Package incompatibility due to mixing *apt* and *pip* installation in Ubuntu.

repository. To achieve this, the repository usually requires its packages explicitly specify dependent packages and corresponding version ranges. From the view of an end user, however, they often need packages cross different repositories. For example, Ubuntu users may alternately use *apt* and *pip*. In these cases, compatibility issues may happen. Figure 1 illustrates a real-world example, which contains a series of commands installing Python packages using different management tools. Line 1 uses *apt* to install "`amp 0.6.1`" (an atomic-level machine learning package), as well as its dependencies "`ase 3.19.0-1`", "`scipy 1.3.3`", and "`numpy 1.17.4`". At this point, "`amp`" can be imported normally (Line 3-4). Line 5 then uses *pip* to install "`pandas 1.5.3`" and its dependency "`numpy 1.24.3`". After that, "`amp`" can not be imported any more (Line 7-8). According to the traceback information, "`amp`" is incompatible with "`numpy`", which locates at "$HOME/.local/lib/python3.8/site-packages/", i.e., the install path of *pip*. This means that when importing "`amp 0.6.1`" (installed via *apt*), Python interpreter will import "`numpy 1.24.3`" (installed via *pip*) instead of "`numpy 1.17.4`" (installed via *apt*) , and thus resulting in a compatibility issue. We refer to this issue as a ***C**ross-repository **C**ompatibility (CC) issue.*

There has been a long research history on addressing compatibility issues, which can be roughly divided into two major categories. First, many works focus on detecting dependency conflicts (i.e., if one package depends on two conflicting versions of another package) [27, 36, 45, 49, 50, 52–55] and dependency resolving (i.e., if there is a package version that is compatible with all installed packages) [24, 25, 48]. The former often addresses in a given programming language community (e.g., Python, JAVA, JavaScript), while the latter usually works on an operating system distribution. All these works are hard to handle cross-repository issues. Second, some works apply testing or program analysis techniques to detect package compatibility (i.e. if two packages witinh given versions are compatible with each other) [28, 33, 40, 42, 46, 56]. These works can determine if the specified version ranges of dependent packages (often in package specification files) contain incompatible versions. Cross-repository compatibility issues, however, often happened even the version ranges inside all repositories are correct.

In this paper, we focus on solving CC issues, which are caused by inter-repository dependencies. We assume both *apt* and *pip* are correct, since they only handle intra-repository dependencies. To achieve this, we propose Hera, an automatic tool to *predict*, *detect* and *fix* CC issues:

- **Predicting:** When user execute installation commands, Hera predicts if the commands could cause CC issues.
- **Detecting:** Hera scans all Python packages installed in user's system, and detects if there are CC issues.
- **Fixing:** When CC issues were detected, Hera provides fixing advice to prevent users from failures.

As all the three scenarios are working in the user's production environment, the overhead should be critical when deploying Hera. In this regard, the insight of Hera is to build a cross-repository compatibility database offline, and then online predict, detect and fix CC issues. Both the offline and online phases are challenging:

- **Building a cross-repository compatibility database is hard.** The software repository may contain a large number of packages (e.g., *pip* is managing nearly half a million packages), and most of them have long evolution histories. These packages keep evolving asynchronously, and thus the database has to keep updating accordingly. The combinations of dependency relationships between packages from different repositories would be huge. It would be nearly impossible to consider all possible combinations of package versions.
- **Predicting, detecting and fixing CC issues are non-trivial.** Each repository has its own management tool with unique installing strategy and directory. When installing a package, *apt* installs *a specific version* if the package is not found in *its own installing directory*, whereas *pip* installs *the latest version* when the package is not found in *any system-level directory*. When importing a package, Python interpreter tries all directories according to a pre-defined order. A CC issue involves interactions of the above three roles, i.e., *apt*, *pip* and Python interpreter, making it complex to be predicted, detected or fixed.

To address the first challenge, we study the root cause of real-world CC issues and find that each issue contains an *application package* and a *library package* (e.g., "`amp`" and "`numpy`" in Figure 1): the application package is always hosted in the *apt* repository, whereas the library package is always hosted in both the *apt* and *pip* repositories. Please note that, the *application* and *library* are relative concepts as an *application* itself may be a *library* of anther *application*. This finding implies two relationships: a) the application package is compatible with the library package hosted in *apt*; b) the library package hosted in *apt* is incompatible with the same package hosted in *pip*. In this regard, Hera creates two tables in the compatibility database:

- **Dependency table:** For each application package in the *apt* repository, this table collects its API usages of each library package in the *apt* repository.
- **Compatibility table:** For each library package in the *apt* repository, this table collects its API compatibility with each version of the same package hosted in the *pip* repository.

These two tables provide sufficient information about CC issues, and the scale is acceptable since *apt* only manage about 3,319 Python packages with selected versions. When an OS distribution is released, Hera builds a new database. After that, Hera incrementally fetches the latest package versions from the *pip* repository.

As for the second challenge, we propose a system-level package dependency graph (S-PDG) to describe the interactions among *apt*, *pip* and Python interpreter. The S-PDG contains dependencies of all Python packages in the user's system. To achieve this, Hera first builds two dependency graphs of packages installed by *apt* and *pip*, respectively. We refer to these two graphs as repository-level package dependency graphs (R-PDG). After that, Hera merges the two R-PDGs into one S-PDG according to the importing rules of Python interpreter. When two versions of a package occur in different R-PDGs, Hera queries the *compatibility table* to determine if there are breaking APIs between the versions. If yes, Hera further queries the *dependency table* to check if there is an application package in the system using the breaking APIs. If yes, Hera reports a CC issue. Finally, Hera can provide fixing advice according to the dependencies in R-PDGs, which are regarded as compatible since only containing intra-repository dependencies. For the prediction scenario, Hera dry-runs the installation command, then temporarily adds the packages to be installed into the S-PDG.

To evaluate Hera, we first constructed a real-world dataset of CC issues based on all Python packages maintained by *apt*. The repository includes 23,866 pairs of application and library packages. We installed the application and library by *apt* and *pip* respectively, and 1,692 pairs failed when importing. We then used Hera to analyze all the pairs and detected 3,689 CC issues with the precision of 90.5%. These issues can cover 93.7% of the above 1,692 ones. To evaluate the effectiveness of analyzing incompatible API changes, we manually checked the results and found the precision and recall are 91.1% and 98.2% at a 95% confidence level and a error margin of 5%, respectively. Finally, we collected 27 real-world CC issues from GitHub and Stack Overflow, and reproduced 26 of them. Hera can detect all of the 26 cases, and provide accurate reasons as well as fixing advice.

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

- To the best of our knowledge, this is the first work addressing compatibility issues across software repositories. We defined the CC issues, and studied them from the Python perspective. The problem could be extended to other programming languages.
- We designed Hera, a novel tool to build a cross-repository compatibility database offline, and the online predict, detect and fix CC issues. Hera could analyze the whole software repositories, and keep lightweight in users' production environment.
- We conducted a comprehensive evaluation and found Hera is effectiveness in detecting CC issues in terms of both precision and recall. Hera could also detect and fix real-world CC issues from GitHub and Stack Overflow.

The source code of Hera and experiment data are publicly available at ***https://github.com/cse0001/Hera***.

2 BACKGROUND

All CC issues involve two important stages of using a package: *installing* and *importing* the package. In this section, we will introduce the installing and importing strategy of Python packages.

