# Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem

Anonymous Author(s)

## ABSTRACT

PyPI is a major central repository for Python projects. It has indexed millions of libraries to allow developers to automatically download and install dependencies of their projects based on the specified version constraints. Despite the convenience brought by automation, version constraints in Python projects can easily conflict, resulting in build failures. We refer to such conflict issues as dependency conflict (DC) issues. Although DC issues are common in Python projects, developers lack tool support to gain a comprehensive knowledge of the version constraints specified by different projects and diagnose the root causes of these issues. In this paper, we conducted an empirical study on 235 real-world DC issues collected from 124 popular Python projects. We studied the manifestation patterns and fixing strategies of these issues and found several key factors leading to potential DC issues and their regressions. Based on our findings, we designed and implemented Watchman, a technique to continuously monitor dependency conflicts for the PyPI ecosystem. In our evaluation, Watchman analyzed PyPI snapshots between 11 Jul 2019 and 16 Aug 2019, and found 117 potential DC issues. We reported these issues to the concerned developers. So far, 63 issues have been confirmed, of which 38 have been quickly fixed using our suggested patches.

## 1 INTRODUCTION

Python projects are commonly shared as third-party libraries in a server-side central repository PyPI [35], and reused by other projects with a client-side library installer pip [31, 39, 47]. By June 2019, the PyPI ecosystem (PyPI for short) has indexed over 1.43 million Python libraries together with their metadata (e.g., version information, dependencies on other libraries, etc.).

To use a library on PyPI, developers need to specify the desired version constraints [43] in a configuration script such as `setup.py` and `requirements.txt` [37]. When a library is reused by another project, this library and other libraries on which it depends will be automatically installed at the project's build time. The automation smartly combines a server-side central repository and a client-side library installer to manage library dependencies. It considerably simplifies the build process of Python projects. Also, the version constraint mechanism for a required library allows developers to restrict the dependencies to a set of compatible versions and enables automatic library evolution [3]. However, such automation comes with the risk of potential dependency conflict (DC) issues, which can cause build failures when the installed version of a library violates certain version constraints on the library.

Figure 1 gives a real example: issue #1277 [6] in `channels`. As shown in `channels` 2.1.7's configuration script, it directly requires libraries `asgiref` (version constraint: ⟨≥ 2.3 ∧ < 3.0 ⟩) and `daphne` (version constraint: ⟨≥ 2.2 ∧ < 3.0 ⟩). Note that when downloading a library, the `pip` installer always chooses the latest version on PyPI that satisfies the library's version constraint [32]. No DC

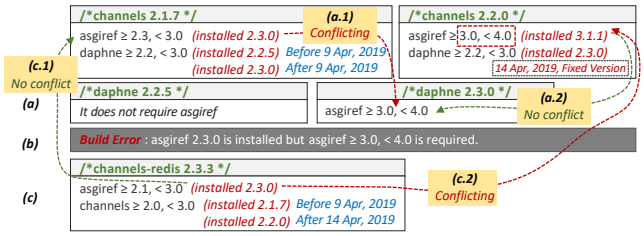



**Figure 1: Illustrative examples of issues #1277 [6] and #152 [7]**

issues occurred when `channels` 2.1.7 was built before 9 Apr 2019. Both `asgiref` 2.3.0 and `daphne` 2.2.5 selected for the build satisfy the concerned constraints. However, issue #1277 [6] arose after 9 Apr 2019 when `channels` 2.1.7 was built via selecting the newly released library `daphne` 2.3.0, which additionally requires library `asgiref` (version constraint: ⟨≥ 3.0 ∧ < 4.0⟩). The DC issue (the red curve a.1) happened because `pip` selected `asgiref` 2.3.0 to satisfy the direct dependency constraint ⟨≥ 2.3 ∧ < 3.0⟩, but this version violated the constraint ⟨≥ 3.0 ∧ < 4.0⟩ specified in `daphne` 2.3.0. This issue caused a build error as shown in Figure 1(b).

To fix the issue, `channels` developers released version 2.2.0 on 14 Apr 2019, which updated the requirement on `asgrief`'s versions to ⟨≥ 3.0 ∧ < 4.0⟩. This update led to the installation of `asgrief` 3.1.1 (the latest version under 4.0) when building `channels` 2.2.0, thus resolving the DC issue (the green curve a.2). However, this fix induced another DC issue in `channels-redis`, as Figure 1(c) shows. After `channels`'s upgrade, to build `channels-redis` 2.3.3, `pip` still selected `asgiref` 2.3.0 to satisfy the direct dependency constraint ⟨≥ 2.1 ∧ < 3.0⟩. Unfortunately, this version is in conflict with the constraint ⟨≥ 3.0 ∧ < 4.0⟩ transitively introduced by `channel` 2.2.0, leading to a build failure (red curve c.2).

To understand the scale of DC issues and their characteristics, we conducted an empirical study on 235 real DC issues with fixing solutions, reported on `Github` in the last five years, from 124 popular Python projects. We thoroughly studied these issues and explored the following two research questions.

