Fixing Dependency Errors for Python Build Reproducibility

By

SUCHITA MUKHERJEE

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Cindy Rubio-González

Premkumar T. Devanbu

Aditya V. Thakur

Committee in Charge

2021

To Ma, Baba and Puchu

Contents

List of Figures

List of Tables

Abstract of the Thesis

Fixing Dependency Errors for Python Build Reproducibility

Software reproducibility is important for re-usability and the cumulative progress of research. An important manifestation of unreproducible software is the outcome of software builds changing over time. While enhancing code reuse, the usage of open-source dependency packages hosted on centralized software repositories like PyPI can have adverse effects on build reproducibility. Frequent updates of these packages often cause their latest versions to have breaking changes for applications using them. Large Python applications risk their historical builds to become 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 such as parsing unstructured build logs, finding transitive dependencies, and dealing with an exponential search space of dependency versions. In this thesis, 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 version errors. The ability of PyDFix to fix unreproducible builds 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.

Acknowledgments

The continued mentorship and support of my advisor, Professor Cindy Rubio-González made this thesis possible and for that I would like to express my gratitude. Her guidance has transformed my experience in the world of research and expanded my appreciation of scientific rigor.

I would also like to thank Professor Aditya V. Thakur for his detailed and insightful comments during the developmental stages of this thesis and Professor Premkumar Devanbu for his encouragement, and both for taking the time to review my work.

Abigail Alamanza, who participated in exploratoration and brainstorming phases has also helped shape the direction of this work.

Last but not the least, I acknowledge the indispensable support provided by my parents, my partner and the UC Davis community all of whom made it possible to accomplish this thesis.

Chapter 1

Introduction

Reproducibility of software artifacts is one of the most significant and consistent 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 [8] and PyPI [13]. 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. [36] found in their study of Java and C++ build failures that nearly half of all build errors are caused by dependencies. When evaluating Python gists available on GitHub, Horton and Parnin [29] 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 [32]. Tomassi et al. [38] 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.

1.1 Advances in Reproducibility

The software engineering community has made great strides towards achieving reproducibility. The advent of tools like Docker [5] and Kubernetes [7] 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 modified 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.

1.2 Reproducibility for Python

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 [15]. 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 their dependencies as these 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.

1.3 Reproducibility for Bug Datasets

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 evaluation of fault localization and repair techniques. Two recent Python bug datasets BugSwarm [38] and BugsInPy [40] 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. Neither dataset has considered the impact of configuration drift as described by Horton and Parnin [31] as the phenomenon of a code snippet going out-of-date because APIs it depends on experience breaking change over time. Consequently, these datasets have not addressed configuration drift in the design of their tool and in their measures to preserve reproducibility of the collected artifacts. In our analysis of 1,981 BugSwarm builds, we find a total of 44,012 installed packages, comprising of 22,264 project and 21,748 transitive dependencies, respectively. 62.4% of project dependencies are pinned while only 4.5% of transitive dependencies are pinned. Overall, 38.1%, 33.7% and 28.1% dependencies were found to be constrained, pinned and unconstrained, respectively.

1.4 Overview of Approach

In this thesis, we look into the usage of dependency packages and their version specifications in builds of open-source Python projects. We focus on current build logs that contain dependency-related errors which were not present in the original logs, thus making the build unreproducible. 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 status 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 commit due to package dependencies can utilize PyDFix.

1.5 Overview of Related Work

Recent approaches [34, 27, 33] repair build failures in Java projects, but do not focus on reproducibility. In terms of Python, DockerizeMe [30] works on inferring environment configurations for Python gists, and V2 [31] 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 [37] 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.

1.6 Contributions

We evaluate PyDFix on a total of 2,702 Python builds from the BugSwarm and BugsInPy datasets. PyDFix identifies a total of 1,921 builds as being unreproducible due to dependency issues. These include 67.2% of analyzed builds from BugSwarm, and 84.90% of analyzed builds from BugsInPy. PyDFix successfully computes a complete fix for 859 builds (40.91% and 55.53% 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 (32.06% of BugSwarm builds and 35.39% of BugsInPy builds).

The main contributions of this thesis are:

- 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 study dependency usage, version specifications and inclusion of transitive dependencies in Python projects (chapter 3).
- 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 5.2 and 5.3).

Chapter 2

Background and Motivation

This section provides background on Python dependencies, an example of broken dependencies, 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 [19] can also contain Python dependency package declarations. Several virtual environment management packages like `tox` [17] and `pyenv` [12] allow configurations to declare dependencies. Moreover, there exist multiple package managers for Python, the two most popular being `pip` [9] and `conda` [3]. We only consider packages installed using `pip` as it is Python's standard package manager [10].

PEP 440 [35] and PEP 508 [22] provide detailed information about the versioning system of Python packages as well as the types of dependency declaration and version specification 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 with the dependency declaration.

