Fixing Dependency Errors for Python Build Reproducibility

Suchita Mukherjee
University of California, Davis
United States of America
sumukherjee@ucdavis.edu

Abigail Almanza
University of California, Davis
United States of America
aalmanza@ucdavis.edu

Cindy Rubio-González
University of California, Davis
United States of America
crubio@ucdavis.edu

ABSTRACT

Software reproducibility is important for re-usability and the cumulative progress of research. An important manifestation of unreproducible software is the changed outcome of software builds over time. While enhancing code reuse, the use of open-source dependency packages hosted on centralized repositories such as PyPI can have adverse effects on build reproducibility. Frequent updates to these packages often cause their latest versions to have breaking changes for applications using them. Large Python applications risk their historical builds becoming unreproducible due to the widespread usage of Python dependencies, and the lack of uniform practices for dependency version specification. Manually fixing dependency errors requires expensive developer time and effort, while automated approaches face challenges of parsing unstructured build logs, finding transitive dependencies, and exploring an exponential search space of dependency versions. In this paper, we investigate how open-source Python projects specify dependency versions, and how their reproducibility is impacted by dependency packages. We propose a tool PyDFix to detect and fix unreproducibility in Python builds caused by dependency errors. PyDFix is evaluated on two bug datasets BugSwarm and BugsInPy, both of which are built from real-world open-source projects. PyDFix analyzes a total of 2,702 builds, identifying 1,921 (71.1%) of them to be unreproducible due to dependency errors. From these, PyDFix provides a complete fix for 859 (44.7%) builds, and partial fixes for an additional 632 (32.9%) builds.

CCS CONCEPTS

• **Software and its engineering** → *Software configuration management and version control systems*; **Software libraries and repositories**; **Software maintenance tools**; **Maintaining software**; **Software evolution**; **Reusability**.

KEYWORDS

software reproducibility, build repair, dependency errors, Python

ACM Reference Format:
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 (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464797

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07.
https://doi.org/10.1145/3460319.3464797

1 INTRODUCTION

Reproducibility of software artifacts is one of the most significant challenges faced by developers and researchers. Reproducibility can be defined as the repeatability of the process of establishing a fact or of conditions under which the same fact can be observed [20]. While the re-use of code components is important for building on existing knowledge and computations, it becomes increasingly more important for open-source software. Open-source software is often built through extensive collaborations, often involving several years of effort. For open-source software to be truly accessible, providing access to source code and documentation may not be sufficient if the reproducibility of the artifact or computational experiment does not stand the test of time.

The adherence to the principle of re-usability is exemplified by the evolution of highly interconnected ecosystems of open-source software libraries hosted on centralized code repositories like Maven Central Repository [7] and PyPI [12]. While this has made the development of new software easier, dependence on other software packages has also led to more build breakage and spread of bugs in dependency networks. Seo et al. [35] found in their study of Java and C++ build failures that nearly half of all build errors are caused by software dependencies. When evaluating Python gists available on GitHub, Horton and Parnin [28] found that 52.4% of the gists failed to execute due to a dependency error. The growth of dependency packages in each programming language's ecosystem has created *transitive dependencies* between these packages. Such dependencies can have the effect of propagating bugs and vulnerabilities, and the removal of a package central to such dependency networks can affect up to 30% of the existing applications [31]. Tomassi et al. [37] while reproducing fail-pass build pairs had a success rate of 5.56% out of the 55,586 pairs collected. Their manual evaluation of 100 unreproducible artifacts showed failure to install dependencies to be the leading cause of unreproducibility.

The software engineering community has made great strides towards achieving reproducibility. The advent of tools like Docker [4] and Kubernetes [6] has simplified the creation and deployment of containers. However, the problem of dependency versions continues to hinder attempts of achieving reproducibility through containerization. Software dependencies are constantly evolving packages and often receive regular updates for fixing bugs and adding features. While the new versions are aimed to enhance the software, they may cause many new issues to surface in applications utilizing these packages. Backward compatibility may change with updated packages, potentially leading to unwanted changed behavior that affects reproducibility. As older versions of dependency packages become deprecated, or the package index URLs become stale, reproducing artifacts that require those older versions becomes difficult.

Recently, Python was reported by several language popularity indices to have become the second most popular language amongst

developers, displacing Java [39]. This shows how ubiquitous Python has become in the programming world. However, with the transition from Python 2.x to 3.x there have been several backward incompatible changes in the language itself [18]. As a consequence, many Python packages have released versions with breaking changes, thus contributing to the unreproducibility of Python artifacts. This is a huge concern even for projects that pin all of their dependencies because dependency versions can be removed from the Python index without warning. While it is possible to manually fix dependency issues when re-using an application, it requires expensive developer hours and domain knowledge.

The issue of reduced reproducibility caused by changing package versions is especially significant for bug datasets, whose growth and longevity depend on the reproducibility of artifacts. Datasets of software bugs play a significant role in the evaluation of fault localization and repair techniques. Two recent Python bug datasets BugSwarm [37] and BugsInPy [41] collect bugs from GitHub open-source projects. While BugSwarm has reported the low reproducibility rate of these bugs as a hindrance for the growth of the dataset, BugsInPy has manually pinned appropriate versions for the dependencies of each project. However, neither dataset has considered the impact of configuration drift in their design and measures to preserve reproducibility. Configuration drift is the phenomenon of a code snippet going out-of-date because APIs it depends on experience breaking change over time [30]. In our analysis of 2,584 BugSwarm builds, we find a total of 46,523 installed packages, comprising of 22,506 project and 24,017 transitive dependencies, respectively. 62.4% of project dependencies are pinned while only 4.0% of transitive dependencies are pinned. Overall, 41.2%, 32.2% and 26.6% dependencies are found to be constrained, pinned and unconstrained, respectively.

In this paper, we look into the usage of dependency packages and their version specifications in builds of open-source Python projects. We focus on builds whose *current build logs* contain dependency-related errors that were not present in the *original build* run at the time the code was committed to GitHub, thus making the builds no longer reproducible. We propose a tool PyDFix that identifies dependency-related error messages in build logs that contribute to unreproducibility, and extracts dependencies that are possibly causing these errors. PyDFix then iteratively builds a final list of *pinned dependencies* to fix dependency-related build errors that form a "patch" to make the build reproducible again. The reproducibility is ensured by validating against the original build logs to check the final outcome of the build *and* the results of any associated tests. Each artifact in the bug datasets consists of a failed build triggered by a buggy commit and a passed build triggered by the commit with a bug fix. PyDFix's goal is to achieve reproducibility. The builds associated with buggy commits should terminate with the same build errors or failed test results as originally observed, and for passed builds, the build should be successful and all tests pass. While we expect PyDFix to be useful for maintainers of bug datasets, any developer facing issues while rebuilding a historical Python build due to package dependencies can utilize PyDFix.

Recent approaches [26, 32, 33] repair build failures in Java projects, but do not focus on reproducibility. In terms of Python, DockerizeMe [29] works on inferring environment configurations for Python gists, and V2 [30] identifies out-of-date gists and notebooks due to breaking changes in APIs used by them. Although these studies have a focus on resolving Python dependency issues, their approaches do not work for large Python applications. Sciunit [36] and ReproZip [21] present a preventive approach based on operating-system call traces to containarize applications and maintain their reproducibility. However, these approaches cannot be used if an artifact is already unreproducible.

We evaluate PyDFix on 2,702 Python builds from the BugSwarm and BugsInPy datasets. Only builds which had the original source code available on GitHub were included in the evaluation as the source code provided for artifacts in both datasets contained modifications aimed at maintaining reproducibility. PyDFix identifies a total of 1,921 builds as being unreproducible due to dependency errors. These include 67.2% of analyzed builds from BugSwarm, and 84.9% of analyzed builds from BugsInPy. PyDFix successfully computes a complete fix for 859 builds (40.9% and 55.5% of identified builds from BugSwarm and BugsInPy, respectively) while also creating partial fixes, which have not restored reproducibility but resolved a number of dependency-related errors for 632 builds (32.1% of BugSwarm builds and 35.4% of BugsInPy builds).