- RQ1 (**Manifestation patterns**): How are DC issues manifested in Python projects? Are there common patterns that can be leveraged for automated diagnosis of these issues?
- RQ2 (**Fixing strategies**): How do developers fix DC issues in Python projects? Are there common practices that can be leveraged for automated repair of these issues?

Through investigating the above questions, we observe that DC issues mainly arise from conflicts caused by remote dependency updates or local environment (see Section 3.2). We also found common strategies to fix DC issues and key factors leading to potential DC issues and their regressions (see Section 3.3).

Diagnosing DC issues is challenging as echoed by developers in their comments on issue report #3118 [16] such as "*the dependency resolution in the Python world is far from being easy.*" The challenges are mainly attributed to the complex dependencies across projects,

which often specify the version constraints on their dependent libraries without considering those specified by other projects. To be specific, we summarize three major challenges as follows.

First, the version of a library installed for a Python project can vary over time. For each required library, `pip` will install its latest version satisfying the concerned constraint. Any updates of libraries on `PyPI` can affect the version of the libraries installed for the downstream projects (i.e., the projects that depend on the libraries), causing potential build failures. We observe that on average there were around 800 library updates on `PyPI` every day, and this number is increasing (see Section 2.3).

Second, when a library updates its version constraints on other libraries, its downstream projects could be affected. An impact could be wide-spreading since it can be propagated transitively to a wide range of downstream projects. Manually identifying the affected downstream projects is impractical for developers.

Third, it is difficult to obtain a full dependency graph with version constraints for projects on `PyPI`. The state-of-the-art tools like `pipenv` and `Poetry` show only which libraries have been installed, rather than their dependencies. Developers are looking for tools that generate dependency graphs when diagnosing DC issues. For instance, a developer left a comment "*A tool that can dig and build the full required (not installed) dependencies graph and report all the union of all requirements is suggested*" in a recent issue report [16].

To address the challenges and help Python developers combat DC issues, we designed a technique Watchman, which performs a holistic analysis from the perspective of the entire `PyPI` ecosystem, to continuously monitor dependency conflicts caused by library updates. For each library version on `PyPI`, Watchman builds a full dependency graph (FDG), a formal model that simulates the process of installing dependencies for library versions. The FDGs can be incrementally updated as the libraries evolve on `PyPI`. Watchman then analyzes them to detect and proactively prevent DC issues. Since FDGs record full dependencies with version constraints, they can also provide useful diagnostic information to help developers understand the root causes of detected DC issues to ease fixing.

To evaluate the effectiveness of Watchman, we played back the evolution history of all libraries on `PyPI`, from 1 Jan 2017 to 30 Jun 2019 and deployed Watchman to detect DC issues. After analyzing `PyPI` snapshots during this period, Watchman detected 515 DC issues and 502 (97.5%) of them were indeed fixed by developers during the evolution of the libraries. To evaluate the usefulness of Watchman, we ran it to monitor dependency conflicts for the `PyPI` ecosystem between 11 Jul 2019 and 16 Aug 2019. During the period, it detected and reported 117 previously-unknown DC issues, 63 of which (53.8%) have already been confirmed by developers. 38 (60.3%) confirmed issues were readily fixed following our suggested patches. Developers also expressed great interests in Watchman. In summary, our work makes three major contributions:

- **Originality:** To the best of our knowledge, we conducted the first empirical study on the DC issues in open-source Python projects. Our findings help understand the characteristics of DC issues and provide guidance to future studies related to this topic.
- **Dataset:** We publicize the empirical study dataset containing 235 DC issues collected from 124 real-world Python projects to facilitate future research.

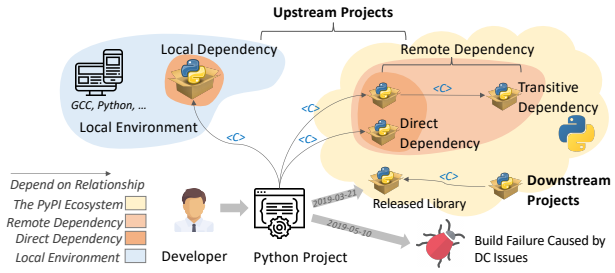



**Figure 2: Dependencies of a Python Project**

- **Technique:** We propose a formal model to simulate the building process of Python projects and developed a DC issue diagnostic technique Watchman (**http://www.watchman-pypi.com/**). Our evaluation shows that Watchman is scalable and useful. It can analyze over one million library releases on `PyPI` and detect DC issues with a high precision. It also helps understand the root causes of detected DC issues to ease fixing.

# 2 BACKGROUND

## 2.1 Dependencies of Python Projects

Figure 2 illustrates the dependencies of Python projects. Code reuse is pervasive in `PyPI`, where projects often reuse other libraries. The configuration script of a project $P$ explicitly constrains the versions of ***direct dependencies*** that $P$ may use. If these direct dependencies further rely on other libraries, such libraries are the ***transitive dependencies*** of $P$. In our paper, all direct and transitive dependencies are collectively called the ***upstream projects*** of $P$. We also call $P$ a ***downstream project*** of its dependencies.