2.1 The Installing Strategy of Python Packages

In OS distributions like Ubuntu, users can use either *apt* or *pip* to install Python packages. The most important step of these tools is to find appropriate versions of the required packages (as well as their dependencies) that satisfy given version constraints. This process is also called dependency resolving [49]. On one hand, *apt* resolves dependency within its own directory and will install some specific versions if not found in the directory. The versions are pre-defined corresponding to the version of OS distribution [24]. On the other hand, *pip* resolves dependency by scanning all system-level Python directories [16], and a) if the required package is not found in any directory, *pip* will by default install the latest version; b) if the package is found and the version satisfied constraints, *pip* will do nothing; c) if the package is found but the version is inappropriate, *pip* will either replace it with an appropriate version (when the package is in *pip* directories) or only install an appropriate version without removing the old one (when the package is in other directories).

```
1.'/usr/lib/python38.zip',
2.'/usr/lib/python3.8',
3.'/usr/lib/python3.8/lib-dynload',
4.'/home/xxx/.local/lib/python3.8/site-packages',
5.'/usr/local/lib/Python3.8/dist-packages',
6.'/usr/lib/python3/dist-packages'
```

Figure 2: The default directory order scanned by Python interpreter in Ubuntu.

In conclusion, *apt* installs a package without considering if *pip* has already installed it. As a result, it is common to have different versions of a package in different directories when using *apt* and *pip* alternately,

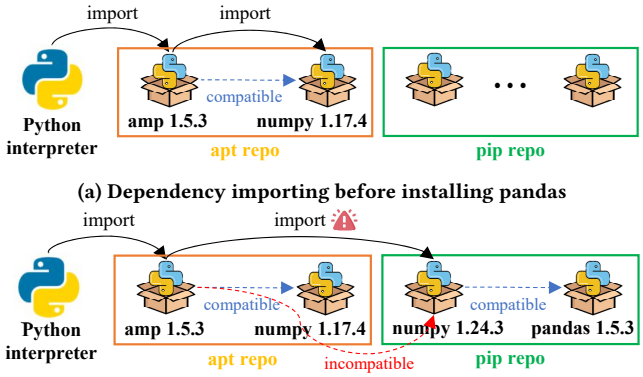
2.2 The Importing Strategy of Python Packages

When importing a package, Python interpreter initiates a sequential search process through various package directories to find the package [16]. Users can view all directories used by the interpreter through *sys.path*. Figure 2 shows the default search order of Python interpreter in Ubuntu system. The first two are Python standard library directories (line 1-2), which store the compressed and uncompressed standard libraries, respectively. The third is for dynamically loaded compiled extension modules (line 3). Others are the third-party package directories, including *pip* user-level directory (line 4), the *pip* system-level directory (line 5), and the *apt* system-level directory (line 6). The user-level and system-level directories store packages installed by the normal user and root users. Python interpreter searches for the required packages by names in the order of the directories with a *find and import* strategy, i.e., not searching further directories if the package has already been found. If not found, the interpreter will report an *ImportError*.

In conclusion, Python interpreter prioritizes importing packages from *pip* directories over *apt* directories without considering the versions by default. As a result, it could be possible to import an incorrect version when the system having multiple versions.

3 MOTIVATION

In this section, we first introduce the reason why both *apt* and *pip* are necessary for installing Python packages, then analyze the root causes of widely presented CC issues, finally summarize the characteristics of CC issues.



(a) Dependency importing before installing pandas

(b) Dependency importing after installing pandas

Figure 3: The root cause of the example CC issue

3.1 Both Repositories Play Significant Roles

The PyPI [18] repository hosts hundreds of thousands of Python packages, each of which contains all history versions. The repository provides *pip* [15], one of the most popular tool for users to install Python packages with significant flexibility. Besides, the Ubuntu repository also provides thousands of Python packages, which can be managed by *apt*. These packages are different from the ones of the PyPI repository in three aspects:

- **Providing unique packages:** The Ubuntu repository provides some packages which are not in the PyPI repository (e.g., *fail2ban*, a security tool [4]), whereas their dependencies may be hosted in both repositories (e.g., *whois* [22], which is depended by *fail2ban*).
- **Ensuring security and stability:** Python packages from *apt* are usually experienced rigorous audit and their versions are carefully selected to match the Ubuntu distribution version. These manual efforts can ensure the system security and stability.
- **Handling non-Python dependencies:** Some Python packages may depend on non-Python system libraries (e.g., lxml [19]). The Ubuntu repository provides packages of many programming language, which can only be managed by *apt*.

Therefore, *apt* and *pip* can provide stability and flexibility for users when installing Python packages, respectively. Thus, it is a common practice for users to use both tools alternately.

3.2 Cross-Repository Issues are Widely Present

Each repository has its own management tool with unique package installing strategy and directory. The tool is often, if not all the times, well-designed to handle intra-repository dependencies without considering inter-repository dependencies. At the same time, Python interpreter will import packages from both repositories. Even if *apt*, *pip* and Python interpreter all work as expected, users may still suffer from compatibility issues like the example in Figure 1. Figure 3 shows the root cause of the example. When installing "`amp`" using *apt*, Python interpreter will import "`numpy 1.17.4`" from the *apt* directory as a dependency of "`amp`" as shown in Figure 3a. After using *pip* to install "`pandas`", the importing dependency is shown as Figure 3b. At this time, due to the import rules of Python interpreter, "`numpy 1.24.3`" from the *pip* repository is regarded as a dependency for "`amp`". However, "`numpy 1.24.3`" has removed the *numpy.typeDict* attribute, leading to the CC issue.

Table 1: Three triggering patterns leading to CC issues

Index	Command Pattern	Description
1	*pip install B* *apt install A*	The default installed version of *B* is incompatible with *A*.
2	*apt install A* *pip install B==version*	The *version* of *B* is incompatible with *A*.
3	*apt install A* *pip install C*	The version of *B* installed with *A* does not satisfy the constraints of *C*, thus *pip* installs a new one.

Assuming there are packages *A*, *B*, *C*, where both *A* and *C* depend on *B*.

The normal use of *apt*, *pip* and Python interpreter could lead to CC issues like Figure 1. As a result, users may suffer from the issues frequently. It is, however, hard to blame any of the *apt*, *pip* or interpreter alone, since they all work as expected. The root cause of CC issues is that *apt* and *pip* do not consider the global dependency relationships within the system, and the dependencies imported by Python interpreter during execution differ from those resolved by the package managers.

3.3 Characteristics of Cross-Repository Issues

Based on the above study of root cause, we summarize **three triggering patterns** of installation commands leading to CC issues as shown in Table 1. In these patterns, *apt* always installs a compatible version of *B* with regard to *A*, whereas *pip* always installs an incompatible version. When Python interpreter imports *A* from the *apt* repository, it will import the incompatible version of *B* from the *pip* repository, and thus triggers compatibility issues.

According to the above analysis, we can further summarize the following **four failure symptoms** of CC issues: a) the application package is always in the *apt* directory; b) the library package is always in both the *apt* and *pip* directories; c) the application package is compatible with the library package in *apt* directory; d) the library package hosted in *apt* directory is incompatible with the same package in *pip* directory.

These patterns and symptoms can guide the design and evaluation of Hera: a) we use the installation patterns to build a dataset of real-world CC issues during evaluation; b) we use the symptoms to build the dependency and compatibility tables during the design of Hera. The fact that "there is always a compatible package in *apt* directory" can help Hera fixing CC issues.

4 APPROACH

In this section, we described the Hera approach, which is designed to detect and predict CC issues, and provide fixing advice for users. The overview of Hera is shown in Figure 4. Hera mainly includes an offline and an online phases. The offline phase constructs a cross-repository compatibility database including two main tables: dependency table and compatibility table (See Section 4.1). The online phase constructs a system-level package dependency graph (S-PDG) to detect, predict CC issues and provide fixing advice (See Section 4.2).