The main contributions of this paper are:

- We study dependency usage, version specifications and inclusion of transitive dependencies in Python projects (Section 3).
- We design an algorithm to automatically identify dependency errors and likely candidate dependencies causing such issues from build logs (Section 4.1).
- We design an iterative solving algorithm to synthesize and apply patches based on identified candidate dependencies (Section 4.2).
- We develop PyDFix to identify and fix unreproducible Python builds caused by dependency packages, and conduct a large-scale evaluation on 2,702 builds from two Python bug datasets BugSwarm and BugsInPy (Sections 5.1 and 5.2).

2 BACKGROUND AND MOTIVATION

This section provides background on Python dependencies, an example of dependency errors, and some terminology.

2.1 Managing Python Dependencies

Python developers have multiple options to specify their applications's dependencies. Dependency requirements can be declared in text files containing one dependency specification per line, or by using the `install_requires` keyword in a `setup.py` file, which is a script used for packaging and distribution of Python projects. Configuration files for continuous integration (CI) tools like TravisCI [16] can also contain Python dependency package declarations. Several virtual environment management packages like `tox` [14] and `pyenv` [10] allow configurations to declare dependencies. Moreover, there exist multiple package managers for Python, the two most popular being `pip` [8] and `conda` [2]. We only consider packages installed using `pip` as it is Python's standard package manager [17].

PEP 440 [34] and PEP 508 [22] provide detailed information about the versioning system of Python packages as well as the types of dependency declarations and version specifications available to Python developers. However, there is no single set of best practices that the Python developer community follows and it can

vary greatly depending on developer choice. Dependency version specification in Python can be done in three ways:

- **Pinned Dependency**: a specific version is included in the dependency declaration e.g., `numpy==1.17.5`.
- **Constrained Dependency**: a range of versions is specified in the dependency declaration e.g., `numpy>=1.17.5,!=1.18.2`.
- **Unconstrained Dependency**: no version specification is mentioned in the dependency declaration.

Dependency packages with a *pinned version* can affect a build outcome if the package is removed from `PyPI`, or if its versioning system changes. For example, the package `pytest-capturelog`, which is documented in libraries.io [11] no longer exists in `PyPI`. Another example is `pyatom`, which still exists in `PyPI` but whose versioning system completely changed in January 2020 [9]. Earlier versions are no longer hosted on `PyPI`.

The default behavior of `pip` for *constrained dependencies* is to fetch the latest available version that satisfies the constraint. Thus, both unconstrained and constrained dependencies can lead to the installation of versions newer than those originally used. While later versions of dependency packages have bug fixes and additional functionalities, they may also contain breaking changes that cause build failure for applications that worked with older versions.

Apart from declared dependencies, unreproducibility can also be caused by *transitive dependencies*. Transitive dependencies are packages not directly used by the application itself, but by a package used by the application. Each package used by an application can have any number of such transitive dependencies, and it is difficult to infer which version would be appropriate for a failing transitive dependency without analyzing the source code of the dependency package directly used by the application.

2.2 An Example of Dependency Errors

Figure 1 shows an example of dependency errors in a Python project. Figure 1b shows the *current* build outcome for a historical commit [1] from the GitHub repository `cloudify-system-tests`. The build terminates with an error due to the package `stevedore` [13], which was not encountered in the *original* build as shown in Figure 1a. However, this dependency is not declared within the application. In the log line documenting the installation of `stevedore` shown in Figure 1c, we observe that `stevedore` is actually a transitive dependency, required by the package `openstacksdk`. Although `cloudify-system-tests`'s source code has pinned a version for `openstacksdk`, the version constraint from `stevedore` is actually being controlled by `openstacksdk`. At the time when this code was committed to the GitHub repo, the `stevedore` version fetched by `pip` was compatible with Python2.7, which is correct for this version of `cloudify-system-tests`. However, the current version of `stevedore` requires a Python version ≥ 3.6 and hence, causes an error when pulled into the build process for this commit.

As shown in Figure 1d, pinning `stevedore` does not repair the build completely and another error is encountered. While the error due to `stevedore` occurred during the installation steps, this new error occurs during the run of `flake8` on the source code. The error message now indicates that it is caused by a failure to find the module `configparser` [3]. Inspecting the installation steps of the logs show that `configparser` was indeed installed and it is a transitive

```
    The command "flake8 ." exited with 0.

    Done. Your build exited with 0.
```

(a) Fragment from *original* build log

```
  error in setup command: Error parsing /tmp/pip-build-D943WL/stevedore/setup.
        cfg:
  ----------------------------------------------
  Command "python setup.py egg_info" failed with error code 1 in /tmp/pip-build
        -D943WL/stevedore/

The command "pip install ." failed and exited with 1 during .
The build has stopped.
```

(b) Fragment from *current* build log

```
Collecting stevedore>=1.17.1 (from openstacksdk0.9.13->cloudify-system-tests
      ==4.0.1)
```

(c) Dependency installation of `stevedore` in build log

```
  File "/home/travis/virtualenv/python2.7.9/lib/python2.7/site-packages/flake8/
        main/mercurial.py",
  line 7, in <module>
     import configparser
  ImportError, No module named configparser

  travis time:end:0cc6ae93:start=1608325842106715960,finish
        =1608325842255517368,duration=148801408

  The command "flake8 ." exited with 1.
  Done. Your build exited with 1.
```

(d) Build error after pinning correct version of `stevedore`

```
  Collecting configparser (from flake8==3.3.0)
```

(e) Dependency installation of `configparser` in build log

```
  stevedore==1.17.1
  configparser==3.5.0
```

(f) Final patch with pinned problematic dependencies

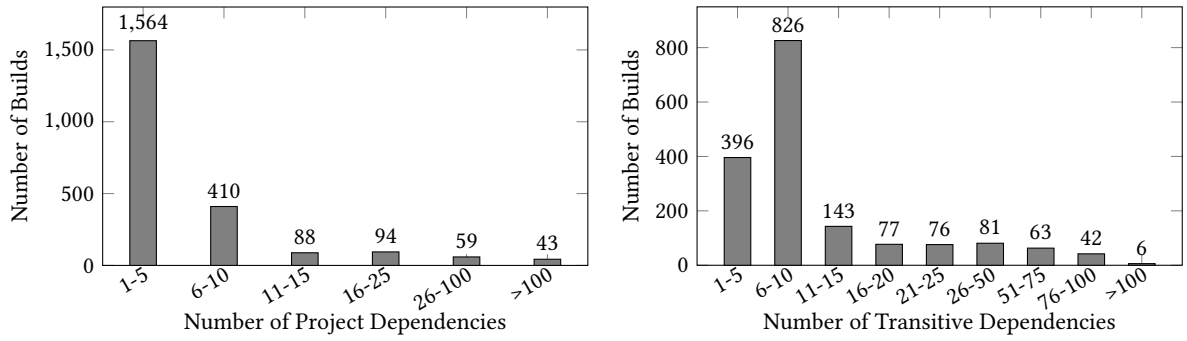
Figure 1: An example requiring multiple version specification changes to restore build reproducibility

dependency of `flake8` [5]. Although the developers specified a particular version for `flake8`, there is no version constraint added for `configparser` within the package `flake8`. The current version of `configparser` again works only with Python versions ≥ 3.6 and hence, is incompatible in the current build process. Pinning the correct version of `configparser` along with `stevedore` results in restoring the build to original status.

2.3 Terminology

Here we introduce some terms to be used in the paper.

- **Triggering Commit:** A commit that triggered a build. For the example in Section 2.2, the triggering commit is the commit [1] that executed the original build shown in Figure 1a.
- **Build Log:** A build log is the log generated when a build is executed. An *original build log* is the log produced by the build started by the triggering commit, which shows the expected build process. A *current build log* is the log generated by executing the same build process at a later time (now). The terminating



(a) Builds by number of project dependencies

(b) Builds by number of transitive dependencies

Figure 2: Builds by number of project and transitive dependencies

lines of the *original* build log are shown in Figure 1a while the *current* build log's terminating error is shown in Figure 1b.