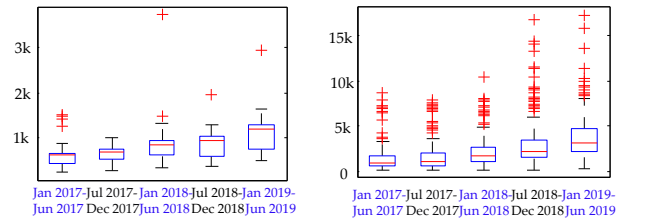
Python projects are mostly developed in a self-contained environment, which can be created by tools such as `virtualenv` [36], `conda` [2], and `pipenv` [33]. When building a Python project, the client-side library installer `pip` downloads most of the required libraries on `PyPI`. We refer to such libraries that need to be downloaded as ***remote dependencies***. For each required remote dependency, `pip` downloads the library from `PyPI` according to its name and version constraint. If `pip` finds multiple releases of a library on `PyPI` satisfying the version constraint, it downloads and installs the latest version of the library [31].

Besides remote dependencies, the development of a Python project can be affected by its ***local environment***, including the local development tool chains (e.g., the Python interpreter and GCC) and potential ***local dependencies***. Local dependencies are libraries installed in the local environment. They exist when the development environment is not clean (e.g., the project is not developed in an isolated virtual environment) and contains preinstalled libraries. If any version of a required dependency has been installed locally, `pip` will not download the dependency from `PyPI`.

`PyPI` allows Python developers to release their projects as libraries. However, when a project has a build failure, the failure can affect the build of all its downstream projects. As such, the consequences of project build failures in `PyPI` are serious. This paper aims to study the build failures caused by DC issues in `PyPI`.

## 2.2 Library Version Constraints

To use a library, a project needs to specify a constraint on desired library versions as shown in Figure 2 (i.e., the C annotations on




(a) The number of library updates each day (b) The number of the direct downstream projects that may be affected by the library updates each day

**Figure 3: Statistics of library evolution history**

some edges). To facilitate subsequent discussion, we formally define *version constraint* using the grammar below:

$$\begin{aligned} C &::= \epsilon \mid \text{range} \wedge \text{extra} \mid = version_{id} \\ \text{range} &::= \text{range} \wedge op\ version_{id} \mid op\ version_{id} \\ \text{extra} &::= \epsilon \mid \neq version_{id} \mid \text{extra} \wedge \neq version_{id} \\ op &::= > \mid \geq \mid < \mid \leq \end{aligned} \tag{1}$$

where $version_{id}$ refers to a specific version of a library (e.g., 1.24.1).

A constraint C could be empty, in which case `pip` will choose to download the latest version of the library from `PyPI` if there is no version installed locally. Developers may also specify a specific version that is desired (e.g., = 1.24.1) or undesired (e.g., ≠ 1.24.1). In practice, developers mostly specify a range of versions in a constraint (e.g., ≤ 1.24.1 ∧ > 1.11.0). Specifically, we investigated the top 1,000 popular Python projects on `PyPI` based on the number of their corresponding downstream projects, and found that 92.2% of their direct dependencies are set to a range of versions in the configuration scripts (in comparison, this ratio is only 0.03% for Java projects managed by `Maven` using the same investigating method). Such heavy usage of ranges in the version constraints for dependencies in the Python world, makes the diagnosis of DC issues complicated and challenging (see Section 3).

## 2.3 Frequent Updates of Python Libraries

Many libraries hosted on `PyPI` are updated frequently. These updates can have serious impacts because they might affect their downstream projects. To understand the scale and impacts of library updates on `PyPI`, we reviewed its library evolution history from Jan 2017 to Jul 2019. Through mining the release information updates daily, we obtained the distribution of the number of library updates in `PyPI` during this period as shown in Figure 3(a). On average, there were 801 updates of different libraries daily, and the number of updates is increasing over time. These library updates affected a large number of downstream projects directly. Figure 3(b) shows the distribution of the number of downstream projects that were directly affected. On average, 2,509 direct downstream projects are potentially affected by these updates. Note that such impacts can be propagated to other Python projects via transitive dependencies. Such frequent updates of libraries in `PyPI` induce imminent risks of DC issues.

# 3 EMPIRICAL STUDY

## 3.1 Data Collection

Following the data collection process adopted by existing studies [41, 50], we prepared our dataset in two steps.

**Step 1: selecting subjects.** To understand the issue manifestation patterns and fixing strategies, we need to study the issue

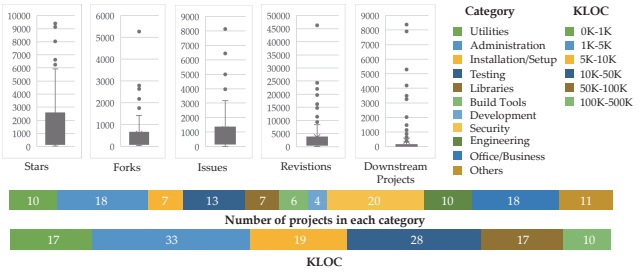



**Figure 4: Statistics of the projects used in our empirical study**

reports (with discussions if any), dependency configuration scripts, issue-fixing patches, and related code revisions. To achieve this goal, we searched `Github` for Python projects that satisfy three conditions: (1) popular: having more than 50 stars or forks, (2) being used as libraries: containing more than three direct downstream projects, and (3) well-maintained: having over 500 code revisions or over 50 issue reports. As such, we obtained 1,596 open-source Python projects.