Dependency packages with a *pinned version* can affect a build outcome if the package gets removed from PyPI, or if its versioning system changes. For example, the package `pytest-capturelog`, which is documented in libraries.io [14] no longer exists in PyPI. Another example is `pyatom`, which still exists in PyPI but whose versioning system completely changed in January 2020 [11]. 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 Broken Dependencies

Figure 2.1 shows an example of broken dependencies in a Python project. Figure 2.1a shows the current build outcome for a historical commit [1] of the GitHub repository `cloudify-system-tests` [2]. The build failure message indicates that the failure is due to the package `stevedore` [16]. However, this dependency is not declared within the application. In the log line documenting the installation of `stevedore` shown in Figure 2.1b, we observe that the package `stevedore` is actually a transitive dependency, required by the package `openstacksdk`. Although `cloudify-system-tests`'s source code has pinned a

```
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.
```

(a) Initial build error

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

(b) 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.
```

(c) Build Error after pinning correct version of `stevedore`

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

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

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

(e) Final patch with pinned problematic dependencies

```
  The command "flake8 ." exited with 0.
  Done.  Your build exited with 0.
```

(f) Build fixed

Figure 2.1: Motivating Example

An example requiring multiple version specification changes to restore build reproducibility.

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 greater than equal to 3.6 and hence, causes an error when pulled into the build process for this commit.

As shown in Figure 2.1c, pinning `stevedore` does not repair the build completely and another error is encountered. While the error due to `stevedore` was 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` [4]. Inspecting the installation steps of the logs show that `configparser` was indeed installed and it is a transitive dependency of `flake8` [6]. 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 greater than 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(Figure 2.1f).

2.3 Terminology

Here we introduce some terms to be used in the following sections.

- **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 Figure 2.1, the original build outcome was a successful build, but the current outcome was a build error, shown in Figure 2.1a.

- **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 2.1e.

- **Patch Candidate:** For PyDFix, a patch candidate consists of a single dependency package and its suitable version. A patch consists of one or more patch candidates.

- **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 2.1b the dependency chain for the transitive dependency `stevedore` is `openstacksdk==0.9.13`→`cl`

- **Triggering Commit:** A commit that triggered a build.

Chapter 3

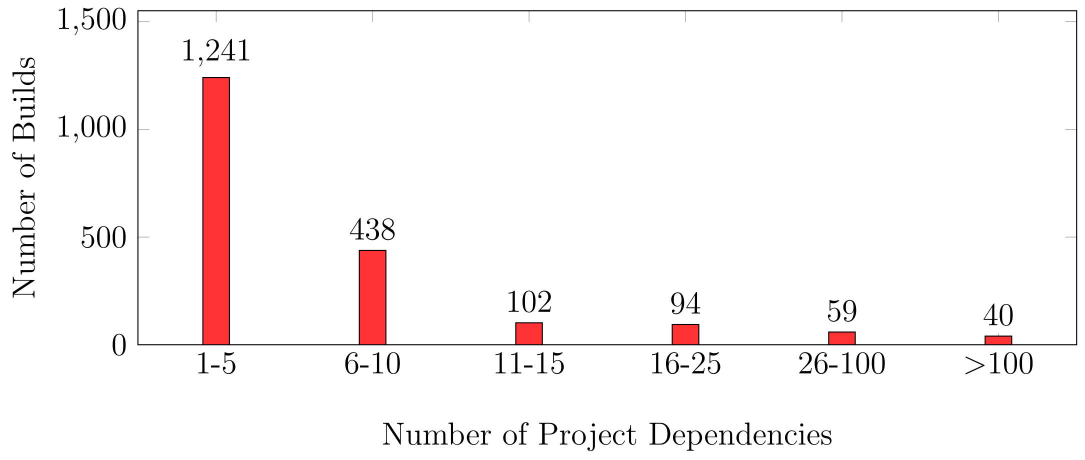
Dependency Version Specifications

To further motivate our work, we investigate the frequency in different dependency version specifications used in 1,119 BugSwarm artifacts, i.e., 2,238 builds after excluding 173 artifacts for which the original logs are not available.[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 redundant 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.

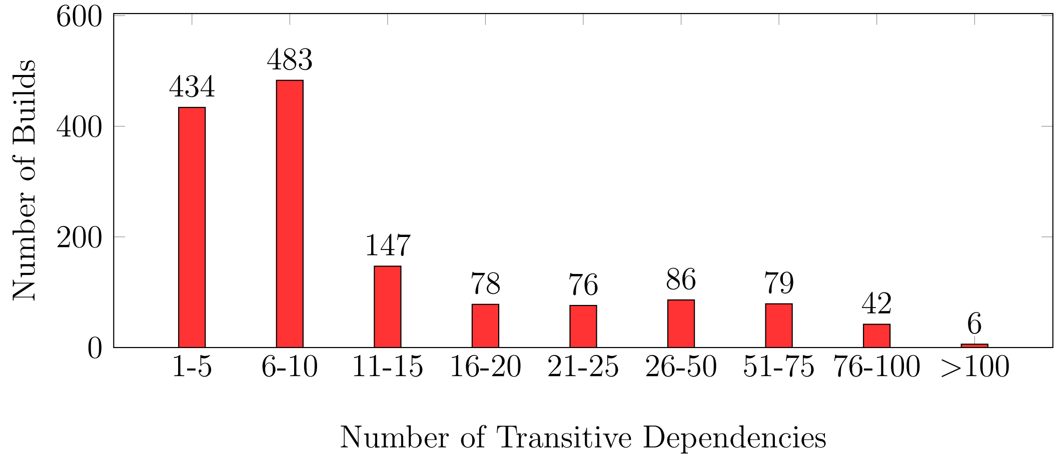
3.1 Frequency of Version Specifications

We observe a widespread use of dependency packages with a total of 44,012 instances of package installations across all builds. Figures 3.1a and 3.1b show the count of builds containing a range of project and transitive dependencies, respectively. We found that 256 builds (11.43%) did not fetch any packages from the PyPI index, i.e., these builds have zero dependencies. Additionally, a total of 550 builds (24.57%) 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

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



(a) Builds by number of project dependencies.



(b) Builds by number of transitive dependencies.

Figure 3.1: Builds by number of project and transitive dependencies.

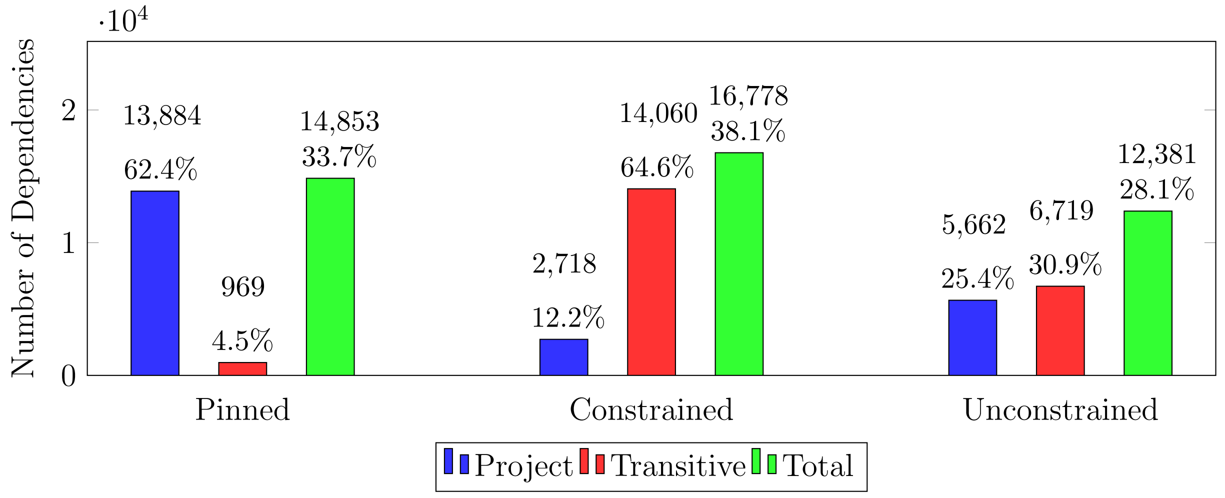
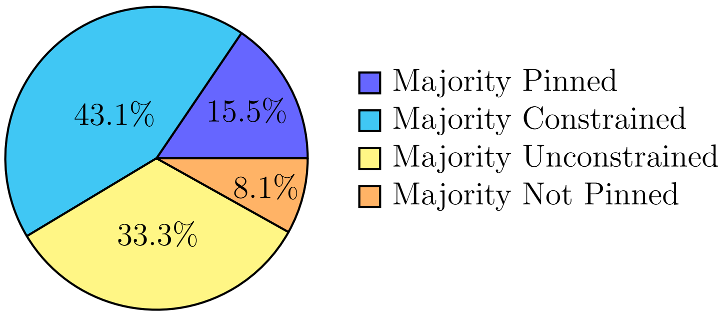
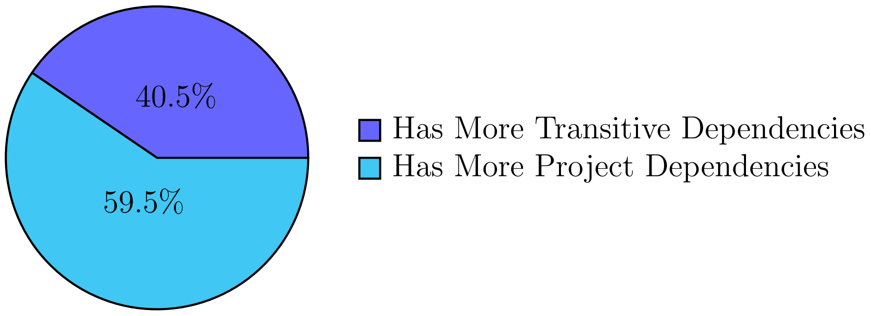


Figure 3.2: Dependencies by Version Specification



(a) Breakdown of Builds by Majority Dependency Version Specification



(b) Breakdown of Builds by Majority Type of Dependency