- **Build Outcome:** A build outcome is the final status of a build. For a passed build, build outcome is success. In case of a failed build, the build outcome is the error that terminates the build.
- **Unreproducible Build:** A build whose current build outcome differs from its original build outcome. For example, a failed build that currently terminates due to an error different from the error of the original build, or a passed build that is currently failing. Both failed and passed builds are considered unreproducible if associated tests have different results from the original logs. In our example, the original build log in Figure 1a shows a successful build outcome, but the current build outcome is the error shown in Figure 1b.
- **Dependency Chain**: Every transitive dependency has a dependency chain showing the dependency packages that led to the inclusion of this transitive dependency. E.g., in Figure 1c the dependency chain for the transitive dependency `stevedore` is `openstacksdk==0.9.13→cloudify-system-tests==4.0.1`.
- **Patch:** A patch is any change to an application intended to fix a problem. In our study, a patch is a list of dependency specifications that pin the versions of dependency packages to fix dependency-related errors that make an application's build unreproducible. An example of a patch is shown in Figure 1f.
- **Patch Candidate:** For PyDFix, a patch candidate consists of a single dependency package and its suitable version. Figure 1f shows the *two* patch candidates in the final patch used to make the build reproducible.
- **Fix Status:** The status of PyDFix's fix on a build which is identified as unreproducible due to dependency errors. If PyDFix is able to restore reproducibility, i.e., make the build outcome as well as the test results of the unreproducible build match that of the original build, then the fix status is considered a *Complete Fix*. If the build was not made reproducible but a non-empty patch is computed, the fix status is considered a *Partial Fix*. Section 2.2 shows how a patch was applied to an unreproducible build to achieve the outcome of a *Complete Fix*.

3 DEPENDENCY VERSION SPECIFICATIONS

To further motivate our work, we investigate the frequency in different dependency version specifications used in 1,292 BugSwarm artifacts, i.e., 2,584 builds.[1] Our approach consists of parsing the original log of each build. We adopt this method instead of directly analyzing the source code because of two reasons. Firstly, we only identify the dependencies that are installed and used, thus avoiding dependencies that may have been declared in unused sections of the source code. Secondly, this approach allows us to identify all transitive dependencies being installed, which are not declared in the source code but are required by other dependencies.

3.1 Frequency of Version Specifications

We observe a widespread use of dependency packages with a total of 46,523 instances of package installations across all builds. Figures 2a and 2b show the count of builds containing a range of project and transitive dependencies, respectively. We found that 326 builds (12.6%) did not fetch any packages from the `PyPI` index, i.e., these builds have zero dependencies. Additionally, 548 builds (21.2%) did not show evidence of installing any transitive dependencies. For both project and transitive dependencies, most builds have less than 10 dependencies per project. However, the number of builds having a large number of dependencies is not insignificant. Especially in the case of transitive dependencies, the number of builds in higher dependency ranges is evenly distributed. While builds can fail due to an error caused by even a single dependency and fixing such an error requires domain knowledge, the need for an automated approach for dependency resolution is more pronounced for artifacts having a large number of dependencies.

In Figure 3 we show the distribution of types of dependency version specifications encountered. These are all dependencies found in the entire set of build logs analyzed. The labels above each bar in the plot refer to the absolute count represented by the bar followed by its percentage across all dependencies of that type, i.e., project, transitive, and total dependencies. The figure shows that the majority, 62.4% of project dependencies are pinned while 12.1% are constrained, and 25.5% are unconstrained. However, most of the transitive dependencies are constrained while only 4.0% are

[1] BugsInPy could not be analyzed because the original logs were not available.

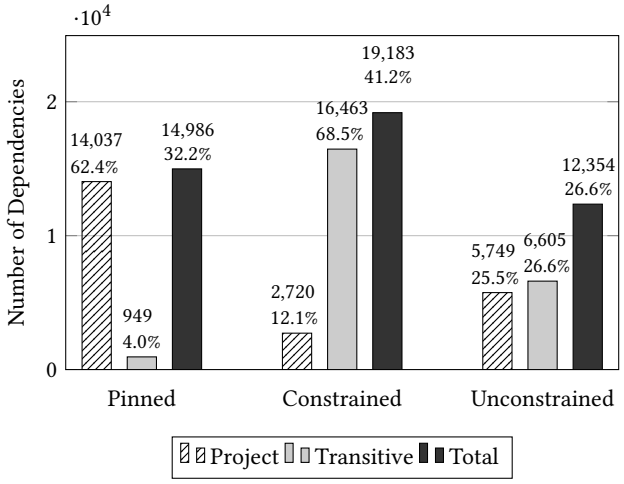
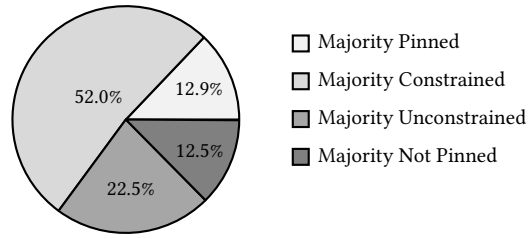
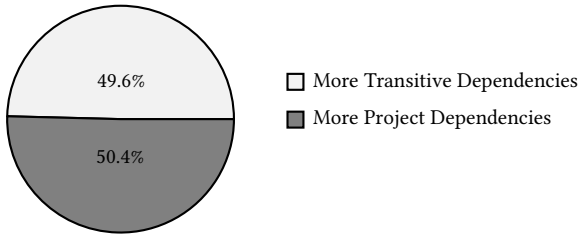


Figure 3: Dependencies by version specification



(a) Breakdown of builds by majority dependency version specification



(b) Breakdown of builds by majority type of dependency

Figure 4: Breakdown of builds by dependency version specifications and dependency type

pinned. A significant percentage of transitive dependencies, 26.6% are also unconstrained. While looking at the total dependencies including both types, the largest portion which is 41.2% is constrained followed by 32.2% pinned and 26.6% unconstrained. This highlights the need for developing tools like PyDFix to address dependency-related errors.

From Figure 4a we also find that only 12.9% of the builds have most of their required dependencies pinned, 52.0% of the builds have the majority of their required dependencies constrained, and 22.5% builds have the majority of the package dependencies unconstrained. As explained in Section 2.1, both constrained and unconstrained dependencies can lead to the installation of newer versions that may contain breaking changes. Finally, we observe from Figure 4b that 49.6% of the builds contain more transitive dependencies than project dependencies. This underscores how important transitive dependencies and their version specifications are while maintaining reproducibility of builds.

Findings: 46,523 packages are installed across 2,518 builds. 1,067 builds have 10+ project or transitive dependencies. While most project dependencies were pinned at 62.4%, only 4.0% of transitive dependencies are pinned. We find 12,354 (26.6%) total unconstrained and 19,183 (41.2%) constrained dependencies, which increase the likelihood of build breakage. The 14,986 (32.2%) pinned dependencies can also lead to unreproducibility if removed from the package index. 49.6% of builds contain more transitive than project dependencies, highlighting the importance of transitive dependencies in build repair.

3.2 Challenges in Fixing Broken Dependencies

The main challenge in identifying dependency-related unreproducible builds is due to the unstructured nature of build logs. Logging of build errors is varied, and often does not provide exact information about the cause of an error. Log parsing approaches have to take into account error traces preceding error messages in the search of a root cause when unable to extract desired information from error messages in logs. Moreover, there exists no exhaustive set of errors that are caused by package dependencies and can be used for identification. Thus, identifying dependency-related build errors is a significant challenge.

Second, for large Python applications using many dependency packages, the search space of version specifications is exponential. Each Python package has several release versions available on package indexes, and a brute force approach to find the correct version is not feasible. Hence, it is important to limit the modification of version specifications to only dependencies causing errors while excluding versions not likely to fix errors.