**Step 2: identifying DC issues.** To locate DC issues in the 1,596 subjects, we searched for the issue reports that contain keywords "dependency conflict" or "dependency hell" (case insensitive), filed between Jul 2014 and Jul 2019 (i.e., in the past five years). 2,593 issue reports were returned by searching "dependency conflict", and that number is 334 by searching "dependency hell". Be noted that the returned search results may overlap. We further removed noises from the returned issue reports in two steps:

- We excluded the reports that are not concerning valid DC issues. For instance, issue #3900 of project `conan` dealt with the message conflict warnings, whose comments accidentally contained our searching keywords.
- For the remaining issue reports, we examined their issue descriptions and fixing solutions. An issue report was included to our dataset if it contains description of root causes, and either: (1) there is a corresponding revision if its status is "closed" or (2) there is an explicit consensus on fixing solutions or patches found in the project's code repository if its status is "open".

Three authors were involved in the process of data collection, analysis, and cross-checking. Eventually, we obtained 235 DC issues from 124 projects, 201 of which have been fixed. Figure 4 shows the statistics of the 124 subjects. They are: (1) large in size (around 38 KLOC on average), (2) well-maintained (containing 78 revisions and 92 issues on average), (3) popular (83% of them have over 100 stars), (4) impactful (86% of them have more than 5 direct downstream projects), and (5) diverse (covering over 10 categories). In the following, we study the 235 collected issues to answer RQ1–2.

## 3.2 RQ1: Manifestation Patterns

DC issues manifest themselves on `PyPI` due to different causes. RQ1 aims at characterizing these manifestation patterns. For this purpose, we performed an in-depth analysis of the 235 collected DC issues. Based on the analysis, we divided these issues into two patterns A and B, according to whether the issues are caused by *remote dependencies* or *local environment*. For each pattern, the issues can be further categorized based on how the dependency conflicts arise. In the following, we discuss these manifestation patterns in detail with illustrative examples.

#### 3.2.1 *Pattern A: conflicts caused by remote dependency updates.*

**Finding 1:** *211 out of the 235 (89.8%) DC issues that involve the violation of library version constraints are introduced by the updates of remote dependencies on PyPI.*

The root cause of the 211 issues is that the updates of some remote dependency might change the version of the concerned library to be installed by `pip`, which is hardly perceptible to project developers. Suppose that a Python project $P$ requires a library $\beta$ with a constraint $C$. If $C$ does not specify an upper bound on $\beta$'s version (e.g., $C = \langle \geq 3.0 \rangle$), or the specified upper bound is greater than the latest version of $\beta$ on `PyPI` (e.g., $C = \langle 2.0 \leq \wedge \leq 4.0 \rangle$, while the latest version of $\beta$ is 3.0), the version of $\beta$ used to build $P$ can be uncontrollable (in developers' perspective), meaning that $\beta$ can be upgraded when there are new versions on `PyPI`. Such upgrading can easily induce DC issues: $\beta$'s new version may not satisfy the constraints specified by other dependencies of $P$; the version constraints specified by $\beta$ for its own libraries may also change in new versions, causing potential conflicts with the constraints for the same libraries introduced by $P$'s other dependencies.

The 211 issues can be further categorized based on where the dependency conflicts come from. Theoretically, conflicts could happen in three different cases: (1) among direct dependencies themselves, (2) between direct dependencies and transitive dependencies, and (3) among transitive dependencies themselves. However, developers usually will not introduce conflicts among direct dependencies by themselves (i.e., mistakenly specifying two conflicting libraries in the configuration script). Indeed, we did not observe any such conflicts. In the following, we discuss the latter two cases.