Figure 3.3: Breakdown of Builds by Dependency Version Specifications and Dependency Type

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 3.2 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 25.4% are unconstrained and 12.2% are constrained. However, most of the transitive dependencies are constrained while only 4.5% are pinned. A significant percentage of transitive dependencies, 30.9% are also unconstrained. While looking at the total dependencies including both types, the largest portion which is 38.1% is constrained followed by 33.7% pinned and 28.1% unconstrained. This highlights the need for developing tools like PyDFix to address dependency-related errors.

We also find that only 15.5% of the builds have most of their required dependencies pinned, 43.1% of the builds have the majority of their required dependencies constrained, and 33.3% builds have the majority of the package dependencies unconstrained as shown in Figure 3.3a. Finally, from Figure 3.3 we observe that 40.5% 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.

Finding: 44,012 packages are installed across 2,238 builds. 1,034 builds have 10+ project or transitive dependencies. While most project dependencies were pinned at 62.4%, only 4.5% of transitive dependencies are pinned. We find 12,381 (28.1%) total unconstrained and 16,778 (38.1%) constrained dependencies, which increase the likelihood of build breakage. The 14,853 (33.7%) pinned dependencies can also lead to unreproducibility if removed from the package index. 40.5% 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. 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.

Chapter 4

Technical Approach

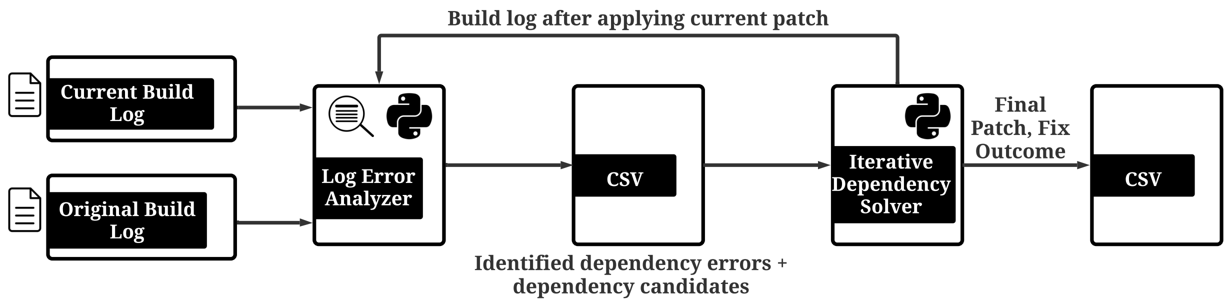

Build log after applying current patch
Current Build Log
Original Build Log
Log Error Analyzer
CSV
Iterative Dependency Solver
Final Patch, Fix Outcome
CSV
Identified dependency errors + dependency candidates

Figure 4.1: PyDFix Workflow

PyDFix is shown in Figure 4.1. The two main components are LogErrorAnalyzer for identifying dependency-related errors causing 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 keeps re-running the build with intermediate patches and analyzing 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.

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 TravisCI build logs we cannot depend on any pre-defined section of log messages showing the reasons for build failure. LogErrorAnalyzer analyzes TravisCI 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?

4.1.1 Error Patterns

To understand which error messages in a build log indicate failure related to dependency packages, we manually inspected the TravisCI 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 which are listen in Table 4.1

The error patterns in Table 4.1 indicate that dependency errors are related to the failure of `pip install`, failure to setup `Python` egg, 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 which 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`

Table 4.1: Log Error Regex Patterns

Error Type	Error Pattern
REQUIREMENTS_TXT_NEEDS_PYTHON	requirements.txt needs (.*) python
PIP_INSTALL_FAILED_AND_EXITED	The command "pip3? install (.*) failed and exited with (.*) during
SETUP_EGG_INFO_FAILED	Command "python3? setup.py egg_info" failed
PIP_INSTALL_FAILED_TIMES	The command "pip3? install (.*) failed (.*) times
PYTHON_SETUP_FAILED_EXITED	The command "python3? (.*) failed and exited (.*)
VIRTUAL_ENV_ERROR	virtualenv.py: error
PACKAGE_REQUIRES_PYTHON	(.*) requires Python (.*)
PACKAGE_REQUIRES_DIFF_PYTHON	(.*) requires a different Python (.*)
SCRIPT_REQUIRES_PYTHON	ERROR: this script requires Python (.*)
PACKAGE_IN_REQ_TXT_NEEDS_PYTHON	(.*) from requirements.txt needs (.*)
IMPORT_ERROR	ImportError (.*)
MODULE_NOT_FOUND_ERROR	ModuleNotFoundError: (.*)
TYPE_ERROR	TypeError (.*)
FLAKE_ERROR	The command "(.*)flake8 (.*) exited with 1
NO_MODULE_FOUND_-ERROR	No module named (.*)
CYTHON_ERROR	Cannot cythonize without Cython installed(.*)
COMMAND_NOT_FOUND_ERROR	(.*) : command not found

Table 4.2: Log Patterns for Package Installation

Package Installation Pattern
ˆCollecting(.*)
ˆSearching(.*)
ˆDownloading(.*)
ˆRequirement already satisfied:(.*)
ˆBest Match(.*)

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 4.2. These patterns are used by LogErrorAnalyzer 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 exist 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 original logs, the error message itself and the associated error trace is parsed to extract packages

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 $errorLines$, $candidates$, $fileNames$
```