A third challenge is posed by transitive dependencies, which are not explicitly included in a project but are required by a project dependency or other transitive dependencies. The inclusion of a transitive dependency cannot be detected from the source code of a project, and can only be inferred from the build logs of the installation process. Transitive dependencies also give rise to dependency chains (explained in Section 2.3) that contribute to the expansion of the search space. Due to the expensive nature of evaluating patch candidates (Algorithm 2) and the inclusion of a large number of transitive dependencies (Section 3.1), it is imperative to identify problematic dependency chains for the practicality and time efficiency of our approach.

To the best of our knowledge, PyDFix is the first to address all above challenges for large Python projects, and be evaluated on a large and wide-ranging set of builds.

4 TECHNICAL APPROACH

PyDFix is shown in Figure 5. The two main components are LogErrorAnalyzer for identifying dependency-related errors causing

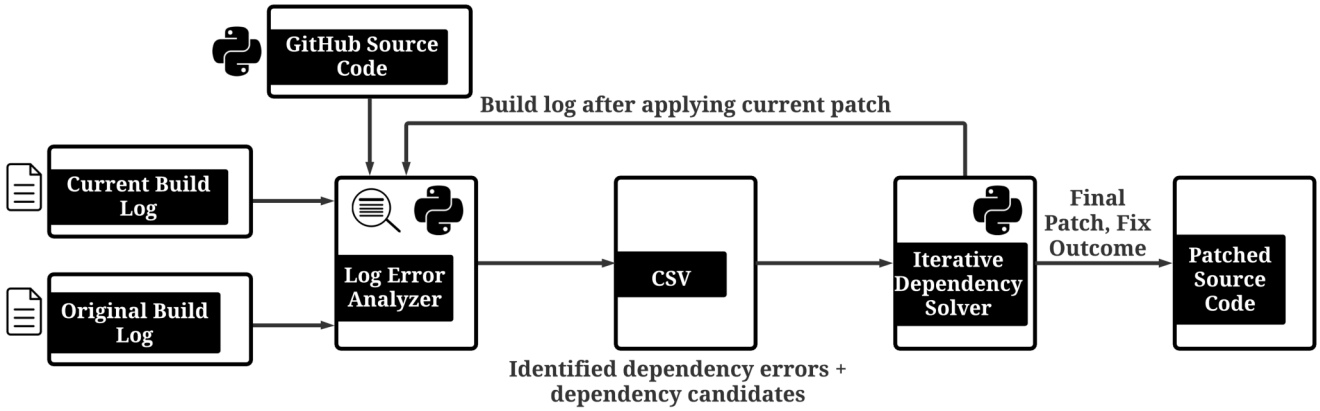


Figure 5: PyDFix workflow

unreproducibility, and IterativeDependencySolver for fixing unreproducible builds due to dependencies. PyDFix takes as input the current build log and the original build log. PyDFix first identifies dependency errors and possible dependency packages causing these errors using LogErrorAnalyzer. This is followed by iteratively building a patch that makes the build reproducible again by IterativeDependencySolver. The iterative algorithm for building the patch re-runs the build with intermediate patches and analyzes the new build logs produced to further identify errors and problematic dependency version specifications. This process continues until the build becomes reproducible, or all patch options have been tested and deemed not useful. The key challenges addressed by PyDFix are the identification of dependency-related unreproducible builds from build logs, and the selection of both project and transitive dependencies along with their appropriate versions to fix dependency errors and test failures. We expect PyDFix to be of great value for maintainers of bug datasets as well as developers attempting to reproduce a historical build. The rest of this section provides further technical details.

4.1 Log Error Analyzer

The first step in solving dependency-related build breakage is the identification and localization of build errors. Build tools like Gradle for Java may have pre-defined sections in the build logs ("What went wrong") where error messages and exceptions are collected. However, not all Python applications need or have build tools. Hence, while analyzing Python build logs we cannot depend on any pre-defined section of log messages showing the reasons for build failure. LogErrorAnalyzer analyzes build logs to extract the following information:

(1) What are the lines indicating errors due to dependency packages, and are these errors absent in the original build?
(2) What are the dependency packages that cause these errors?
(3) What files lead to the inclusion of these dependency packages in the project?

Table 1: Subset of log error regex patterns

Error Pattern
requirements\.txt needs (.*) python
The command "pip3? install(.*)failed and exited with(.*)
Command "python3? setup\.py egg_info" failed
The command "python3? (.*) failed and exited (.*)
virtualenv\.py: error
(.*) requires a different Python (.*)
No module named (.*)
ImportError (.*)
ModuleNotFoundError: (.*)
(.*) : command not found

4.1.1 Error Patterns. To understand which error messages in a build log indicate failure related to dependency packages, we manually inspected the build logs of 40 unreproducible artifacts from the BugSwarm dataset, which led to the identification of 20 different error messages. Based on these error messages, we created 17 `regex` patterns, 10 of which are shown in Table 1.

The error patterns in Table 1 indicate that dependency errors are related to the failure of `pip install`, failure to setup `Python` egg, a package or dependency file requiring a different `Python` version, an error from a virtual environment, `ImportError` and `TypeError`. An additional indication of an incorrect version installation appeared to be code style checks failing. For example, `flake8` is a `PyPI` package that keeps adding new code style rules with every new version. However, older artifacts that use `flake8` to check for code style issues may not be compliant with newer rules in the latest version of `flake8`. In such a case, `flake8` needs to be pinned to an appropriate version and hence, we have included errors from such code style checking packages into our list of dependency-related errors.

4.1.2 Installation Patterns. Based on the log inspection from Section 4.1.1, we also created a set of `regex` patterns to identify package installation messages in build logs. The installation patterns are shown in Table 2. These patterns are used by LogErrorAnalyzer

Table 2: List of log patterns for package installation

Package Installation Pattern
^Collecting(.*)
^Searching(.*)
^Downloading(.*)
^Requirement already satisfied:(.*)
^Best Match(.*)

to answer questions (2) and (3). In particular, the analyzer tracks: (i) required packages, (ii) package versions fetched and installed, (iii) whether a package is a transitive dependency in which case the analyzer also extracts the dependency chain, and (iv) files in which the dependencies are specified.

4.1.3 Extracting Candidate Packages for Fix. Algorithm 1 shows how LogErrorAnalyzer parses build logs to extract information required to address the errors of an unreproducible Python build. LogErrorAnalyzer iterates over each line of the current log for which the build is unreproducible. The analyzer looks to match error patterns (Section 4.1.1) and installation patterns (Section 4.1.2). For every match against an installation pattern, further package details are extracted such as pinned version, version constraints, and transitive dependency chain, after which the package is added to the set of installed packages.

When an error pattern is matched, the analyzer first checks whether the same error (and error trace) already exists in the original log (Algorithm 1 Lines 10-16). If that is the case, then the error is discarded and not considered to be a new error contributing to the unreproducible state of the build. For errors not found in the original logs, the error message itself and the associated error trace is parsed to extract packages associated with the error. The detailed information about the error-related dependency packages is collected from the already gathered information on installed packages and these are added to the possible candidates for a fix. If any of these packages are transitive dependencies, then all packages appearing in its dependency chain are also considered possible candidates.

If an identified error trace is not associated with any known error pattern, and the error does not exist in the original log, then the trace is still parsed and analyzed to extract mentions of dependency packages. Finally, the last installed package before the build error occurred is also included in the possible candidates since it is highly likely that the error occurred during the installation of that package.

LogErrorAnalyzer arranges the candidate packages in a priority order used by IterativeDependencySolver (Section 4.2.2) when applying candidate patches. The order is described as follows:

(1) The dependency package mentioned in the error line itself.
(2) Dependency packages listed in the error trace associated with the error line, with priority decreasing with further distance from the error line.
(3) Dependency packages listed in an error trace not associated with any of the recognized error patterns, with decreasing priority as we go down the error frames of the trace.
(4) The dependency installed right before the error occurred.