- *a. Conflicts between direct and transitive dependencies (139/211).* Suppose that a Python project $P$ directly depends on two libraries $\alpha$ with $C_{P\to\alpha}$ and $\beta$ with $C_{P\to\beta}$, and $\beta$ further depends on $\alpha$ with $C_{\beta\to\alpha}$, where $C_{P\to\alpha}$, $C_{P\to\beta}$ and $C_{\beta\to\alpha}$ are version constraints of the corresponding libraries. In other words, $\alpha$ is not only a direct dependency of $P$, but also required by other direct dependencies of $P$ (i.e., $\alpha$ can also be seen as a transitive dependency of $P$). When building $P$, `pip` will always install the latest version $v$ of library $\alpha$ that satisfies $C_{P\to\alpha}$, as $\alpha$ is at the top level of $P$'s dependency tree [31]. If $v$ falls into the version range specified by $C_{\beta\to\alpha}$, the project $P$ will be built successfully. However, once $\alpha$ has been updated on `PyPI`, the update may lead to the installation of another version $v'$ of $\alpha$. If $v'$ falls out of the range specified by $C_{\beta\to\alpha}$, the project $P$ will not be built successfully. For instance, in issue #229, `gallery-dl` directly requires libraries `requests` $\langle \geq 2.11.0 \rangle$ and `urllib3` $\langle \geq 1.16 \wedge \neq 1.24.1 \rangle$, and the installed version 2.13.0 of `requests` depends on `urllib3` $\langle < 1.25.0 \wedge \geq 1.21.1 \rangle$. The project `gallery-dl` worked well when it was released on `PyPI`, as the latest version of `urllib3` at that time was 1.24.2. This version of `urllib3`, which would be installed by `pip`, satisfies both $\langle \geq 1.16 \wedge \neq 1.24.1 \rangle$ and $\langle < 1.25.0 \wedge \geq 1.21.1 \rangle$. However, when `urllib3` was updated to 1.25.0 on 18th Apr, 2019, `gallery-dl` began to suffer from build failures caused by DC issues. This is because when building `gallery-dl`, `pip` will install the latest version, which is 1.25.0 to satisfy the direct dependency `urllib3` (for simplicity, let us suppose that there is no preinstalled `urllib3`). However, this violates the constraint $\langle < 1.25.0 \wedge \geq 1.21.1 \rangle$ specified in `requests`, thus leading to a DC issue.
- *b. Conflicts between transitive dependencies (72 out of 211 issues).* Suppose that a Python project $P$ directly depends on two libraries $\alpha$ and $\beta$, both of which depend on another library $\theta$ but with two different version constraints $C_{\alpha\to\theta}$ and $C_{\beta\to\theta}$, respectively. If the version $v$ downloaded by `pip` referring to $C_{\alpha\to\theta}$ (suppose it has a higher priority) also satisfies $C_{\beta\to\theta}$, the project $P$ can be built successfully. However, since $\alpha$ and $\beta$ are two separate projects, their dependency relationship on $\theta$ may change over time. There can be cases where the updates of $\alpha$ or $\beta$ would result in conflicting version constraints of $\theta$, consequently causing DC issues when building $P$. We observed 72 such issues in our study. For example, issue #3826 of `rasa` complained the incident that a project was forced to introduce multiple version constraints of the library `request` by its direct dependencies `rasa` and `sagemaker`. The reason is that `rasa` released a new version 1.0.4 and added a constraint $\langle$=2.22.0$\rangle$ on `request`. However, this constraint is conflicting with another constraint on `request` $\langle \geq 2.20.0 \wedge < 2.21.0 \rangle$ introduced by `sagemaker`.

#### 3.2.2 *Pattern B: conflicts affected by local environment.*

**Finding 2:** *24 out of the 235 (10.2%) DC issues arose due to the conflicts between remote dependencies and the tools/libraries installed in the local environment.*

Such issues can happen when the required development tool of a remote dependency is incompatible with the local installed one (e.g., requires Python 3.7.* but installed Python 3.6.*). They can also happen when the version of a dependency, which is already installed in the local environment, does not satisfy the constraint specified by a remote dependency. Take issue #25316 as an example. Project `gradient` failed to be built because there was already one version (1.13.3) of the library `numpy` installed in the locally before the build, and this version is in conflict with the constraint $\langle \geq 1.15 \rangle$ specified by `pandas` v0.24.1, a direct dependency of `gradient`.

#### 3.2.3 *Dependency Smells.*

By further analyzing developers' discussions in the issue reports and the dependency configuration scripts of the project versions that were benign (i.e., not affected by the reported DC issues of **Pattern A**). We observed several types of "dependency smells". These smells are interesting in that they do not immediately cause DC issues but are likely to induce DC issues as the projects evolve.

**Finding 3:** *Restricting dependencies to specific versions for common libraries could easily induce DC issues to downstream projects.*

Build failures can easily happen if library version constraints are too restrictive (e.g., only accepting specific versions), especially for those common libraries. 59 of our studied 235 issues belong to this case. For instance, the project `docker-py` depends on a specific version of `request` 2.2.1, a common library that is used by many other projects. This makes `docker-py`'s downstream projects that also depend on `request` particularly sensitive to the updates of `request`. We observe that whenever there was a new version of `request` released on `PyPI`, `docker-py`'s developers would receive requests from downstream projects to upgrade its version constraint on `request` (e.g., issues #3404 [18], #4431 [22]). As there were too many such requests, `docker-py` developers finally chose to loosen the version constraint on `request` to a range, thus allowing more downstream projects to work well with it.

**Finding 4:** *DC issues can easily occur when the installed version of a library satisfying one constraint is close to the upper bound specified in another version constraint.*

67 of the 235 issues belong to this category. A library version installed by `pip` in the concerned project is close to the upper bound of the its another version constraint. Therefore, updates of these libraries will likely induce build failures. For instance, the projects that directly require both `request` and `urllib3` have often encountered DC issues (e.g., [19, 23–25]). The reason is that these projects will always install the latest version of `urllib3` since the direct dependency constraints on `urllib3` do not set an upper bound. Besides, `request` also depends on `urllib3` with a version constraint $\langle \geq 1.21.1 \wedge < 1.23 \rangle$. These projects were built successfully when `urllib3`'s latest version was 1.22.4, which satisfies $\langle \geq 1.21.1 \wedge < 1.23 \rangle$. However, the installed latest version was close to the upper bound, and thus DC issues will arise once `urllib3` updated a newer version (e.g., 1.23.1) that exceeds the upper bound 1.23.0 set by `request`.