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 fixes. 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.

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. Every iteration, a previously unapplied patch candidate is added to the final accepted patch or discarded based on the build outcome. Algorithm 3 is used to reason with the progress of the newly added patch candidate depending on build outcome. Thus, pinned dependencies are incrementally added to the final accepted patch until encountering either one of the terminal states in Table 4.3 (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 (Line 1) and Algorithm 3 (Line 21), patch candidates are generated at the beginning, and again every time the patch candidates need to be updated due to new

Algorithm 2: IterativeDependencySolver

```
   Input: buildDetails, errors, candidates
   Output: A patch that pins dependencies
1  $patchCandidates = genPatches(candidates)$
2  $acceptedPatch \leftarrow [], currentPatch \leftarrow [], fixOutcome = None$
3  while $True$ do
4      $currentPatch = acceptedPatch$
5      $newPatchCandidate = None$
       /* Patch Candidate Selection */
6      $newPatchCandidate \leftarrow getUnappliedPatch(patchCandidates, acceptedPatch)$
7      if $newPatchCandidate == None$ then
           /* Encountered terminal state */
8          $fixOutcome \leftarrow$ Exhausted All Options
9          break
       /* Patch Candidate Application */
10     $newBuildOutcome, newBuildLog \leftarrow runBuildJob(currentPatch, buildDetails)$
       /* Patch Impact Evaluation */
11     $fixOutcome, accpetedPatch, updatedCandidates, updatedErrors,$
        $updatedPatchCandidates \leftarrow EvaluatePatchImpact(newBuildOutcome,$
        $acceptedPatch, currentPatch, newBuildLog)$
12 end
13 if $fixOutcome$ then // Terminal State
14     break
15 else if $updatedCandidates$ then// New Patches
16     $candidates \leftarrow updatedCandidates, errors \leftarrow updatedErrors$
17     $patchCandidates \leftarrow updatedPatchCandidates$
18 return $fixOutcome, acceptedPatch$
```

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.
- 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 Line 7).