Algorithm 1: LogErrorAnalyzer

Data: Dependency error regex, Package installation regex
Input: Current build log, TravisCI original build log
Output: Dependency errors, Candidate dependency packages, Dependency files

1 $installed \leftarrow []$, $candidates \leftarrow []$
2 $errorLines \leftarrow []$, $fileNames \leftarrow []$
3 **for** *each line in log* **do**
4 **if** *line matches package installation regex* **then**
5 $pkgInfo \leftarrow$ package details extracted from line
6 $installed.insert(packageInfo)$
7 **if** $fileName$ *present in line* **then**
8 $fileNames.insert(fileName)$
9 **end**
10 **else if** *line matches dependency error regex and error not in original log* **then**
11 $errorLines.insert(line)$
12 $pkgNames \leftarrow$ package names in error trace
13 $pkgInfoList \leftarrow$ packages in $pkgNames$ from $installed$
14 $candidates.insert(pkgInfoList)$
15 **else if** *line shows start of an error trace and error trace not in original log* **then**
16 $pkgNames \leftarrow$ package names in error trace
17 $pkgInfoList \leftarrow$ packages in $pkgNames$ from $installed$
18 $candidates.insert(pkgInfoList)$
19 **end**
20 $orderPossibleCandidatesByPriority(candidates)$
21 **return** $errorLines$, $candidates$, $fileNames$

4.2 Iterative Dependency Solver

Once LogErrorAnalyzer has identified the possible candidates and their priorities, the IterativeDependencySolver shown in Algorithm 2 generates patch candidates for each possible candidate and adds them to the dependency requirements of the build according to the prioritized order. In every iteration, a previously unapplied patch candidate is added to the final accepted patch or discarded based on the build outcome. Thus, pinned dependencies are incrementally added to the final accepted patch until encountering either one of the terminal states in Table 3 (Section 4.2.3), or a non-dependency error. Based on each iteration's build outcome, new candidates are found by analyzing the build log generated after applying the current patch.

4.2.1 Generating Patch Candidates. As shown in Algorithm 2 (Lines 1 and 32), patch candidates are generated at the beginning, and again every time the patch candidates need to be updated due to new possible candidates. The patch generation function identifies one or more suitable versions of the dependencies included in the possible candidates list. The criterion for a suitable version based on version specification is as follows:

- For **Unconstrained Dependencies**, the latest version available before the date of the triggering commit of the build.

Algorithm 2: IterativeDependencySolver

```
Input: buildDetails, errors, candidates
Output: A patch that pins dependencies
1  patchCandidates = genPatches(candidates)
2  acceptedPatch ← [], currentPatch ← [],
     fixOutcome = None
3  while True do
4      currentPatch = acceptedPatch
5      newPatchCandidate = None
       /* Patch Candidate Selection */
6      newPatchCandidate ←
         getUnappliedPatch(patchCandidates, acceptedPatch)
7      if newPatchCandidate == None then
           /* Encountered terminal state */
8          fixOutcome ← Exhausted All Options
9          break
       /* Patch Candidate Application */
10     newBuildOutcome, newBuildLog ←
         runBuildJob(currentPatch, buildDetails)
       /* Patch Impact Evaluation */
11     if newBuildOutcome == SUCCESSFUL then
           /* Encountered terminal state */
12         acceptedPatch ← currentPatch
13         fixOutcome ← Successfully Fixed Build
14         break
15     else if newBuildOutcome == ORIGINAL_LOG_ERROR
         then
           /* Encountered terminal state */
16         acceptedPatch ← currentPatch
17         fixOutcome ← Restored to Original Error
18         break
19     else if newBuildOutcome == NO_CHANGE then
           /* Discard new patch candidate */
20         continue
21     else if newBuildOutcome == DIFFERENT_ERROR then
22         errorType, newErrors, newCandidates ←
             runLogErrorAnalyzer(newBuildLog)
23         if errorType ! =
             DEPENDENCY_ERROR or no newCandidates then
               /* Encountered terminal state */
24             fixOutcome = No longer a dependency error
25             break
26         else if newCandidates[0] == newPatchCandidate
             then
27             continue /* Discard new patch candidate */
28         else
               /* Patch Candidates Elimination */
29             candidates ← newCandidates
30             errors ← newErrors
31             acceptedPatch ← currentPatch
32             patchCandidates ← genPatches(candidates)
33 end
34 return fixOutcome, acceptedPatch
```

- For **Constrained Dependencies**, the latest version available that satisfies the given constraints and was released before the date of the triggering commit. If such a version is not available, the latest available version is used.
- For **Pinned Dependencies**, the latest version available except the version originally pinned which was released before the date of the triggering commit. An additional patch candidate is created with the latest available version of the package and is given lower priority.
- For **Transitive Dependencies**, the above rules apply to transitive dependencies based on the specification type. Similarly, additional patch candidates are created for all packages included in the dependency chain of transitive dependencies, the highest priority given to the dependency preceeding the broken dependency in the chain.

For fetching the version history of the packages, IterativeDependencySolver uses the PyPI JSON API to get all available versions and then applies the above conditions to find the best suited versions for patch candidates.

4.2.2 Selecting and Applying Patch Candidates. The list of patch candidates is sorted by priority according to the rules described in Section 4.2.1. In every iteration, the first patch candidate in the list of candidates that has not been applied yet is added to the current patch (Algorithm 2, Line 6). If all patches have been applied and discarded, the IterativeDependencySolver can no longer proceed and returns the current patch with the status "Exhausted All Options" (Algorithm 2, Lines 7-9).

Before the iterative application of patches, a modification to the build script is made such that patch dependencies are installed *before* the installation of any other project dependencies. This is sufficient to force the new version for originally constrained or unconstrained dependencies. However, in the case of already *pinned* dependencies, it is necessary to change the source code to replace the previous pinned version with the version specified in the patch. After build and source code modifications are completed, the build is run with the current patch in a clean environment with no Python packages installed. This is particularly important because the success of the fix may depend on discarding some dependencies included in previous patches as explained in Section 4.2.4.

One of the goals when repairing unreproducible software artifacts is to minimize the number of changes made to the existing code and configuration. Unnecessary changes to a project's dependency specifications can cause unprecedented errors while not resolving the original cause of unreproducibility. Thus, an approach which pins all unpinned dependencies to versions the original build uses (if available) may introduce additional security vulnerabilities and bugs through dependencies that did not need to be fixed. Furthermore, the correct patch for a build may change over time due to the ever changing nature of dependency packages. In general, breakage due to dependencies can occur at any time, and fewer changes can facilitate debugging. Moreover, for a purpose similar to our study, users may wish to check the reproducibility of a build without any modifications and juxtapose it against the fix outcome using the patch. In such a case a patch applied with minimum source code editing can be helpful.

Table 3: Fix outcomes

Fix Outcome	Fix Status
Successfully Fixed Build	Complete Fix
Restored to Original Error	Complete Fix
No longer recognized as Dependency Error	Partial Fix
Exhausted All Options	Partial Fix

4.2.3 Evaluating Patch Impact. After the patched build is run, the log status is checked (Algorithm 2, Lines 11, 15, 19 and 21) and the algorithm either exits with one of the fix outcomes in Table 3, or proceeds to the next iteration. If the build is successful, the current patch is accepted as final and IterativeDependencySolver exits with the outcome "Successfully Fixed Build" which is considered to have restored reproducibility and labeled as a *Complete Fix* (Algorithm 2, Lines 11-14). If the current patch makes the build terminate with the same error as in the original build log, the fix outcome is "Restored to Original Error" which is also a *Complete Fix* (Algorithm 2, Lines 15-18).

On the other hand, if the build error remains unchanged, the last dependency package added to the patch is rejected and the next iteration starts. If a new build error is encountered, IterativeDependencySolver runs LogErrorAnalyzer to analyze the new build log. If LogErrorAnalyzer outputs no identified dependency errors, the algorithm outputs the current patch as the final patch and the outcome "No longer recognized as Dependency Error" (Algorithm 2, Lines 23-25). In the case when the highest priority possible candidate is the same dependency that was added to the patch in the current iteration, the algorithm decides that the added dependency has caused the new error and removes it from the patch (Algorithm 2, Line 27). Otherwise, the current patch list is accepted as the correct one so far. If none of the new patch candidates are found to change the dependency related error and are discarded, IterativeDependencySolver exits with the outcome "Exhausted All Options" (Algorithm 2, Lines 7-9). Both the outcomes "No longer recognized as Dependency Error" and "Exhausted All Options" are considered to be *Partial Fix*es if the patches associated with these fixes are non-empty.