These findings are useful. We will show that identifying the two types of smells can help perform predictive analysis to proactively prevent DC issues before they cause real build failures.

## 3.3 RQ2: Fixing Strategies

To answer RQ2, we further studied: (1) the patches of the 201 closed issue reports, (2) solution descriptions of the 34 open issue reports, and (3) the comments in issue reports. We observed seven fixing strategies, which altogether resolved 93.6% of our collected issues.

### 3.3.1 *Common fixing strategies with illustrative examples.*

**Strategy 1:** *Adjusting the version constraints of direct dependencies (98/235).* The conflicts between direct and transitive dependencies were commonly fixed by adjusting the version constraint of direct dependencies to be compatible with those of transitive dependencies. For example, in issue #32 [17], project `valinor` contained two conflicting version constraints on the library `pyyaml`. One constraint $\langle \geq 3 \wedge < 5 \rangle$ was directly specified by `valinor`. The other constraint $\langle < 6.0 \wedge \geq 5.1 \rangle$ was transitively introduced by `pyOCD`, a dependency of `valinor`. For such case, `pip` will always install the latest version satisfying the direct dependency's constraint (i.e., $\langle \geq 3 \wedge < 5 \rangle$), thus conflicting with the transitive dependency. To fix the problem, developers of `valinor` revised their version constraint on `pyyaml` to $\langle < 6.0 \wedge \geq 5.1 \rangle$.

**Strategy 2:** *Upgrading or downgrading the direct dependencies that require conflicting libraries (27/235).* DC issues caused between transitive dependencies can be solved by upgrading or downgrading the direct dependencies that introduce the transitive dependency. Take issue #66 [26] of `zhmcclient` as an example. The two conflicting version constraints, i.e., $\langle = 4.0.3 \rangle$ and $\langle \geq 4.4 \rangle$, on `coverage` were transitively introduced by `zhmcclient`'s direct dependencies `python-coveralls` $\langle = 2.9.1 \rangle$ and `pytest-cov` $\langle \geq 2.4.0 \rangle$, respectively. Since the installed version `pytest-cov` 2.6.0 added `coverage` $\langle \geq 4.4 \rangle$ as its direct dependency, which caused the conflict, `zhmcclient`'s developers downgraded `pytest-cov` by changing its version constraint to $\langle \geq 2.4.0 \wedge < 2.6.0 \rangle$. After revising the constraint, `pytest-cov` 2.5.1, an older version of `pytest-cov` that requires `coverage` $\langle \geq 3.71 \rangle$, was installed. This constraint is not conflicting with $\langle = 4.0.3 \rangle$ and thus the DC issue was resolved.

**Table 1: The relations between manifestation and fixing strategies**

| | Strategy 1 | Strategy 2 | Strategy 3 | Strategy 4 | Strategy 5 | Strategy 6 | Strategy 7 |
|---|---|---|---|---|---|---|---|
| Pattern A.a | 94 | 9 | 19 | 8 | | | |
| Pattern A.b | | 18 | 32 | | 16 | | |
| Pattern B | 4 | | | | | 12 | 8 |

**Strategy 3:** *Coordinating with upstream projects to adjust conflicting version constraints (51/235).* DC issues can also be fixed via coordinating with the upstream projects. Take issue #740 [28] of the project `yotta` as an example. Although the conflict can be resolved by adjusting the direct dependency's version constraint (i.e., following **Strategy 1**), the developers chose to coordinate with the upstream projects to solve the problem. This avoids changing the version of the directly required library.

**Strategy 4:** *Removing conflicting direct dependencies and keeping the transitive ones (8/235).* When it is difficult to make a project's direct dependencies in line with the transitive ones, developers may choose to remove the conflicting direct dependencies. For example, as described in issue #407 [20] of the project `wandb`, conflicts occurred when its upstream project updated its version constraint on a direct dependency `PyYAML`, and this happened several times. As the developers had no direct control on the upstream projects, they removed the conflicting direct dependency from the configuration script, and used the transitively introduced one instead.

**Strategy 5:** *Adding direct dependencies (16/235).* There are cases when the version constraints $C_{\alpha \to \theta}$ and $C_{\beta \to \theta}$ of two conflicting transitive dependencies overlap, meaning that one can find some versions of the concerned library $\theta$ to satisfy both constraints. The DC issue in such a case can be resolved by adding $\theta$ as a direct dependency with a constraint that entails both $C_{\alpha \to \theta}$ and $C_{\beta \to \theta}$. This will instruct `pip` to install the version specified by the direct dependency that satisfies both the transitive dependencies. For instance, in issue #1586 [8] of `crossbar`, there exist two conflicting transitive dependencies: `urllib3` $\langle < 1.25 \wedge \geq 1.21.1 \rangle$ and `urllib3` $\langle \geq 1.24.2 \rangle$. To resolve the conflict, developers added `urllib3` $\langle \geq 1.24.2 \wedge < 1.25 \rangle$ as a direct dependency to override the two conflicting ones to avoid build failures.