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

Table 4.3: 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

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 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.

4.2.3 Evaluating Patch Impact

After the patched build is run, the log status is checked (Algorithm 3) and the algorithm either exits with one of the fix outcomes in Table 4.3, or proceeds to the next iteration. If the build

Algorithm 3: EvaluatePatchImpact

```
Input: newBuildOutcome, acceptedPatch, currentPatch, newBuildLog,
       newPatchCandidate
Output: fixOutcome, acceptedPatch, candidates, errors, patchCandidates
1  $fixOutcome \leftarrow None$
2  if $newBuildOutcome == SUCCESSFUL$ then // Terminal state
3      $acceptedPatch \leftarrow currentPatch$
4      $fixOutcome \leftarrow$ Successfully Fixed Build
5      return $fixOutcome, acceptedPatch, None, None, None$
6  else if $newBuildOutcome == NO\_CHANGE$ then// Discard
7      return $fixOutcome, acceptedPatch, None, None, None$
8  else if $newBuildOutcome == DIFFERENT\_ERROR$ then
9      $errorType, newErrors, newCandidates \leftarrow$
        $runLogErrorAnalyzer(newBuildLog)$
10     if $errorType\ ! = DEPENDENCY\_ERROR$ or no $newCandidates$ then
        // Terminal state
11         if $newErrors$ in $originalLog$ then // Same error as original build
12             $fixOutcome =$ Restored to original error
13         else // Partial Fix
14             $fixOutcome =$ No longer a dependency error
15         $acceptedPatch \leftarrow currentPatch$
16         return $fixOutcome, acceptedPatch, None, None, None$
17     else if $newCandidates[0] == newPatchCandidate$ then// Discard
18         return $fixOutcome, acceptedPatch, None, None, None$
19     else // Patch Candidates Elimination
20         $candidates \leftarrow newCandidates, errors \leftarrow newErrors$
21         $acceptedPatch \leftarrow currentPatch$
22         $patchCandidates \leftarrow genPatches(candidates)$
23         return $fixOutcome, acceptedPatch, candidates, errors, patchCandidates$
```

is successful, the current patch is accepted as final and the algorithm returns the outcome "Successfully Fixed Build". If the build error remains unchanged, the last dependency package added to the patch is rejected and the next iteration of Algorithm 2 starts. If a new build error is encountered, Algorithm 3 runs LogErrorAnalyzer to analyze the new build log. If LogErrorAnalyzer outputs no possible candidates or no identified dependency errors, and the build terminating error is the same as in the original log, then the final fix outcome is "Restored to original error" (Algorithm 3 Lines 10-12). If the new error is different from the terminating error in the original build, the algorithm returns the current patch as the final patch and the outcome "No longer recognized as dependency error" (Algorithm 3, Lines 13-15). In case 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 3, Lines 16-17). Otherwise, the current patch list is accepted as the correct one so far (Algorithm 3, Lines 19-23).

4.2.4 Elimination of Patch Candidates

When IterativeDependencySolver reaches a build outcome that still contains a dependency-related 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 15-17 and Algorithm 3 Lines 19-23). 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.

Chapter 5

Experimental Evaluation

This experimental evaluation answers the following questions:

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

RQ2 How effective is PyDFix in restoring unreproducible builds to reproducible status?

5.1 Experimental Setup

To evaluate our approach to fix unreproducible builds by resolving dependency issues, we use the software artifacts from two bug datasets built from real-world open-source projects: BugSwarm [38] and BugsInPy [40]. 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 used 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` [18], as done by the BugSwarm infrastructure [38]. We locally run 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 [19] 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, like 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. All experiments were run on a workstation with 88 Intel(R) Xeon(R) Gold 2.10GHz CPUs and 384GB RAM.

5.2 Impact of Dependency Errors

In chapter 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 issues and not all causes of unreproducibility are due to dependency packages. As explained in Section 3.2, there can be a number of

Table 5.1: 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.

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	1921 (71.1%)

different reasons for unreproducible builds and in this study we focus on identification of builds that have become unreproducible due to dependency issues. 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 original and current build logs as input for each build to be analyzed and provides a list of dependency errors found, a list of possible candidate dependencies and identified dependency specification files as output. We analyzed a total of 2,106 and 596 builds from BugSwarm and BugsInPy, respectively. As shown in Table 5.1, LogErrorAnalyzer identified 1,415 (67.2%) BugSwarm builds and 506 (84.9%) BugsInPy builds as having dependency-related errors not present in the original log.

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 issues 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 candidates. This observation further emphasizes the need for automated fixing of unreproducible builds due to dependency issues.

Table 5.2: 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 unreproducible did no longer had dependency errors.

Patch Result	Identified	Complete Fix	Partial Fix
BugSwarm	1,415	578 (40.91%)	453 (32.06%)
BugsInPy	506	281 (55.53%)	179 (35.39%)
Total	1,921	859 (44.7%)	632 (32.9%)

RQ1 Answer: For BugSwarm, out of 2,106 builds analyzed, 1,415 (67.2%) were identified as having dependency issues. For BugsInPy out of 596 builds analyzed, 506 (84.9%) were found to have dependency issues. This shows that breakage due to dependencies is quite common.