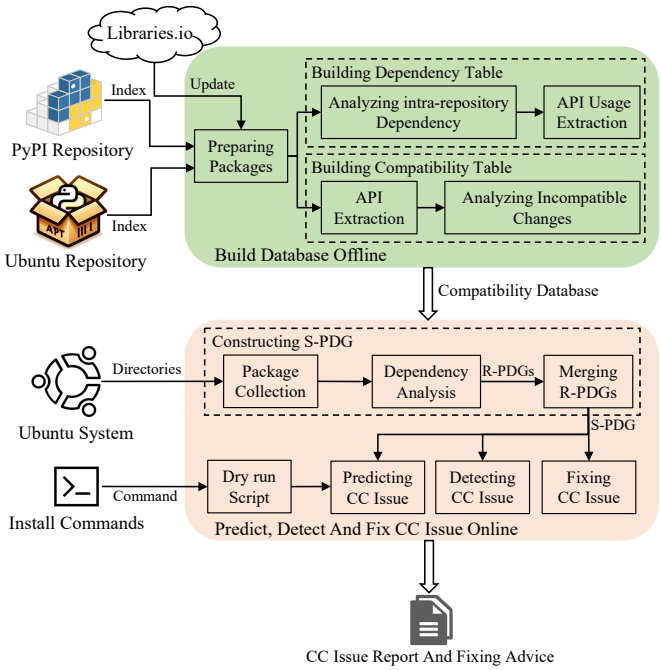


Figure 4: Overview of Hera.

4.1 Building Compatibility Database

As Hera is designed to work in the user's production environment, its overhead should be important. However, online analyzing compatibility between all application and library packages requires high overhead. To overcome this limitation, based on the CC issue characteristics in Section 3.3, we transform the compatibility issue between application package and library package into the compatibility issue of the same library package hosted in the *apt* repository and the *pip* repository. For example, in Figure 1, there is a CC issue between the application package "`amp`" and the library package "`numpy`". Since "`amp`" is compatible with the library package "`numpy 1.17.4`", we can determine the compatibility between "`amp`" and "`numpy 1.24.3`" by analyzing the API compatibility between "`numpy 1.17.4`" and "`numpy 1.24.3`' and whether "`amp`" calls any breaking APIs.

To achieve this, we build a cross-repository compatibility database offline, which can prevent the online phase from complex program analysis. The compatibility database includes a dependency table and a compatibility table. For each application package in the *apt* repository, the dependency table collects its API usages of each library package in the *apt* repository. For each library package in the *apt* repository, the compatibility table collects its API compatibility with each version of the same package hosted in the *pip* repository. The build process first prepares the application packages and library packages from the packages indexed in the two repositories (See Section 4.1.1), then builds the dependency table through API usage extraction (See Section 4.1.2), finally builds the the compatibility table through compatibility analysis (See Section 4.1.3).

4.1.1 Preparing Packages. First, we collected application packages from the *apt* repository. Ubuntu 20.04 maintains a total of 3,319

Dependency Table

Application Package	Library Package	API Usage
ase	scipy	……
	numpy	……
amp	ase	……
	numpy	……
	scipy	……
scipy	numpy	typeDict
		……
……		

Compatibility Table

Library Package	*Apt* version	*Pip* version	Compatibility	Breaking API	Pattern
numpy	1:1.17.4-5ubuntu3	……			
		1.17.4	True	None	
		……			
		1.24.3	Fasle	typeDict	API removal
				……	
……					

Figure 5: Examples of dependency and compatibility tables.

Python packages. We used *apt* to obtain the source code for all packages, which would be used for API usage extraction. Then, we analyzed whether each application package is also hosted in the *pip* repository. The central repository for *pip* is PyPI [18], which contains over 400,000 third-party Python packages. Specifically, we retrieved the installable versions for each package from PyPI. If an installable version exists, we considered the package to be a library package and use *pip* to download the source code for all versions of the package. However, due to different naming policy in the two central repositories, the same Python package may have different names in the two repositories. To solve this problem, we obtained the top-level module name for each application package and used it to match packages in PyPI, thus avoiding omissions. In this way, we totally identified 2,419 library packages and all their installable versions in PyPI.

Since these library packages keep evolving asynchronously, in PyPI, the database has to keep updating accordingly. However, manual updating is impractical because it can not keep up with the pace of PyPI updates. To solve this problem, we resort to *Libraries.io* [8], a platform that monitors repository updates from 32 ecosystems including PyPI. Based on this, we can incrementally fetch the latest package versions and update the compatibility database timely.

4.1.2 Building Dependency Table. The dependency table stores all application packages' API usages of library packages in the *apt* repository. The storage format of the dependency table is shown in Figure 5. The data retrieval process is divided into two steps:

Step 1: Analyzing intra-repository dependencies of apt repository. All application packages in the *apt* repository depend on a pre-defined version of library packages in the *apt* repository. Therefore, analyzing the intra-repository dependencies of *apt* requires retrieving the library packages that each application package depends on. However, the *apt* package documentation categorizes dependencies into *Depends*, *Recommends*, and *Suggests*. *Depends* are definite dependencies, but application packages may also depend on the other two category packages. This makes it difficult to accurately determine dependencies from the documentation. To solve thisi issue, we installed each application package in a clean environment to obtain accurate dependencies, ensuring the correct analysis of the intra-repository dependencies of *apt* repository.

Step 2: Extract all API usages of the library packages. Hera extracted API usage through static analysis. Specifically, for each application package and dependent library package, we first traversed the source code files of the application package to check if the library package is imported. If yes, Hera would parse the code to abstract syntax tree (AST), and scan AST nodes to detect

Table 2: Rules for determining incompatible changes

Index	Change Pattern	Rule	Frequency	IC Type
1	API_addition	$API \notin API_{old} \wedge API \in API_{new}$	32,674,616 (77.69%)	*BIC*
2	API_removal	$API \in API_{old} \wedge API \notin API_{new}$	8,831,893 (21.00%)	*FIC*
3	Param_addition	$API \in API_{old} \cap API_{new} \wedge p \notin P_{old} \wedge p \in P_{new} \wedge p \notin OP_{new}$	102,490 (0.24%)	*BIC/FIC*
4	Param_removal	$API \in API_{old} \cap API_{new} \wedge p \in P_{old} \wedge p \notin OP_{old} \wedge p \notin P_{new}$	71,221 (0.17%)	*BIC/FIC*
5	Optional_param_addition	$API \in API_{old} \cap API_{new} \wedge p \notin OP_{old} \wedge p \in OP_{new}$	328,988(0.78%)	*BIC*
6	Optional_param_removal	$API \in API_{old} \cap API_{new} \wedge p \in OP_{old} \wedge p \notin OP_{new}$	46,275(0.11%)	*FIC*
7	Param_reordering	$API \in API_{old} \cap API_{new} \wedge rp(p, P_{old}, OP_{old}) \neq rp(p, P_{new}, OP_{new})$	3,973 (0.01%)	*BIC/FIC*

"API" denotes a single API and "$API_{old/new}$" denote the API set. "p" denotes a single parameter, "$P_{old/new}$" denote the necessary parameter list and "$OP_{old/new}$" denote the optional parameter list. $rp(p, P_{old/new}, OP_{old/new})$ denotes the relative position of "p" in the parameter list.

the usage of the library package's API. If detected, the application package, the library package, and the used API would be recorded in the dependency table. If not detected, it was because the application package and the library package are indirectly dependent, which would also be recorded in the dependency table.

4.1.3 Building Compatibility Table. The compatibility table stores the API compatibility between each library package and each version of the same package in the *pip* repository. The storage format of the compatibility table is shown in Figure 5. If there are incompatible changes in the APIs of different versions of the library package, Hera will record the incompatibility between them and all the incompatible APIs in the compatibility table. To achieve this goal, Hera perform the following two steps:

Step 1: API extraction. To accurately analyze the compatibility between different versions of library packages, it is first necessary to obtain all APIs of each version of the library package. For most static programming languages, static analysis can effectively extract the APIs provided by packages. However, Python is a dynamic programming language, and some APIs are created and modified by Python objects at runtime, which cannot be obtained through static analysis. To overcome this problem, we employ a strategy that combines dynamic and static analysis to extract APIs, covering classes, functions (and their parameters), and attributes, etc. For dynamic analysis, we use the *inspect* module for reflective dynamic API extraction after importing Python packages. The *inspect* module is able to dynamically access the internal structure of live objects, thus extracting APIs created and modified at runtime by Python objects. Additionally, we use the *parso* [13] module for static analysis to supplement the extraction of function definitions and other information.