4.2.4 Elimination of Patch Candidates. When IterativeDependencySolver reaches a build outcome that still contains a dependency error that is *not* caused by the last added patch, it eliminates all remaining patch candidates in the current list of patch candidates (Algorithm 2, Lines 29-32). The possible candidates that are identified by LogErrorAnalyzer from the current build log are used for generating a new patch candidate list, which constitutes the search space for subsequent iterations.

5 EXPERIMENTAL EVALUATION

This experimental evaluation is designed to answer the following research questions:

RQ1 How many Python builds become unreproducible over time due to dependency errors?

RQ2 How effective is PyDFix in fixing dependency errors to restore unreproducible builds to reproducible status?

Experimental Setup. To evaluate our approach to fix unreproducible builds by fixing dependency errors, we use the software artifacts from two bug datasets built from real-world open-source projects: BugSwarm [37] and BugsInPy [41]. The BugSwarm dataset version 1.1.3 contains 1,292 Python artifacts from 56 unique open-source projects, each artifact is a fail-pass build pair whose source code and build scripts are packaged in a Docker container. BugsInPy consists of 501 Python artifacts from 17 open-source projects, each having a buggy and fixed commit pair, with the buggy commit causing test failures and the fix commit passing those same tests. In the end, 1,351 artifacts were eligible for our study: 1,053 from BugSwarm, and 298 from BugsInPy. Each artifact includes two builds, thus a total of 2,702 builds were considered in this evaluation: 2,106 from BugSwarm and 596 from BugsInPy.

Both the BugSwarm and BugsInPy datasets have measures in place to maintain reproducibility. For BugSwarm, we noticed some differences from the original source code: some dependencies have been pinned. Similarly, BugsInPy artifacts include a dependency specification file that lists manually pinned dependency packages. These measures would interfere with our evaluation in determining reproducibility of the original build, and the actual effectiveness of PyDFix. To avoid this problem, we used the original code obtained from GitHub instead of the code available through BugSwarm and BugsInPy. This led to the exclusion of 239 BugSwarm artifacts whose source code is no longer available on GitHub.

For BugsInPy, we used the setup and test information provided as a part of the metadata for each artifact to generate a `.travis.yml` file. We then used the generated YAML file to create a build script using `travis-build` [15], as done by the BugSwarm infrastructure [37]. We locally ran the build on the source code fetched from GitHub repositories. We had to omit 203 BugsInPy artifacts that did not contain setup instructions, and thus could not be built.

To identify whether a build is reproducible, LogErrorAnalyzer requires the original build logs. BugSwarm artifacts are mined from TravisCI [16] history, and each artifact includes the original build logs in its Docker image. BugsInPy artifacts do not include any build information and thus original build logs are not available for the dataset. To overcome this, we used the logs generated by running the BugsInPy commands i.e., `bugsinpy-checkout`, `bugsinpy-compile` and `bugsinpy-test` on the modified source (with pinned dependencies) as a substitute for original logs.

State-of-the-art tools that fix Python dependency errors, such as V2 [30] and DockerizeMe [29], are not a suitable baseline for PyDFix. Both work on Python gists and V2 also works on notebooks, but not on entire applications. In fact, V2's evaluation excluded all gists with 10 or more direct dependencies to restrict the number of solutions. PyDFix's goal is to restore reproducibility of a build for an application, and it can handle hundreds of direct (and transitive) dependencies. PyDFix is publicly available at https://github.com/ucd-plse/PyDFix. All experiments were run on a workstation with 88 Intel(R) Xeon(R) Gold 2.10GHz CPUs and 384 GB of RAM.

5.1 Evaluation of Dependency Error Impact

In Section 3 we observed that the use of dependency packages is widespread in Python projects, and that many projects have a significant number of unpinned dependencies. However, not all builds

become unreproducible due to dependency errors, and not all cases of unreproducibility are due to dependency packages. As explained in Section 3.2, there can be a number of different reasons for unreproducible builds and in this study we focus on identification of builds that have become unreproducible due to dependency errors. For this purpose, we use LogErrorAnalyzer to analyze current build logs of artifacts from BugSwarm and BugsInPy datasets using the approach presented in Section 4.1.

LogErrorAnalyzer takes the original and current build logs as input for each build to be analyzed and produces a list of dependency errors found, a list of possible candidate dependencies, and identified dependency specification files. We analyzed a total of 2,106 and 596 builds from BugSwarm and BugsInPy, respectively. As shown in Table 4, LogErrorAnalyzer identified 1,415 (67.2%) BugSwarm builds and 506 (84.9%) BugsInPy builds as having dependency errors not present in the original log that cause the builds to be unreproducible. Table 5 shows the distribution of builds across the top 5 error patterns identified. Note that each build log may contain multiple errors that match different error patterns.

From the identification results of LogErrorAnalyzer, we see that without the measures taken in BugSwarm and BugsInPy to maintain reproducibility, a large number of builds from open-source Python projects encounter dependency-related errors that did not occur at the time of the original build. It is evident that manually repairing so many builds affected by different kinds of dependency errors is extremely time consuming. Furthermore, resolving errors from transitive dependencies may require domain knowledge. For example, on inspecting the possible package candidates suggested by LogErrorAnalyzer, we find that 324 and 90 identified builds from BugSwarm and BugsInPy respectively include at least one transitive dependency in their possible candidate sets. This observation further emphasizes the need for automated fixing of unreproducible builds due to dependency errors.

To evaluate the precision of LogErrorAnalyzer, we manually inspected 100 builds chosen at random: 50 builds that were reported by LogErrorAnalyzer to be unreproducible due to dependency errors, and 50 builds that were not reported. Out of the inspected reported builds, 16 (32%) were identified as false positives: the builds had an underlying cause for unreproducibility other than dependency errors. The most common source of false positives was errors due to the unavailability of required environment variables leading to failure of setup steps or Python commands in a build. False negatives, which is the failure to flag a build as unreproducible due to dependency errors, can occur because the error pattern set used by LogErrorAnalyzer is non-exhaustive. However, we did not find any missing error pattern while inspecting the set of 50 non-reported builds.

RQ1 Answer: For BugSwarm, out of 2,106 builds analyzed, 1,415 (67.2%) were identified as having dependency errors. For BugsInPy, out of 596 builds analyzed, 506 (84.9%) were found to have dependency errors. This shows that breakage due to dependencies is quite common. Further manual inspection on a subset of builds revealed a false-positive rate of 32% while no false negatives were found.

Table 4: Builds identified. Columns "#Available" and "#Analyzed" show the total number of builds in the datasets and the number of builds analyzed. Column "# Identified" shows the number and percentage of builds identified by PyDFix as no longer reproducible due to dependency errors.

Dataset	# Available	# Analyzed	# Identified
BugSwarm	2,584	2,106	1,415 (67.2%)
BugsInPy	1,002	596	506 (84.9%)
Total	3,586	2,702	1,921 (71.1%)

Table 5: Distribution of builds in top 5 error patterns. Column "Pattern" shows the error pattern identified and "# Builds" show the total number of builds in which the pattern has been identified.

Error Pattern Description	# Builds
The command "pip3? install(.*)failed and exited with(.*)	711
Command "python3? setup\.py egg_info" failed	502
No module named (.*)	322
ImportError (.*)	255
(.*) requires a different Python (.*)	214

Table 6: PyDFix results. "Identified" shows number of builds to be fixed. "Complete Fix" and "Partial Fix" give the builds that were made reproducible, and those that although still unreproducible had some dependency errors resolved.