**Strategy 6:** *Upgrading/downgrading the development tool (12/235).* The dependency conflicts between the local environment and the remote dependencies are often solved by upgrading or downgrading the development tools (e.g., issue #409 [21] of `bandit`).

**Strategy 7:** *Creating an isolated environment (8/235).* This is considered as a viable solution for the dependency conflicts between remote and local installed dependencies. As recommended by developers in issue #9090 of `spyder` and issue #25487 [13] of `pandas`, there are several tools such as `virtualenv` [36], `conda` [2], and `pipenv` [33], which can create virtual environments to isolate the impacts of local installed dependencies to avoid such DC issues.

There are nine issues that were fixed by restricting the conflicting library to a specific version. However, this is not a good practice for managing dependencies and can induce new issues (e.g., issues #2483 [12], issues #1824 [9], and #2195 [11], etc.) as discussed in **Finding 3**. Therefore, we do not present it as a fixing strategy. The remaining six issues were fixed by specific workarounds. We do not further discuss the details in this paper.

Table 1 summarizes how each pattern of issues were fixed. We can make the following observations based on the table. There can be multiple fixes feasible for a DC issue. In particular, **Pattern A.a** issues can be fixed by adopting four different strategies,

among which **Strategy 1** is the most adopted one. This is because the project developers have full control of the direct version constraints. If adopting such strategy will cause side effects such as security loopholes, developers may solve the conflicts by upgrading or downgrading the direct dependencies of their projects (**Strategy 2**) or coordinating with upstream projects to adjust conflicting version constraints (**Strategy 3**). For Pattern A.b, developers often adopt **Strategies 2**, **3** and **5** to resolve adjust the version constraints of the conflicting libraries. Issues of **Pattern B** are mainly resolved via dealing with the local environments. Based on these results, we distill the following findings:

**Finding 5:** *There can be multiple fixes for a DC issue. The solutions can be affected by the issue's manifestation pattern, the topological structure of the project's dependency graph, `pip`'s installation rules, and the interference between the version constraints of upstream projects and those of downstream projects.*

# 4 DEPENDENCY CONFLICT DIAGNOSIS

## 4.1 Overview

Our empirical findings suggest the limitation of diagnosing DC issues based on a single project is due to complex dependencies among upstream and downstream projects. This motivates us to propose a technique, Watchman, to continuously monitor dependency conflicts from the perspective of the entire ecosystem.

Figure 5 gives an overview of our technique. A major challenge is to perform a holistic analysis of the huge number of projects on PyPI, the dependency relationships among which cannot be easily modeled and are subject to change. To address the challenge, Watchman first collects the metadata for each library version, including its direct dependencies with version constraints and their declaration order. Second, it constructs a metadata repository for all the libraries hosted on PyPI to enable the analysis of the interference between the version constraints across upstream and downstream projects. Then, by continuously monitoring library release information on PyPI, Watchman synchronously updates the metadata repository to precisely model the dependency relationships. For the captured library updates, Watchman uses a depth-first searching strategy to identify the affected downstream projects. It also performs a breadth-first search on the metadata repository to construct a full dependency graph for each potentially affected downstream project, according to library installation rules of `pip`. Finally, it performs automatic DC issue diagnosis. The following subsections introduce the details of Watchman.

## 4.2 Constructing Metadata Repository

To model the dependency relationships among libraries, Watchman uses the *metadata structure* defined below to capture the version constraints of the direct dependencies of each library version and the declaration order of these direct dependencies. For ease of understanding, in the subsequent discussions, we shall use lowercase Greek letters to denote libraries and superscripts to denote versions.

**Definition 1. Metadata Structure:** For a library version $\zeta^v$, i.e., the version $v$ of library $\zeta$, Watchman captures a collection of information $G(\zeta^v) = (D, R, P)$ , where

- $D = \{\alpha, \beta, \gamma \cdots\}$ is a set of direct dependencies of $\zeta^v$.
- $R = \{C_{\zeta^v \to \delta} | \ \delta \in D\}$, where $C_{\zeta^v \to \delta}$ denotes the version constraint on the dependency $\delta$ specified by $\zeta^v$.
- $P$ is a function that maps each dependency $\delta \in D$ to its declaration order.

**Algorithm 1:** Identifying Affected Downstream Projects

```
Input: L_up and G
Output: L_af
1 L_af ← {};
2 foreach ζ^v ∈ L_up do
3     identifyAffectedLibrary(ζ^v, L_af, G);
4 Function identifyAffectedLibrary(ζ^v, L_af, G)
5 foreach G(δ^u) = (D, R, P) ∈ G do
6     if ζ ∈ D && v satisfies the constraint C_{δ^u→ζ} then
7         L_af ← L_af ∪ {δ^u};
8         identifyAffectedLibrary(δ^u, L_af, G);
```

In our experiment to detect unknown DC issues, Watchman first extracted 1,423,291 versions of 191,787 distinct libraries from a snapshot of PyPI on 15th Jun, 2019. For each library version $\zeta^v \in L$, where $L$ represents all library versions, it obtained the structured metadata $G(\zeta^v)$ via analyzing the dependency configuration script of $\zeta^v$. Such metadata of all extracted library versions then formed an initial *metadata repository* $\mathcal{G}$, which can be defined as $\mathcal{G} = \{G(\zeta^v) | \zeta^v \in L\}$. This process took around 8.5 days. The constructed metadata repository enables the queries of dependency relationships among upstream and downstream projects on PyPI.