Step 2: Analyzing incompatible changes. Correctly analyzing incompatible changes (IC) between different versions of a library package is a guarantee of effectiveness of Hera. There are two types of incompatible changes including forward incompatible changes (FIC) (e.g., adding an API) and backward incompatible changes (BIC) (e.g., removing an API) [33]. Since the version order of library packages in the *apt* repository and the *pip* repository in CC issues is not necessarily sequential, both types of incompatible changes can lead to package incompatibility and may cause CC issues. The target of incompatible change analysis, Python, has flexible syntax, so the complexity of its API changes far exceeds that of languages like C++ and Java. Therefore, it is difficult to determine which API changes are incompatible.

To address this issue, inspired by existing work [56], we consider both BIC and FIC, summarize seven patterns of incompatible changes, and design determining rules. The patterns and rules are shown in Table 2, and the frequency represents the proportion of each pattern in the compatibility table. For API-related changes, we detect the addition or deletion of classes, functions, and attributes within a Python program based on the results of the API extraction step. For API parameter-related changes, we consider the changes in required and optional parameters as well as the changes in parameter order. It is noteworthy that a parameter changing from required to optional (or vice versa) will be detected by Rule 3 (4). We refer to the API with incompatible changes as a breaking API. The compatibility table will store the breaking API information of each incompatible change, including name, parameters, type, and change pattern, etc.

Based on the above two steps, in the Ubuntu 20.04 and Python 3.8.10 environment, we extract 62,552,811 APIs across 47,387 versions of 2,419 library packages and found a total of 42,059,456 incompatible changes.

4.2 Detecting, Predicting And Fixing CC Issues

A CC issue involves interactions of the above three parts, i.e., *apt*, *pip* and Python interpreter, making it non-trivial to be predicted, detected or fixed. To address this challenge, we propose a system-level package dependency graph (S-PDG) to describe the interactions among *apt*, *pip* and Python interpreter. In this section, we mainly explain how to construct the S-PDG (See Section 4.2.1 and use it along with the cross-repository compatibility database to detect CC issues (See Section 4.2.2). We also explain how to dry run commands to predict CC issues (See Section 4.2.3) and provide users advice for fixing CC issues (See Section 4.2.4).

4.2.1 Constructing S-PDG.. First, Hera will collect packages and analyze dependency in the installing directory of *apt* and *pip*, and construct two repository-level package dependency graphs (R-PDG). Then, Hera will merge the two R-PDGs into one S-PDG and detect CC issues. The construction of the S-PDG is divided into the following three steps:

Step 1: Collect packages in the apt and pip directory. For the *apt* directory, Hera uses the *dpkg* [3] tool, a low-level package manager for Debian-based systems, to collect all packages in the *apt* repository based on the package installation records. *Pip* provides a similar function, however, it can only collect packages from all system-level directories and can not specify to collect packages from

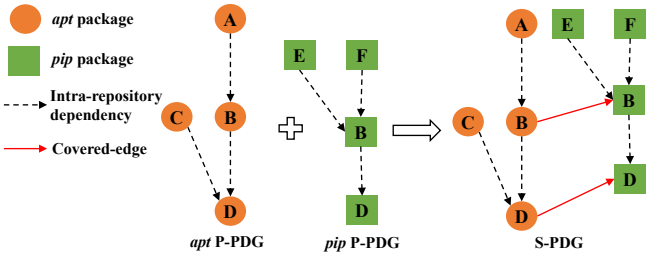


Figure 6: Merge two R-PDGs to construct one S-PDG.

the *pip* directory. To solve this issue, Hera uses the *pkg_resources* module to collect all packages from the *pip* directory based on the directory path.

Step 2: Analyze dependency and construct R-PDGs. In this step, Hera analyzes the intra-directory dependencies of *apt* and *pip* and constructs two repository-level package dependency graphs (R-PDG). The R-PDG is a directed acyclic graph where nodes represent packages and edges represent the internal dependencies between packages within the directory, which are regarded as compatible in this paper. For the *apt* directory, we construct the R-PDG based on the intra-repository dependencies of the *apt* repository obtained in section 4.1.2. For the *pip* directory, we employ pipdeptree [17], a dependency analysis tool, to analyze the internal dependencies of the *pip* directory and construct the R-PDG.

Step 3: Merge R-PDGs to construct S-PDG. Hera constructs the system-level package dependency graph (S-PDG) by merging the two R-PDGs. Figure 6 shows a schematic figure of merging two R-PDGs to construct one S-PDG, where packages B and D each have a version in both the *apt* and *pip* directories. When a package has two versions in the two R-PDGs respectively, Hera adds an edge to connect those two versions, meaning that the version in *apt* is *covered* by the version in *pip*. This edge directs from a node in the *apt* R-PDG to a node in the *pip* R-PDG, and is referred to as a covered-edge. The covered version will not be used any more according to the importing rules of Python interpreter.

4.2.2 Detecting CC Issues. After constructing the S-PDG, we can detect CC issues in the users' system environment. To achieve this, Hera queries the compatibility table to determine if there are breaking APIs between the two versions connected by each covered-edge (e.g., between two versions of package B and D in Figure 6). If yes, it means the users' system contains incompatible versions of a library package, which may lead to potential CC issues. To confirm the potential issues, Hera queries the dependency table to check if there is an application package in the *apt* directory uses the breaking API (e.g., checking package A and C in Figure 6). If yes, Hera reports a CC issue.

4.2.3 Predicting CC Issues. For prediction, Hera dry-runs the installation command, and then temporarily adds the packages to be installed into the S-PDG. Specifically, for each *apt* or *pip* installation command, Hera uses the "apt install -s" or "pip install –dry-run" command to preview the packages that will be installed. Subsequently, Hera temporarily adds these packages to the S-PDG and checks whether a new covered-edge should be added. If there is a new covered-edge should be added, Hera queries the compatibility and dependency tables to detect if a new CC issue will arise. If so, Hera warns the user that executing the installation command will cause a CC issue, and reports detailed information about the predicted CC issue, such as the packages at both ends of the covered-edge. This prediction mechanism effectively prevents the introduction of CC issues during the installation process.

Table 3: Statistics of CC Issue Dataset

Group	Pkg	Avg. dep	Pair	Crash	Breaking API
G1 [1,5]	1,048	2.24	2,234	89	82
G2 [6,10]	282	7.80	2,189	261	255
G3 [11,20]	218	14.60	2,322	91	89
G4 [21,49]	211	31.96	6,751	677	674
G5 [50,+∞]	158	89.84	10,370	574	574
Total/Average	1,917	14.96	23,866	1,692	1,674

4.2.4 Fixing CC Issues. When a CC issue is detected, as R-PDGs contain compatible dependencies, Hera can provide fixing advice accordingly. There are generally two solutions to fix CC issues: **a)** breaking the covered-edge in the S-PDG by deleting and reinstalling packages, **b)** specify importing compatible library package from the *apt* directory. Since solution **a** might cause other system-level dependency issues, therefore, we recommend using solution **b** to fix the CC issue. Based on the R-PDG of the *apt* directory, Hera will advise users to use *importlib* module to import compatible library packages from the *apt* directory before importing the application package.