5.3 Evaluation of Fixing Dependency Issues

We ran PyDFix on the broken builds identified by LogErrorAnalyzer as dependency related: 1,415 and 506 builds from BugSwarm and BugsInPy, respectively. Table 5.2 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 (40.91%), and partial fixes for 453 (32.06%) of 1,415 identified builds. While for BugsInPy, PyDFix found complete fixes for 281 (55.53%), and partial fixes for 179 (35.39%) out of 506 identified builds. Overall, PyDFix was able to create complete fixes for 859 (44.72%) and partial fixes for an additional 632 (32.90%) of the identified artifacts. 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 5.3 shows that a total of 2,497 dependency packages were included in patches, with the largest patch including 22 dependencies. This showcases how time

Table 5.3: BugSwarm Patch Metrics
"All" presents metrics for all patches, Columns "Complete" and "Partial" list complete and partial fixes, "Max" and "Average" show the maximum and average number of dependencies in a patches.

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

consuming and difficult it can be to manually resolve an unreproducible build with multiple dependency issues. 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 while 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.

For 22.4% of builds PyDFix was not successful in finding a patch. A preliminary manual inspection revealed that this is caused by an incorrect identification of the build as having dependency-related errors, or to not identifying the correct packages to patch because of log errors not considered by PyDFix.

Due to the large number of builds to process, serial execution of each process was extremely time consuming. Since the processing of each build is completely independent, PyDFix was implemented using Python multiprocessing, processing each build in a separate process. The average, median and maximum time taken by PyDFix to fix an artifact is 70.64, 16.07 and 414.14 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 (40.91%) and a partial fix for 453 (32.06%) builds. For BugsInPy, out of 506 identified builds, 281 (55.53%) builds were made reproducible while 179 builds (35.39%) were partially fixed. The average and median time PyDFix required to fix artifacts are 70.64 and 16.07 minutes, respectively.

Chapter 6

Threats to Validity

In this study, we analyze 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 contain dependency-related build errors that are not included in our set. Note that such errors could be easily added to PyDFix, which would improve its effectiveness. Finally, there also exists the possibility that changing a version specification of a dependency may resolve build errors but changes application behavior. We try to 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.

Chapter 7

Related Work

7.1 Reusable Research Objects

The need for reproducibility in computational experiments is extremely important for researchers to collaborate and build on others' work. Containerization of artifacts using platforms like Docker have made reproducibility easier and faster but still require substantial user actions. As mentioned earlier, containerization does not guarantee that the correct dependencies are available as it depends on the metadata provided in the software project. SCIUNIT [37] provides an open-source command-line interface for creating reusable research objects that are lightweight and robust. SCIUNIT uses OS-specific monitoring utilities to automatically generate the required dependencies in an application during its runtime. It also facilitates versioned storage using a common block-based storage based on content de-duplication techniques.

Similar to SCIUNIT, REPROZIP[21] is an open-source desktop application that traces system calls to identify the files essential for reproducing the application. With this meta-data gathered on the fly, ReproZip creates a distributable bundle which provides the flexibility of being reproduced in chroot environments, Vagrant-built VMs and Docker Containers. However, in comparison with SCIUNIT REPROZIP does not provide support for versioned storage and also requires the download of a separate "unzipper" application.

The method used in this project is different from these previous approaches in that PYDFIX does not require to be running simultaneously during the runtime of the application. However, the idea of enhancing space utilization can be adapted by having a common cache of pinned dependencies for different Docker images to use. This project is not currently focused on how to prevent build breakage or maintain build stability, but instead on repairing those builds that break due to incompatible dependency versions.

7.2 Automatic Build Repair

BuildMedic[34] is a tool to automatically repair dependency related build breakages for Maven builds in Java projects. The data gathering procedure employed by BuildMedic is similar to PyDFix, as it uses regular expression matching in Maven LogAnalyzer to analyze the execution logs of Maven build to identify artifacts that showed dependency related issues. BuildMedic narrows down to three repair strategies, namely *Version Update*, *Dependency Delete* and *Add Repository*. However, BuildMedic is not focussed 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. Macho et al. [34]'s focus 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.

An approach for automated program repair using build history was introduced by HireBuild[27]. HireBuild identifies unique generalized patterns in how build scripts were modified to repair build failure from historical build breakage data. These identified patterns are matched with the generalized AST nodes of the buggy build script. A patch is generated for all the patterns that match the faulty build script and are applied based on a descending order of priority. The patches are applied till they are exhausted or the build is fixed. HireBuild focuses only on modifying Gradle build scripts for fixing build failure and not on resolving the dependency issues or reproducibility of builds. Moreover, 23% of builds which could not be resolved by HireBuild were due to Dependency Resolution failures which brings to focus the need for PyDFix and the focus on build breakage caused by problems arising from package dependencies.

Lou et al. [33] evaluate the state-of-the-art HireBuild on an extended dataset of real world build failure and concluded that the history-driven build fixing technique implemented in HireBuild may suffer from the overfitting problem and the historical information plays a marginal role in a successful fix. The authors present HoBuff which utilizes the current information present in build logs for generating build fix patches for erroneous build scripts. After identifying the bug revealing statement from either explicit mention in the build logs or using Levenshtein distance of element names in the build script with the configuration name extracted from the build logs, HoBuff finds the set of potential root cause statement through interprocedural dataflow analysis. A list of patch candidates are generated using fixing operators *Insert, Delete and Update* and fixing ingredients gathered by searching within the project for internal configuration elements or by searching Gradle Central Repository or Gradle DSL Documentation for external configuration elements like third party libraries. All patch candidates are applied to the associated root cause statement till the build is fixed or the candidates are exhausted. HoBuff is evaluated to be capable of fixing 2X real world reproducible build failures than HireBuild and is faster. Neither of HireBuild nor HoBuff address the reproducibility of the builds they fix and are focussed on implementing their solution for the Gradle build tool for Java projects using Maven. Gradle makes it relatively easier to identify the cause of build failure owing to its dedicated section of errors and exceptions in its build logs. PyDFix cannot benefit from such specifics in the build logs because most python applications do not need or use build tools.