Patch Result	Identified	Complete Fix	Partial Fix
BugSwarm	1,415	578 (40.9%)	453 (32.1%)
BugsInPy	506	281 (55.5%)	179 (35.4%)
Total	1,921	859 (44.7%)	632 (32.9%)

5.2 Evaluation of Dependency Error Fixing

We ran PyDFix on the unreproducible builds identified by LogErrorAnalyzer as dependency related: 1,415 and 506 builds from BugSwarm and BugsInPy, respectively. Table 6 shows the results. The patches under "Complete Fix" were successful in making the build reproducible. Patches under "Partial Fix" were not entirely successful but resolved some dependency errors. For BugSwarm, PyDFix was successful in providing complete fixes for 578 (40.9%), and partial fixes for 453 (32.1%) of 1,415 identified builds. While for BugsInPy, PyDFix found complete fixes for 281 (55.5%), and partial fixes for 179 (35.4%) out of 506 identified builds. Overall, PyDFix was able to create complete fixes for 859 (44.7%) and partial fixes for an additional 632 (32.9%) of the identified builds. Among the partial fixes, 204 are no longer dependency-related errors while 428 still contained dependency errors but PyDFix has no more patch candidates to explore.

We also analyzed the dependencies included in the patches computed by PyDFix for BugSwarm builds. Table 7 shows that a total of 2,497 dependency packages are found in patches, with the largest

Table 7: BugSwarm patch metrics. Dependency types are broken into: (1) Unconstrained, Constrained and Pinned, and (2) Project and Transitive. "All" gives the number of dependencies included in patches for each kind. Columns "Complete" and "Partial" list complete and partial fixes, "Max" and "Average" show the maximum and average number of dependencies in a patch.

Dependency Type	All	Complete	Partial	Max	Average
Unconstrained	1,031	577	454	11	0.99
Constrained	1,068	436	632	18	1.03
Pinned	398	175	223	9	0.38
Project	1,780	815	965	14	1.71
Transitive	717	373	344	16	0.68
Total	2,497	1,188	1,309	22	2.40

patch including 22 dependencies. This showcases how time consuming and difficult it can be to manually resolve an unreproducible build with multiple dependency errors. On average, a patch includes 2.4 dependencies. Among all patches, PyDFix pins correct versions of 1,031 originally unconstrained, 1,068 constrained, and 398 pinned dependencies. The smaller number of pinned dependencies show that these cause less dependency-related errors as compared to constrained and unconstrained dependencies. Out of the total number of dependencies, 1,780 are project dependencies, and 717 are transitive dependencies, which illustrates the importance of handling transitive dependencies in fixing dependency errors.

An interesting observation from these results is that PyDFix was successful in automatically finding the correct patch for the problem described in Section 2.2. While patching builds from the same Python projects, PyDFix computed similar patches. For example, for 70 complete patches computed by PyDFix for builds from the project `numpy`, 64 of these patches required version pinning for the same packages. However, for builds from different projects we do not see repetitions of many packages across patches.

Note that IterativeDependencySolver avoids false positives (cases where successful fix outcomes do not actually fix reproducibility) by ensuring that the final patch is non-empty, and the build outcome and test results match the original logs. On the other hand, PyDFix does suffer from false negatives (failure to fix dependency errors). We manually inspected 50 builds from the 430 builds (22.4%) for which PyDFix could not produce a successful patch. We found that 34 (68%) of the inspected builds had legitimate dependency errors causing unreproducibility. The remaining 32% of builds were incorrectly identified as dependency errors by LogErrorAnalyzer and hence, could not be fixed by PyDFix. One of the reasons observed for false negatives is the failure to identify the correct package dependency causing the error. In some builds, dependency errors exist due to the removal of an entire package from the `PyPI` index, as it was the case for the package `pytest-capturelog` discussed in Section 2.1.

Performance of PyDFix. Due to the large number of builds to process, serial execution was extremely time consuming. Since the processing of each build is independent of each other, PyDFix was implemented using Python multiprocessing. The average, median and maximum time taken by PyDFix to fix a build is 70.6, 16.1 and 414.1 minutes, respectively. Note that PyDFix's runtime is highly impacted by the runtime of each build and its associated tests.

RQ2 Answer: Out of 1,415 BugSwarm builds, PyDFix found a complete fix for 578 (40.9%) and a partial fix for 453 (32.1%) builds. For BugsInPy, out of 506 identified builds, 281 (55.5%) builds were made reproducible while 179 builds (35.4%) were partially fixed. The average and median time PyDFix required to fix artifacts are 70.6 and 16.1 minutes, respectively.

6 THREATS TO VALIDITY

PyDFix is evaluated on 2,702 builds from two state-of-the-art Python bug datasets, BugSwarm and BugsInPy, which are built from 56 and 17 open-source Python projects, respectively. Thus, although the number of builds is large, the variety of projects is still limited. Moreover, the set of error messages that we have compiled and used in LogErrorAnalyzer is not inclusive of all types of build errors that may arise from dependency issues. Builds from a more wide variety of Python projects may suffer from dependency errors that are not included in our set. Note that such errors could be easily added to PyDFix, which would improve its effectiveness. There also exists the possibility that changing a version specification of a dependency may resolve build errors but changes application behavior. We address this concern by taking into account the results of tests included in the build to ensure consistent application behavior. But in doing so, we are dependent on the thoroughness of the tests included in the build. Finally, the cause of unreproducibility in a build may change over time and not all causes of unreproducibility are dependency-related. While the approach presented in PyDFix is aimed at handling the changing nature of unreproducible builds, we recognize that this may cause future evaluations of PyDFix to differ from this paper. For this reason, we have maintained detailed records of the original experiments to demonstrate PyDFix's effectiveness at the time of writing. Our data and source code can be found at https://github.com/ucd-plse/PyDFix.

7 RELATED WORK

Reusable Research Objects. Sciunit [36] and ReproZip [21] focus on a preventive approach to reproducibility by capturing requirements and configurations while the application is still running and before distribution of the artifact. Our approach focuses on addressing builds that have become unreproducible and PyDFix only requires the original build log for gathering information instead of requiring the artifact to be functional.

Automatic Build Repair. BuildMedic [33] is a tool to automatically repair dependency-related build breakages for Maven builds in Java projects. Similar to PyDFix, BuildMedic also uses a set of regular expressions to extract build outcome and details from build logs and iteratively apply repair plans to fix the build. However, BuildMedic is not focused on reproducibility of builds, and only tries to repair failed builds. BuildMedic also skips running tests associated with the build, while the results of tests is an important factor in evaluating PyDFix's success. The authors do not take

into account that their build fixes may alter application behaviour, which tests expose. This is especially important since only 36% of BuildMedic's repairs are identical to developer-performed repairs. The focus of BuildMedic is emphasized by their investigation of developers' fixing strategies while our study investigates the usage patterns of dependency packages, their version specifications and their impact on reproducibility of builds.

HireBuild [26] uses historical build fix information to generate patches for build scripts that have led to similar kinds of build failure. HoBuff [32] utilizes the current information present in build logs to extract error information, followed by fault localization using dataflow analysis to generate patches. Neither of these tools address the reproducibility of the builds they fix, and their approaches are designed for Java projects that use the Gradle build system's logging structure. PyDFix cannot benefit from such specifics in the build logs because most Python applications do not need or use build tools, and ensures that the build outcome is reproducible.

Inferring Environment Dependencies. DockerizeMe [29] is a technique for inferring environment dependencies of a Python code snippet to resolve ImportErrors using an offline knowledge base of Python packages in the PyPI index. DockerizeMe has the limitation of having only ImportErrors within its scope while dependency errors can cause varied and complex issues as seen by error patterns used by PyDFix. V2 [30], an extension of DockerizeMe, identifies out-of-date code snippets by detecting configuration drift. Both tools focus on Python dependency issues similar to our approach, but they can only be used for code snippets and V2 additionally works on Jupyter notebooks. Moreover, V2 excluded all code snippets having more than 10 dependencies from their evaluation to restrict the number of possible solutions. In contrast, PyDFix works on builds from real-world Python applications and thus is capable of resolving more complex dependency problems and handling a large number of dependencies.