5 EVALUATION

We study three research questions in our evaluation section:

- **RQ1 (Detecting CC issues):** ***How effective is Hera in detecting CC issues?*** To anwser RQ1, we constructed a large-scale dataset of CC issues within the Ubuntu 20.04 system to evaluate the effectiveness of Hera, and confirmed its reliability through a sampling-based manual verification process.
- **RQ2 (Analyzing Incompatible Changes):** ***How effective is Hera in analyzing incompatible changes?*** To answer RQ2, We extracted breaking APIs from each CC issue's *traceback* in the dataset, thus evaluating the recall of analyzing incompatible changes, and additionally evaluating the precision through sample validation.
- **RQ3 (Fixing Real-world Issues):** ***Can Hera detect real-world CC issues and provide effective fixing advice?*** To answer RQ3, We collected 27 real-world CC issues from Stack Overflow and GitHub to evaluate whether Hera could detect and fix real-world CC issues.

5.1 Dataset Construction

We first constructed a real-world dataset of CC issues in the Ubuntu 20.04 system, and the construction process involves two steps:

- ***Step 1: Identifying application and library pairs.*** To identify real-world CC issues, it is first necessary to identify pairs of application packages and library packages that may encounter CC issues. We use *(A, B)* to refer to the pair in the following part. We identify pairs based on the following two conditions:

```
1.$ pip install -y B
2.$ apt install -y A
3.>>> import A
4.$ pip freeze | xargs pip uninstall -y
5.>>> import A
```

Figure 7: Commands for testing CC issues.

- ***Condition 1:*** Package *A* should be an application package and depend on at least one other application package;
- ***Condition 2:*** Package *B* should be hosted in both *apt* and *pip* repositories, and should be a direct dependency or transitive dependency of *A*.

Based on the *apt* intra-repository dependency obtained in Section 4.1.2 and ***Condition 1***, we identified 1,917 application packages that depend on at least one other application. Based on the library packages obtained in Section 4.1.1 and ***Condition 2***, we identified 23,866 pairs that may encounter CC issues. To assess the impact of the number of dependencies of the application package on CC issue occurrence, we categorized these application packages into five groups based on their number of dependencies.

- ***Step 2: Testing for real-world CC issues.*** To test whether each package pair will encounter a CC issue, we used **triggering pattern 1** in Section 3.3 to design testing commands shown in Figure 7. We used Docker [2] to execute the commands, ensuring a clean system environment. First, we install *B* and *A* using *pip* and *apt* respectively (line 1-2). Second, we import *A* in the Python interpreter (line 3). If it crashes, we will uninstall all packages in the *pip* directory and imported *A* again (line 4-5). If it imports normally, it means *B* makes *A* fail to be imported and there is a CC issue with *(A, B)*. Additionally, we also analyzed *traceback* information to extract the breaking API that caused the crash.

Overall, we tested all 23,866 pairs and identified 1,692 pairs (7%) with CC issues. The statistics of CC issue dataset are shown in Table 3, where application packages with more dependencies are relatively more prone to CC issues. And we successfully extracted 1,674 (98.9%) breaking APIs from the *traceback*. Among the 18 (1.1%) failures, 11 cases were due to the incompatibility between the package and the Python interpreter, and 7 cases lacked clear breaking API information in the *traceback*.

5.2 RQ1: Detecting CC Issues.

Study Methodology. In this study, we employed Hera to detect CC issues across all 23,866 package pairs in our dataset. Each pair *(A, B)* consists of application package *A* (a pre-defined version from *apt* repository) and library package *B* (the default version installed via *pip*). It's crucial to understand that Hera does not delve into the API call chain, meaning that if *B* is a transitive dependency of *A* without a direct call, this might lead to false negatives. However, if a dependency *C* of *A* calls a breaking API in *B*, the installation of *A* and *B* can also introduce a CC issue into the system. Thus, if a CC issue is identified between *C* and *B*, we consider there is a CC issue between *A* and *B* as well. We used sample manual verification to check the precision of identified issues. For detected issues, we manually verified their correctness. The sample size was calculated through the finite population correction, ensuring our sample accurately represents the dataset. We set the confidence level to 95% and the margin of error to 5%, which are standard statistical thresholds.

Table 4: Effectiveness of Hera in detecting CC issues.

Group	Pair	*VTP*	*FP*	*DTP*	*FN*	*Pre.*	*Rec.*	*F1 Score*
G1	309	27	2	55	34	93.1	61.8	74.3
G2	472	44	1	225	36	97.8	86.2	91.6
G3	327	27	4	81	10	87.1	89.0	88.0
G4	1,214	100	14	669	8	87.7	98.8	92.9
G5	1,367	117	12	555	19	90.7	96.7	93.6
T/A	3,689	315	33	1,585	107	90.5	93.7	92.1

Evaluation Metrics. Leveraging the CC issues dataset we constructed, we considered seven metrics in our evaluation: (1) ***Verified True Positive (VTP)***: CC issues reported by Hera and passed manual validation; (2) ***Direct True Positive (DTP)***: CC issues reported by Hera and in the dataset; (3) ***False Positive (FP)***: CC issues reported by Hera that did not pass manual validation; (4) ***False Negative (FN)***: CC issues in the dataset but not reported by Hera.

Based on the above four metrics, we can calculate *Precision*, *Recall*, and *F1 score* as follows: (5) ***Precision*** = *VTP / (VTP + FP)*; (6) ***Recall***: *DTP / (DTP + FN)*; (7) ***F1 score***: *2 × Precision × Recall / (Precision + Recall)*. *Precision* evaluates whether Hera can precisely detect CC issues. *Recall* evaluates the capability of Hera to detect all CC issues. *F1 score* combines *Precision* with *Recall* [38, 39].

Result. Using the above methodology, Hera reported a total of 3,689 CC issues. It should be noted that many issues Hera reported were not identified during CC issue dataset construction because our initial compatibility tests only involved importing library packages, that might not trigger all breaking APIs. From these reported issues, we sampled 348 cases to manually verify. Table 4 shows the experiment results of RQ1. Hera detected CC issue with a *Precision* of 90.5%, a Recall rate of 93.7% and a F1 score of 92.1%. Based on the results, we could conclude that Hera can effectively detect CC issues.

FP Analysis. We summarize three main reasons for *FPs*:

- **(a) Non-use of optional parameters or extra parameter list (17/33):** In the evolution of third-party packages, the removal of optional parameters or extra parameter lists from functions is commonly regarded as a BIC. However, when such function calls only use required parameters, they are not affected by this type incompatible changes. When detecting CC issues, Hera queries the dependency table only to check for breaking APIs usage without considering parameter usage, leading to *FPs*.
- **(b) Breaking function API used for assignment (10/33):** When the breaking function API is used for assignment without being called, it would not trigger CC issue. Currently, the dependency table only stores API usage without considering the type of usage, leading to *FNs*. For example, in application package "`partd 1.11.1`" [14], the code *decompress_bytes = blosc.decompress* assigns the breaking API *blosc.decompress* to *decompress_bytes* without calling it, thereby not causing a CC issue.
- **(c) Incorrect breaking APIs (6/33):** In some versions of some library packages, API extraction fails to capture all provided APIs or extracts API names that do not match the actual API call names. This makes the compatibility table stores incorrect breaking APIs and leads to these *FPs*. For instance, the library package

"humanfriendly 10.0" [7] identified an API name as *humanfriendly.model.coerce_boolean* when API extraction, whereas the actual API call is *humanfriendly.coerce_boolean*.

FN Analysis. Hera produced a total of 107 *FNs*, 59 of which were due to the dependency table not storing the usage of the corresponding breaking APIs. When extracting API usage, Hera mainly detects AST nodes with frequent API calls, such as *atom_expr* node and *argument* node. The usage of breaking APIs in other nodes caused these *FNs*. Additionally, some breaking API usages do not occur in **.py* files, for example, the breaking API usages between "pipdeptree 0.13.2-1build1" [17] and "pip 23.1.2" [15] occur in **.pyx* files, and Hera is unable to analyze **.pyx* or **.pyc* files. The remaining 48 *FNs* were due to the compatibility table not storing the breaking APIs corresponding to these cases. This was mainly because of failures in API extraction or the determining rules not covering all incompatible change scenarios during the analysis of incompatible changes.