7.3 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.

This tool draws upon the information source created through an iteration of pre-processing of the existing packages in `PyPi` and project configuration files on `GitHub`. DockerizeMe creates environment configurations when it is not available as in the case of Python gists. While the algorithm attempts to understand required package dependencies and also maintain their installation order, it does not handle the issue of installing the correct version of the package. DockerizeMe also 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[31], an extension of DockerizeMe adds support for package versions and detects out-of-date code snippets by detecting configuration drift. V2 which works on Python gists as well as Jupyter notebooks, creates initial candidate environment configurations based on the inferred Python version of the code. Using a *Feedback-Directed Search* the correct versions of package dependencies included in each candidate environment is found.

Both DockerizeMe and V2 focus on Python dependency issues similar to our approach, but these tools can only be used for code snippets and V2 additionally works on Jupyter notebooks. Their approach can be time consuming and less effective for fixing unreproducible software artifacts of a larger scale like those that are present in the BugSwarm and BugsInPy datasets. This is supported by the fact that Horton and Parnin [31] excluded all code snippets having more than 10 dependencies from their dataset used for evaluating V2. 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.

7.4 Dependency Graphs

VeriBuild [26] approaches the problem of declared dependencies and actual dependencies in Makefile build scripts by narrowing it down to missing and redundant dependencies. A new type of dependency graph is proposed, namely unified dependency graph which captured both static and dynamic dependencies thus making the identification and analysis of dependency errors easier. VeriBuild[26] only addresses discrepancies between dependencies of build targets while PyDFix fixes unreproducibility of builds caused by incompatible dependency package versions.

The studies presented in [28], [32] and [23] provide insightful information about dependency packages, their networks, evolution and impact. However, the focus of our study is not understanding dependency networks. We show through our collection of metrics, how widespread the use of dependency packages is, how frequently transitive dependencies are included and the impact of their dependency version specifications.

7.5 Log Parsing to Detect Build Anomalies

A rich body of work exists for parsing of system logs with the focus concentrated on generating features from raw logs, building execution models to compare with future execution patterns and identifying system dependencies. Spell [24] is a parser for streaming log data which parses unstructured log messages into structured message types and parameters which facilitates the storage and querying of log messages. Each incoming log entry is tokenized using system defined delimiters and the sequence of tokens are compared to existing sequences using the longest common subsequence (LCS), clustering together the sequences whose LCS is more than half the length of the incoming sequence. The intuition for this technique is that the log entries coming from the same print statements will have a constant LCS once the parameters are abstracted.

DeepLog [25] uses the intuition of applying natural language processing for parsing logs since log entries have distinct patterns and adhere to a strict grammar. DeepLog framework extracts log keys and parameter values from log entries and trains a LSTM neural network to detect execution path anomaly as well as parameter value and performance anomaly. Both Spell and DeepLog both are used for parsing system logs while PyDFix works with build logs from TravisCI, however it can be argued that parsing system logs is similar to build logs although they differ in the ways that they are generated. The main difference is that PyDFix's objective while parsing logs is not to identify deviations from normal execution but to identify a particular type of error. Due to this, it is not necessary for PyDFix to create a vocabulary of all log sequences or learn patterns in the order of log sequences.

Chapter 8

Conclusion and Future Work

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. While regular expression matching is an effective way of identifying known patterns, a set of error patterns have to be manually created for the basis of these expressions. This method also requires new errors to be added to the set of errors each time a new error is manually identified. Automated log parsing techniques can be used for identification of dependency errors which would improve our identification of unreproducible builds due to dependency errors, and the precision of PyDFix at identifying candidate dependencies.

References

[1] cloudify-system-tests triggering commit. https://github.com/cloudify-cosmo/cloudify-system-tests/tree/bf27ad94b2fb11183beb2f374f5eb06b7af31bdf, Accessed 2021.

[2] cloudify-system-tests GitHub repo. https://github.com/cloudify-cosmo/cloudify-system-tests, Accessed 2021.

[3] Conda. https://docs.conda.io/en/latest/, Accessed 2021.

[4] configparser. https://pypi.org/project/configparser/, Accessed 2021.

[5] Docker. https://www.docker.com/, Accessed 2021.

[6] flake8. https://pypi.org/project/flake8/, Accessed 2021.

[7] Kubernetes. https://kubernetes.io/, Accessed 2021.

[8] Maven Central Repository. https://repo1.maven.org/maven2/, Accessed 2021.

[9] pip. https://pypi.org/project/pip/, Accessed 2021.

[10] What Is Pip? A Guide for New Pythonistas. https://realpython.com/what-is-pip/, Accessed 2021.

[11] pyatom versions. https://libraries.io/pypi/pyatom/versions, Accessed 2021.

[12] pyenv. https://pypi.org/project/pyenv/, Accessed 2021.

[13] Python Package Index. https://pypi.org/, Accessed 2021.

[14] pytest-capturelog. https://libraries.io/pypi/pytest-capturelog, Accessed 2021.

[15] What's New In Python 3.0. https://docs.python.org/3/whatsnew/3.0.html, Accessed 2021.

[16] stevedore. https://pypi.org/project/stevedore/, Accessed 2021.

[17] tox. `https://pypi.org/project/tox/`, Accessed 2021.