Dependency Graphs and Ecosystems. VeriBuild [25] uses a unified dependency graph (UDG) to solve the problem of missing or redundant dependencies in build scripts. However, VeriBuild only addresses discrepancies between dependencies of build targets while PyDFix fixes unreproducibility of builds caused by incompatible dependency package versions. Recent work [24, 27, 31] provide insightful information about dependency packages used in several programming languages other than Python, their networks, evolution and impact. Decan et al. [23] compare and contrast the component dependency graphs of the Python, Javascript and R ecosystems. Valiev et al. [38] present a study of PyPI to understand ecosystem-level factors affecting the sustainability of open-source Python projects. In contrast to the aforementioned studies, our work does not focus on understanding dependency networks and ecosystems, but on exploring the widespread use of Python dependency packages and their impact in reproducibility.

Other Dependency Issues. Watchman [40] addresses the problem of dependency conflicts arising in PyPI ecosystem and is designed to help the developer community detect and predict dependency conflicts and also generate diagnostic information to help fix these issues. However, this tool does not work on the dependency requirements of individual builds, and thus cannot fix unreproducibility in builds caused by dependency packages. Abate and Cosmo [19] propose an algorithm to predict operational failures in a system introduced by the upgrade of a single component. The algorithm analyzes inter-component dependencies to predict consequences of an upgrade and applies this technique to the Debian distribution. This algorithm has a pre-emptive approach to failures caused by components while PyDFix addresses the unreproducibility once it has already been caused by dependency errors.

8 CONCLUSION

Dependencies find widespread use in open-source Python projects. We investigated the impact of dependency package usage on unreproducibility of builds and propose PyDFix to identify and fix unreproducible builds due to dependency errors. PyDFix was evaluated on two Python bug datasets, BugSwarm and BugsInPy, which are built from real-world open-source projects. PyDFix analyzed 2,702 builds in total, identifying 1,921 (71.1%) of them to be unreproducible due to dependency errors. Out of these, PyDFix computed complete fixes for 859 (44.7%) builds, and partial fixes for an additional 632 (32.9%) builds. In the future, we would like to automate the process of extracting error patterns for dependencies, which would improve our identification of unreproducible builds due to dependency errors, and the precision of PyDFix at identifying candidate dependencies.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under award CNS-2016735. We would like to thank Aditya V. Thakur and Premkumar T. Devanbu for providing feedback on earlier drafts of this paper. We also thank Erin M. Winter and Ryan Jae for their help replicating the experiments conducted in this study.

REFERENCES

[1] Accessed 2021. cloudify-system-tests triggering commit. https://github.com/cloudify-cosmo/cloudify-system-tests/tree/bf27ad94b2fb11183beb2f374f5eb06b7af31bdf.
[2] Accessed 2021. Conda. https://docs.conda.io/en/latest/.
[3] Accessed 2021. configparser. https://pypi.org/project/configparser/.
[4] Accessed 2021. Docker. https://www.docker.com/.
[5] Accessed 2021. flake8. https://pypi.org/project/flake8/.
[6] Accessed 2021. Kubernetes. https://kubernetes.io/.
[7] Accessed 2021. Maven Central Repository. https://repo1.maven.org/maven2/.
[8] Accessed 2021. pip. https://pypi.org/project/pip/.
[9] Accessed 2021. pyatom versions. https://libraries.io/pypi/pyatom/versions.
[10] Accessed 2021. pyenv. https://pypi.org/project/pyenv/.
[11] Accessed 2021. pytest-capturelog. https://libraries.io/pypi/pytest-capturelog.
[12] Accessed 2021. Python Package Index. https://pypi.org/.
[13] Accessed 2021. stevedore. https://pypi.org/project/stevedore/.
[14] Accessed 2021. tox. https://pypi.org/project/tox/.
[15] Accessed 2021. travis-build. https://github.com/travis-ci/travis-build.
[16] Accessed 2021. Travis CI. https://travis-ci.org/.
[17] Accessed 2021. What Is Pip? A Guide for New Pythonistas. https://realpython.com/what-is-pip/.
[18] Accessed 2021. What's New In Python 3.0. https://docs.python.org/3/whatsnew/3.0.html.
[19] Pietro Abate and Roberto Di Cosmo. 2011. Predicting upgrade failures using dependency analysis. In *Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 145–150. https://doi.org/10.1109/ICDEW.2011.5767626
[20] Bente Anda, Dag I. K. Sjøberg, and Audris Mockus. 2009. Variability and Reproducibility in Software Engineering: A Study of Four Companies that Developed the Same System. *IEEE Trans. Software Eng.* 35, 3 (2009), 407–429.

https://doi.org/10.1109/TSE.2008.89

[21] Fernando Chirigati, Rémi Rampin, Dennis E. Shasha, and Juliana Freire. 2016. ReproZip: Computational Reproducibility With Ease. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 2085–2088. https://doi.org/10.1145/2882903.2899401

[22] Robert Collins. 2015. *PEP 508 – Dependency specification for Python Software Packages.* Retrieved January 23, 2021 from https://www.python.org/dev/peps/pep-0508/

[23] Alexandre Decan, Tom Mens, and Maëlick Claes. 2016. On the topology of package dependency networks: a comparison of three programming language ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops, Copenhagen, Denmark, November 28 - December 2, 2016*, Rami Bahsoon and Rainer Weinreich (Eds.). ACM, 21. http://dl.acm.org/citation.cfm?id=3003382

[24] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2017. An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. *CoRR* abs/1710.04936 (2017). arXiv:1710.04936 http://arxiv.org/abs/1710.04936

[25] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 463–474. https://doi.org/10.1145/3395363.3397388

[26] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: an automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1078–1089. https://doi.org/10.1145/3180155.3180181

[27] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. 2018. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Andrea Zisman and Sven Apel (Eds.). ACM, 101–104. https://doi.org/10.1145/3183399.3183417

[28] Eric Horton and Chris Parnin. 2018. Gistable: Evaluating the Executability of Python Code Snippets on GitHub. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 217–227. https://doi.org/10.1109/ICSME.2018.00031

[29] Eric Horton and Chris Parnin. 2019. 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*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 328–338. https://doi.org/10.1109/ICSE.2019.00047

[30] Eric Horton and Chris Parnin. 2019. V2: Fast Detection of Configuration Drift in Python. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 477–488. https://doi.org/10.1109/ASE.2019.00052

[31] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 102–112. https://doi.org/10.1109/MSR.2017.55

[32] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 43–54. https://doi.org/10.1145/3293882.3330578

[33] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 106–117. https://doi.org/10.1109/SANER.2018.8330201

[34] Donald Stufft Nick Coghlan. 2013. *PEP 440 – Version Identification and Dependency Specification.* Retrieved January 23, 2021 from https://www.python.org/dev/peps/pep-0440/

[35] Hyunmin Seo, Caitlin Sadowski, Sebastian G. Elbaum, Edward Aftandilian, and Robert W. Bowdidge. 2014. Programmers' build errors: a case study (at google). In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 724–734. https://doi.org/10.1145/2568225.2568255

[36] Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, and Tanu Malik. 2017. Sciunits: Reusable Research Objects. In *13th IEEE International Conference on e-Science, e-Science 2017, Auckland, New Zealand, October 24-27, 2017*. IEEE Computer Society, 374–383. https://doi.org/10.1109/eScience.2017.51

[37] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 339–349. https://doi.org/10.1109/ICSE.2019.00048

[38] Marat Valiev, Bogdan Vasilescu, and James D. Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 644–655. https://doi.org/10.1145/3236024.3236062

[39] Brandon Vigliarolo. 2020. *Python overtakes Java to become the second-most popular programming language.* Retrieved January 24, 2021 from https://www.techrepublic.com/article/python-overtakes-java-to-become-the-second-most-popular-programming-language/

[40] 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 *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 125–135. https://doi.org/10.1145/3377811.3380426

[41] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1556–1560. https://doi.org/10.1145/3368089.3417943