Conclusion: Hera can effectively detect CC issues, achieving a precision of 90.5% and a recall rate of 93.7%.

5.3 RQ2: Analyzing Incompatible Changes.

Study Methodology. RQ2 aims to evaluate the compatibility table, the core component of Hera, focusing on the effectiveness of analyzing incompatible changes. To evaluate the precision, we performed stratified random sampling of breaking APIs according to the proportion of each incompatible changes pattern in the compatibility table, using the same sample size calculation method as RQ1. For each sampled breaking API, we installed the corresponding *apt* version and *pip* version of the library package, and manually verified whether calling the breaking API would cause compatibility issue. To evaluate recall rate, we verified whether the breaking APIs in CC issues dataset were present in the compatibility table.

Evaluation Metrics. We measure six metrics: (1) ***Verified True Positive (VTP)***: Breaking APIs sampled from the compatibility table and passed manual validation; (2) ***Direct True Positive (DTP)***: Breaking APIs in the CC issues dataset and also in the compatibility table; (3) ***False Positive (FP)***: Breaking APIs sampled from the compatibility table that did not pass manual verification; (4) ***False Negative (FN)***: Breaking APIs in the CC issues dataset but not in the compatibility table. Based on these metrics, we calculate *Precision* and *Recall* in the same manner as with **RQ1**. The *Precision* evaluates whether Hera can accurately analyze incompatible changes, and the *Recall* assesses whether Hera can analyze all incompatible changes.

Result. Figure 8 shows the experiment results of RQ2. Among 384 breaking APIs sampled for manual validation, 350 were validated as while 34 were not validated, with a precision of 91.1%. The *FNs* came from four categories, including *API_addition*, *API_removal*, *Param_addition*, and *Optional_param_addition*. Among the total of 1674 breaking APIs in the CC issue dataset, 1644 were found in the compatibility table, with a *Recall* of 98.2%. Based on the experimental results, we could conclude that Hera can effectively analyze incompatible changes.

FP Analysis. We analyze *FPs* separately according to IC patterns:

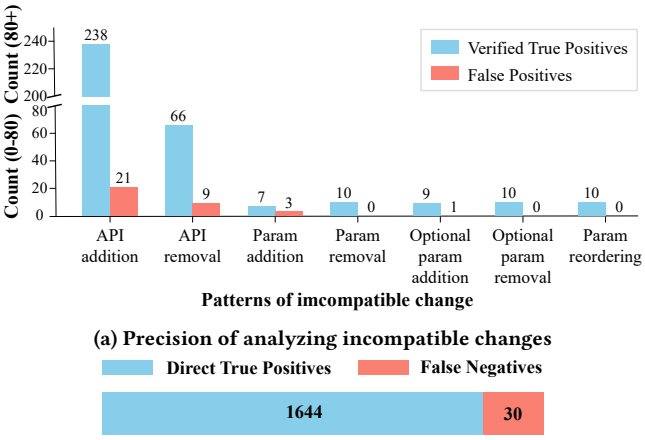


Figure 8: Experiment Results of RQ2

- **(a) API addition and removal (30/34):** The *FPs* in these two categories were primarily due to the API extraction process not extracting all APIs. Hera employs dynamic reflection to extract APIs, which fails when Python interpreter cannot import the module due to missing dependencies. Some versions of library packages do not declare all their dependencies, leading to a failure in automatically installing the necessary dependencies. As a result, modules that rely on these uninstalled dependencies cannot be imported, leading to missing APIs.
- **(b) Parameter addition (3/34):** The three *FPs* in this category occurred because the added parameter was part of an extra parameter list(e.g., *xarray.open_rasterio* in "xarray 0.15.0" and "xarray 2022.10.0" [23]). In our rules, we considered parameter additions to cause both *FIC* and *BIC*, but when the addition is to a parameter list, it only causes *BIC*. We consider this misjudgment of incompatible change types as *FPs*.
- **(c) Optional parameter addition (1/34):** The single FP in the *optional_param_addition* category was also a result of a misjudgment of incompatible change types. This occurred in the constructor of the *marshmallow.fields.Field* class in library package "marshmallow" [9]. The new version modified the class to require keyword arguments and also added optional parameters to the parameter list. Hera reported this as *optional_param_addition* and BIC. However, such changes are considered both BIC and FIC, which led to an *FN*.

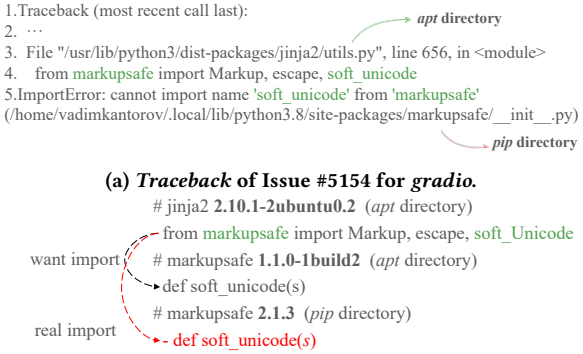
FN Analysis. All *FNs* were caused by omissions of some APIs during the API extraction step when constructing the compatibility table. Predominantly, these omissions were linked to the absence of required dependencies, which led to the failure to import the relevant modules and the inability to dynamically extract APIs. Furthermore, due to the complexity of some library package design and the limitations of the tool design, Hera was unable to extract the relevant APIs.

Conclusion: At a 95% confidence level with a 5% margin of error, Hera demonstrates effectiveness in analyzing incompatible changes, achieving a precision of 91.1% and a recall rate of 98.2%.

Table 5: Experiment results of RQ3

27 Real-world CC Issues
questions#75244186♠■■■; questions#61237965♠■; questions#76383191♠■■■;
questions#72593814♠■■■; questions#74843336♠■■■; questions#74411148♠■■■;
questions#74515643♠■■■; linode_panel#1♣■■■; ethz_piksi_ros#234♣■■■;
gradio#5154♠■■■; frr#14816♣■■■; In0ri#10♣■■■; gramine#1142♣■■■;
gsc#121♠■■■; PyDodge#24♣■■■; kasm-dof-workspace#9♣■■■; linkml#966♣■■■;
quart#171♠■■■; LittlePaimon#283♣■■■; fprime-tools#48♣■■■; taipy#255♣■■■;
stable-diffusion-webui#231♠■■■; mkchromecast#451♣■■■; InstallScript#396♣■■■;
cserbot#9688♣■■■; fact_CORE#858♣■■■; frr#14820♢■■■;

♠: CC issues from Stackoverflow; ♣: CC issues from Github Issue;
♢: CC issues from Github Discussion; ■: CC issues successfully reproduced;
■: CC issues unsuccessfully reproduced; ■: CC issues successfully detected by Hera;
■: CC issues successfully fixed by Hera.



(a) *Traceback* of Issue #5154 for *gradio*.

(b) API import chain of Issue #5154 for *gradio*.

Figure 9: A real-world CC issue example.

5.4 RQ3: Fixing Real-world Issues.

Study Methodology. RQ3 evaluate the Usefulness of Hera by collecting and reproducing real-world CC issues from well-known Q&A websites or open-source platforms such as Stack Overflow [21] and GitHub [6]. The collection process is divided into two steps:

- ***Step 1: Searching keywords.*** We first used keywords such as "Ubuntu", "ImportError", and "dist-packages" to preliminarily filter out compatibility issues related to third-party packages on the Ubuntu system.
- ***Step 2: Verifying traceback information.*** The most direct distinction between CC issues and traditional incompatibility issues is that CC issues occur between *apt* and *pip* directories. Therefore, we employed an engineer with over five years of Python programming experience manually reviewed the *traceback* information for each issue to determine whether it is a CC issue.