[18] travis-build. `https://github.com/travis-ci/travis-build`, Accessed 2021.

[19] Travis CI. `https://travis-ci.org/`, Accessed 2021.

[20] B. Anda, D. I. K. Sjøberg, and A. Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Trans. Software Eng.*, 35(3):407–429, 2009. doi: 10.1109/TSE.2008.89. URL `https://doi.org/10.1109/TSE.2008.89`.

[21] F. Chirigati, R. Rampin, D. E. Shasha, and J. Freire. Reprozip: Computational reproducibility with ease. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2085–2088. ACM, 2016. doi: 10.1145/2882903.2899401. URL `https://doi.org/10.1145/2882903.2899401`.

[22] R. Collins. Pep 508 – dependency specification for python software packages, Jan. 2015. URL `https://www.python.org/dev/peps/pep-0508/`.

[23] A. Decan, T. Mens, and P. Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *CoRR*, abs/1710.04936, 2017. URL `http://arxiv.org/abs/1710.04936`.

[24] M. Du and F. Li. Spell: Streaming parsing of system event logs. In F. Bonchi, J. Domingo-Ferrer, R. Baeza-Yates, Z. Zhou, and X. Wu, editors, *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, pages 859–864. IEEE Computer Society, 2016. doi: 10.1109/ICDM.2016.0103. URL `https://doi.org/10.1109/ICDM.2016.0103`.

[25] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November*

03, 2017, pages 1285–1298. ACM, 2017. doi: 10.1145/3133956.3134015. URL `https://doi.org/10.1145/3133956.3134015`.

[26] G. Fan, C. Wang, R. Wu, X. Xiao, Q. Shi, and C. Zhang. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In S. Khurshid and C. S. Pasareanu, editors, *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 463–474. ACM, 2020. doi: 10.1145/3395363.3397388. URL `https://doi.org/10.1145/3395363.3397388`.

[27] F. Hassan and X. Wang. Hirebuild: an automatic approach to history-driven repair of build scripts. In M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1078–1089. ACM, 2018. doi: 10.1145/3180155.3180181. URL `https://doi.org/10.1145/3180155.3180181`.

[28] J. Hejderup, A. van Deursen, and G. Gousios. Software ecosystem call graph for dependency management. In A. Zisman and S. Apel, editors, *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 101–104. ACM, 2018. doi: 10.1145/3183399.3183417. URL `https://doi.org/10.1145/3183399.3183417`.

[29] E. Horton and C. Parnin. 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*, pages 217–227. IEEE Computer Society, 2018. doi: 10.1109/ICSME.2018.00031. URL `https://doi.org/10.1109/ICSME.2018.00031`.

[30] E. Horton and C. Parnin. Dockerizeme: automatic inference of environment dependencies for python code snippets. In J. M. Atlee, T. Bultan, and J. Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal,*

QC, Canada, May 25-31, 2019, pages 328–338. IEEE / ACM, 2019. doi: 10.1109/ICSE.2019.00047. URL `https://doi.org/10.1109/ICSE.2019.00047`.

[31] E. Horton and C. Parnin. 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*, pages 477–488. IEEE, 2019. doi: 10.1109/ASE.2019.00052. URL `https://doi.org/10.1109/ASE.2019.00052`.

[32] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl. Structure and evolution of package dependency networks. In J. M. González-Barahona, A. Hindle, and L. Tan, editors, *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 102–112. IEEE Computer Society, 2017. doi: 10.1109/MSR.2017.55. URL `https://doi.org/10.1109/MSR.2017.55`.

[33] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang. History-driven build failure fixing: how far are we? In D. Zhang and A. Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 43–54. ACM, 2019. doi: 10.1145/3293882.3330578. URL `https://doi.org/10.1145/3293882.3330578`.

[34] C. Macho, S. McIntosh, and M. Pinzger. Automatically repairing dependency-related build breakage. In R. Oliveto, M. D. Penta, and D. C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 106–117. IEEE Computer Society, 2018. doi: 10.1109/SANER.2018.8330201. URL `https://doi.org/10.1109/SANER.2018.8330201`.

[35] D. S. Nick Coghlan. Pep 440 – version identification and dependency specification, Jan. 2013. URL `https://www.python.org/dev/peps/pep-0440/`.

[36] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge. Programmers' build errors: a case study (at google). In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14,*

Hyderabad, India - May 31 - June 07, 2014, pages 724–734. ACM, 2014. doi: 10.1145/2568225.2568255. URL `https://doi.org/10.1145/2568225.2568255`.

[37] D. H. T. That, G. Fils, Z. Yuan, and T. Malik. Sciunits: Reusable research objects. In *13th IEEE International Conference on e-Science, e-Science 2017, Auckland, New Zealand, October 24-27, 2017*, pages 374–383. IEEE Computer Society, 2017. doi: 10.1109/eScience.2017.51. URL `https://doi.org/10.1109/eScience.2017.51`.

[38] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In J. M. Atlee, T. Bultan, and J. Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 339–349. IEEE / ACM, 2019. doi: 10.1109/ICSE.2019.00048. URL `https://doi.org/10.1109/ICSE.2019.00048`.

[39] B. Vigliarolo. Python overtakes java to become the second-most popular programming language, Jan. 2020. URL `https://www.techrepublic.com/article/python-overtakes-java-to-become-the-second-most-popular-programming-language/`.

[40] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In P. Devanbu, M. B. Cohen, and T. Zimmermann, editors, *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*, pages 1556–1560. ACM, 2020. doi: 10.1145/3368089.3417943. URL `https://doi.org/10.1145/3368089.3417943`.