Based the above two steps, we identified 27 real-world CC issues and we tried to reproduce them in the system. For issues that were successfully reproduced, we used Hera to detect CC issues and followed the advice provided by Hera to fix CC issues.

Result. Based the above method, we successfully reproduced 26 CC issues and Hera was able to detect all of them. Additionally, we followed the advice to import the library package in the *apt* directory before importing the application package and successfully fixed all CC issues. All issues are presented in Table 5. The issue that could not be reproduced was question #61237965 [20], due to insufficient environmental information reported by the developer.

Example. Many CC issues have troubled developers within the community, and Issue #5154 for *gradio* [29] is an example. The developer faced a compatibility issue between the application package "`jinja2`" and the library package "`markupsafe`" when running the *gradio* program. Figure 9a shows the critical part of the *traceback* information for this issue, illustrating that "`jinja2`" from the *apt* directory encountered an error while attempting to import the *soft_unicode* function from "`markupsafe`" in the *pip* directory. Further, figure 9b details the API import chain, showing that "`jinja2 2.10.1-2ubuntu0.2`" in *apt* directory tried to import *soft_unicode* from "`markupsafe 1.1.0-1build2`" in the *apt* directory, but the Python interpreter imported the function from "`markupsafe 2.1.3`" in the *pip* directory. But the *2.1.3* version deleted the attribute, which led to the issue. The developers ultimately resolved the issue by downgrading the library package "`markupsafe`" in the *pip* directory to a version supporting *soft_unicode*. Hera advised specifying to import the library package "`markupsafe`" from the *apt* directory, which could help users avoid reinstalling a compatible version of "`markupsafe`" in the *pip* directory and solve this issue more easily. Additionally, the package manager did not provide any warning to users about this issue, while Hera could predict this issue when package installation.

Conclusion: Among the 26 real-world CC issues reproduced, Hera can detect all of the 26 cases, and provide accurate reasons as well as effective fixing advice.

6 DISCUSSION

In this section, we discuss the limitations and generalizability of this work.

Limitations. Firstly, Hera detects CC issues and advises users to import compatible library packages from the *apt* repository before importing the application package. However, the user program that depends on the application packages may not necessarily be affected by the CC issue. This is because the user program may not call the breaking API. To determine whether the user program will encounter errors due to the CC issue, it is necessary to perform online program analysis to obtain the API call chain. Currently, our work does not perform online program analysis in order to reduce deployment overhead. Secondly, some cross-repository compatibility issues are related to native modules and binary libraries, which Hera cannot currently address. This is because Hera's current program analysis focuses on the Python language and cannot analyze the API compatibility of such modules. Our work focuses on solving cross-repository third-party package compatibility issues at the Python source code level, which has already covered the majority of incompatibility issues encountered by users. We will consider the above limitations in our future work to optimize Hera, and our technique should be evaluated over time by using it in real-world development environments. In the future, we will conduct user studies to enhance the usability of Hera.

Generalizability. Evaluation results show that Hera is effective in solving the CC issues between the Ubuntu system and the Python language. In our research, we also found similar CC issues in other Linux distribution ecosystems and other dynamic programming language ecosystems (e.g., CentOS, Ruby), and we found the root causes of such issues are consistent. Therefore, the overall

framework of our work is applicable to other ecosystems. However, the methods for analyzing package compatibility are not generally applicable to packages in different languages. We determine compatibility by analyzing incompatible APIs, but since the way APIs are implemented varies between languages, it is difficult to create a general method for analyzing the compatibility of third-party packages in all languages. In the future, we will explore more general methods for analyzing compatibility to address CC issues in additional ecosystems.

7 RELATED WORK

Dependency Conflict Diagnosis. Many works focus on the Dependency Conflict (DC) issues. The most relevant works to this paper are smartPip [49], EasyPip [36], WatchMan [52], PyDFix [43], and SnifferDog [51]. WatchMan [52], samrtPip [49], and EasyPip [36] aim to solve Python DC issues that cannot be solved by the most popular Python package manager, *pip*. WatchMan first summarizes the manifestation patterns of DC issues caused by *pip* installation rules and proposes corresponding detection tools. SamrtPip reveals new features of DC issues under the new dependency resolution strategy released after Watchman. Their approach can also solve DC issues without changing version constraints. EasyPip is used to automatically detect and fix issues in Python dependency declaration files, which is more helpful than SamrtPip in locating the root cause of DC issues. However, they are limited to a single software repository and do not take into account DC issues that are affected by multiple package managers and cannot solve such issues. PyDFix [43] aims to solve the issue that some packages on PyPI are not able to be installed correctly due to environment compatibility issues. However, cross-repository compatibility issues often do not occur during the installation phase but during the runtime phase of the project. SnifferDog [51] fixes the environment of a Jupyter Notebook project with module information and dependency information, but does not address the system Python environment. In addition to these, many works focus on DC issues in other ecosystems [32, 37, 45, 53–55]. Studies [32, 53–55] mainly focus on dependency management in Java ecosystem. They found DC issues in Maven [10] do not cause build failures, but can lead to inconsistent semantic behavior [55] or runtime exceptions [53]. Based on this finding, some researchers have proposed to use static analysis [53]and dynamic analysis [54, 55] to identify DC issues in Maven. For *.Net* ecosystem, wang et al. conducted an empirical study of DC issues in NuGet [12]. They summarized manifestation patterns and fixing strategies and proposed an effective tool NuFix [37] to fix DC issues by adjusting the version constraints. ConflictJS [45] targets DC issues in the JavaScript ecosystem due to the fact that third-party libraries share the same global namespace. Most of them are limited within a single software repository and do not consider dependency conflicts under the joint influence of multiple repository package managers.

Incompatible Change Detection. API incompatible changes are also known as breaking changes. There are two main methods of detection: testing [28, 40, 42] and program analysis [26, 28, 30, 31, 33, 46, 47, 56]. NoRegrets [40] is a regression testing tool that can be used to determine whether updates to Javascript third-party packages affect the use of the updated API. NoRegrets+ [42] can automatically generate test cases to find incompatible changes in Javascript third-party packages. DeBBI [28] detects incompatibilities between Java packages and projects via testing and analysis. For static programming language, studies [26, 34, 35, 41, 44, 47] focus on analyzing Java API and detecting incompatible changes. For dynamtic programming language, V2 [31] detects incompatible changes based on Python program crash information. Studies [30, 56] use rules to detecting incompatible changes in the Evolution of Python third-party packages. There are also studies that detect incompatible changes at the binary-level [33, 46]. Ponomarenko et al. present a method for automatically detecting backward compatibility issues with third-party libraries at the binary level, available for multiple languages [46]. Jia et al. proposed a more effective binary-level tool Depowl [33] that can detect both forward and backward incompatibility issues. All of the above studies perform well in detecting incompatible changes, however, many of these methods have too much overhead to build databases with multi-million APIs. Therefore, we designed rules to detect most of the incompatible changes with minimal overhead.

8 CONCLUSION

In this paper, we defined the CC issue, and we used both Python and Ubuntu repositories as representatives to study the root causes of CC issues, then summarized their triggering patterns and failure symptoms. Guided by the above analysis, we designed Hera, an automatic tool to solve CC issues. Hera first builds a cross-repository compatibility database offline, and then online predicts, detects and fixes CC issues in the user's system environment. We conducted a comprehensive evaluation of Hera, which showed that Hera could effectively detect CC issues and provided accurate reasons as well as fixing advice. In the future, we plan to deploy Hera in real-world developers' development environments to further evaluate its usefulness.

ACKNOWLEDGMENTS

The authors express thanks to the anonymous reviewers for their insightful comments. This research was funded by NSFC (No. 62272473 and No. 62202474) and the Science and Technology Innovation Program of Hunan Province (No. 2023RC1001 and No. 2023RC3012).

REFERENCES

[1] 2024. apt. https://help.ubuntu.com/lts/serverguide/apt.html.en Accessed: 2024-6-6.
[2] 2024. Docker. https://www.docker.com/ Accessed: 2024-6-6.
[3] 2024. Dpkg. https://wiki.debian.org/Teams/Dpkg/ Accessed: 2024-6-6.
[4] 2024. Fail2ban. http://www.fail2ban.org Accessed: 2024-6-6.
[5] 2024. Fedora. https://fedoraproject.org/ Accessed: 2024-6-6.
[6] 2024. GitHub. https://github.com/ Accessed: 2024-6-6.
[7] 2024. humanfriendly. https://pypi.org/project/humanfriendly/ Accessed: 2024-6-6.
[8] 2024. Libraries.io. https://libraries.io/ Accessed: 2024-6-6.
[9] 2024. marshmallow. https://pypi.org/project/marshmallow/ Accessed: 2024-6-6.
[10] 2024. *Maven.* https://mvnrepository.com/repos Accessed: 2024-6-6.
[11] 2024. npm. https://www.npmjs.com/ Accessed: 2024-6-6.
[12] 2024. NuGet. https://www.nuget.org Accessed: 2024-6-6.
[13] 2024. parso. https://pypi.org/project/parso/ Accessed: 2024-6-6.
[14] 2024. partd. https://pypi.org/project/partd/ Accessed: 2024-6-6.
[15] 2024. pip. https://pypi.org/project/pip/ Accessed: 2024-6-6.
[16] 2024. *pip documentation.* https://pip.pypa.io/en/stable/cli/pip_install/ Accessed: 2024-6-6.
[17] 2024. pipdeptree. https://github.com/naiquevin/pipdeptree Accessed: 2024-6-6.
[18] 2024. *PyPI.* https://pypi.org/ Accessed: 2024-6-6.

[19] 2024. Python3-lxml. https://lxml.de/ Accessed: 2024-6-6.

[20] 2024. questions 61237965. https://stackoverflow.com/questions/61237965/no-module-named-numpy-testing-decorators Accessed: 2024-6-6.

[21] 2024. Stack Overflow. https://stackoverflow.com/ Accessed: 2024-6-6.

[22] 2024. whois. https://pypi.org/project/whois/ Accessed: 2024-6-6.

[23] 2024. xarray. https://pypi.org/project/xarray/ Accessed: 2024-6-6.

[24] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. 2020. Dependency solving is still hard, but we are getting better at it. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 547–551. https://doi.org/10.1109/SANER48275.2020.9054837

[25] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2012. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software* 85, 10 (2012), 2228–2240. https://doi.org/10.1016/j.jss.2012.02.018

[26] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 507–511. https://doi.org/10.1109/SANER.2018.8330249

[27] Yulu Cao, Zhifei Chen, Xiaowei Zhang, Yanhui Li, Lin Chen, and Linzhang Wang. 2024. Diagnosis of package installation incompatibility via knowledge base. *Science of Computer Programming* (2024), 103098. https://doi.org/10.1016/j.scico.2024.103098

[28] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 112–124. https://doi.org/10.1145/3377811.3380436

[29] Gradio App Contributors. 2024. Issue 5154 on gradio-app/gradio. https://github.com/gradio-app/gradio/issues/5154. Accessed: 2024-6-6.

[30] Xingliang Du and Jun Ma. 2022. Aexpy: Detecting api breaking changes in python packages. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 470–481. https://doi.org/10.1109/ISSRE55969.2022.00052

[31] Eric Horton and Chris Parnin. 2019. V2: Fast detection of configuration drift in python. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 477–488. https://doi.org/10.1109/ASE.2019.00052

[32] Kaifeng Huang, Bihuan Chen, Bowen Shi, Ying Wang, Congying Xu, and Xin Peng. 2020. Interactive, effort-aware library version harmonization. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 518–529. https://doi.org/10.1145/3368089.3409689

[33] Zhouyang Jia, Shanshan Li, Tingting Yu, Chen Zeng, Erci Xu, Xiaodong Liu, Ji Wang, and Xiangke Liao. 2021. DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 86–98. https://doi.org/10.1109/ICSE43902.2021.00021

[34] L. Krejci. 2024. revapi. https://revapi.org/revapi-site/main/index.html Accessed: 2024-6-6.

[35] L. K¨uhne. 2024. clirr. https://clirr.sourceforge.net/ Accessed: 2024-6-6.

[36] Shuo Li. [n. d.]. EasyPip: Detect and Fix Dependency Problems in Python Dependency Declaration Files. ([n. d.]).

[37] Zhenming Li, Ying Wang, Zeqi Lin, Shing-Chi Cheung, and Jian-Guang Lou. 2022. Nufix: escape from NuGet dependency maze. In *Proceedings of the 44th International Conference on Software Engineering*. 1545–1557. https://doi.org/10.1145/3510003.3510118

[38] Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At Which Training Stage Does Code Data Help LLMs Reasoning? *arXiv preprint arXiv:2309.16298* (2023). https://doi.org/10.48550/arXiv.2309.16298

[39] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? *arXiv preprint arXiv:2406.01422* (2024). https://doi.org/10.48550/arXiv.2406.01422

[40] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type regression testing to detect breaking changes in Node. js libraries. In *32nd european conference on object-oriented programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2018.7

[41] M. Mois. 2024. japicmp. https://github.com/siom79/japicmp Accessed: 2024-6-6.

[42] Anders Møller and Martin Toldam Torp. 2019. Model-based testing of breaking changes in Node. js libraries. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 409–419. https://doi.org/10.1145/3338906.3338940

[43] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 439–451. https://doi.org/10.1145/3460319.3464797

[44] Oracle. 2024. SigTest. https://wiki.openjdk.java.net/display/CodeTools/SigTest Accessed: 2024-6-6.

[45] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*. 741–751. https://doi.org/10.1145/3180155.3180184

[46] Andrey Ponomarenko and Vladimir Rubanov. 2011. Automatic backward compatibility analysis of software component binary interfaces. In *2011 IEEE International Conference on Computer Science and Automation Engineering*, Vol. 3. IEEE, 167–173. https://doi.org/10.1109/CSAE.2011.5952657

[47] Danilo Silva and Marco Tulio Valente. 2017. Refdiff: detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 269–279. https://doi.org/10.1109/MSR.2017.14

[48] Paulo Trezentos, Inês Lynce, and Arlindo L Oliveira. 2010. Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. 427–436. https://doi.org/10.1145/1858996.1859087

[49] Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu, and Guoqing Li. 2022. smartPip: A Smart Approach to Resolving Python Dependency Conflict Issues. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12. https://doi.org/10.1145/3551349.3560437

[50] Huiyan Wang, Shuguan Liu, Lingyu Zhang, and Chang Xu. 2023. Automatically resolving dependency-conflict building failures via behavior-consistent loosening of library version constraints. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 198–210. https://doi.org/10.1145/3611643.3616264

[51] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring execution environments of Jupyter notebooks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1622–1633. https://doi.org/10.1109/ICSE43902.2021.00144

[52] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: Monitoring dependency conflicts for python library ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 125–135. https://doi.org/10.1145/3377811.3380426

[53] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. 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*. 319–330. https://doi.org/10.1145/3236024.3236056

[54] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could I have a stack trace to examine the dependency conflict issue?. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 572–583. https://doi.org/10.1109/ICSE.2019.00068

[55] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. 2021. Will dependency conflicts affect my program's semantics? *IEEE Transactions on Software Engineering* 48, 7 (2021), 2295–2316. https://doi.org/10.1109/TSE.2021.3057767

[56] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How do python framework apis evolve? an exploratory study. In *2020 ieee 27th international conference on software analysis, evolution and reengineering (saner)*. IEEE, 81–92. https://doi.org/10.1109/SANER48275.2020.9054800