## 4.3 Analyzing the Impacts of Library Updates

Analyzing the impacts of library updates mainly involves two steps:

**Step 1: Monitoring library updates.** Library updates on PyPI often cause DC issues (cf. **Finding 1** in Section 3.2). There are two types of library updates on PyPI: *new versions of an existing library being released* and *new libraries being released*. As discussed in Section 2.3, the library updates on PyPI each day may potentially affect thousands of their downstream projects. To reduce such adverse effects, Watchman computes $L_{up}$ by monitoring the two types of library updates on a daily basis. For each library version $\zeta^v \in L_{up}$, it collects the metadata $G(\zeta^v)$ and adds it to the metadata repository $\mathcal{G}$. In this manner, the metadata repository $\mathcal{G}$ can be updated synchronously with the evolution of the libraries on PyPI. On average, the repository update takes about 2.5 hours each day.

**Step 2: Identifying affected downstream projects.** Watchman performs backward search for identifying the set of library versions of downstream projects affected by $L_{up}$, denoted $L_{af}$, following the process as described in Algorithm 1. The algorithm works as follows. First, it initializes $L_{af}$ to an empty set (Line 1). For each library $\zeta^v \in L_{up}$, Watchman analyzes which library in the ecosystem may be directly affected by the update, with the aid of function IdentifyAffectedLibrary (Lines 2–4), which takes $\zeta^v$, $L_{af}$, and $\mathcal{G}$ as input and updates $L_{af}$ when needed. For each piece of metadata $G(\delta^u) = (D, R, P)$ in $\mathcal{G}$, if $\zeta$ is directly referenced by $\delta^u$ (i.e., $\zeta \in D$) and the version number $v$ satisfies the version constraint on $\zeta$ set by $\delta^u$ (i.e., $v$ satisfies $C_{\delta^u \to \zeta}$), then $\delta^u$ is considered to be possibly affected by $\zeta^v$ and added to $L_{af}$. Then, Watchman performs a depth-first search to recursively find more downstream projects affected by $\delta^u$ and updates $L_{af}$ accordingly (Lines 4-8).

The time cost of this step may vary. During our study period, we observed the largest number (17,597) of direct downstream projects

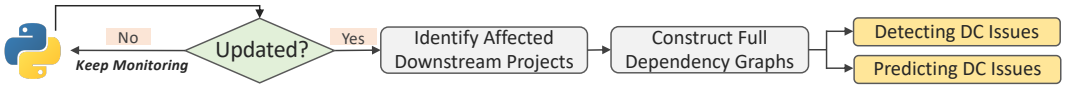



**Figure 5: The overall architecture of WATCHMAN**

affected by library updates on 20 Apr 2019. On that day, WATCHMAN was able to finish the search process within 38 minutes.

## 4.4 Detecting DC Issues

As discussed in Section 3.2, the topological structure of a Python project's dependency tree determines the installed library versions. In order to diagnose DC issues, for each potentially affected library version $\zeta^v \in L_{af}$, we need to analyze the relationships among all library versions that would be installed by `pip` to build $\zeta^v$. To capture such relationships, we propose a formal model named *full dependency graph*, which is defined below.

**Definition 2. (Full Dependency Graph).** The full dependency graph of a library version $\zeta^v$, denoted $FDG(\zeta^v)$, is a three-tuple, $(N, E, FR)$, where

- $N = N' \cup \{\zeta^v\}$ is the set of nodes in the graph, and $N'$ denotes a set of library versions that would be installed by `pip` to build $\zeta^v$. The libraries here include both direct and transitive dependencies.
- $E = \{\langle \alpha^x, \beta^y \rangle | \alpha^x, \beta^y \in N\}$ is a set of directed edges, where the edge from $\alpha^x$ to $\beta^y$ represents that the version $x$ of library $\alpha$ directly depends on library $\beta$.
- $FR$ is a function. It maps each edges $e = \langle \alpha^x, \beta^y \rangle \in E$ to the version constraint that the library version $\alpha^x$ sets on the library $\beta$, i.e., $C_{\alpha^x \to \beta}$.

Note that the full dependency graph (FDG for short) of a library version is subject to change when its upstream projects are updated on `PyPI`. Algorithm 2 describes the process of constructing the full dependency graph of the library version $\zeta^v$. WATCHMAN constructs $FDG(\zeta^v)$ following `pip`'s breadth-first installation strategy: `pip` first installs direct dependencies for a project, and then install dependencies at the next level according to the project's dependency tree and the process continues until all dependencies are installed. In the algorithm, we use a queue named *Queue* to record the order of traversing and installing dependencies, and $\zeta^v$ is initially added to the queue. When visiting each dependency $\alpha^x$ in *Queue*, WATCHMAN first retrieves its metadata $G(\alpha^x) \equiv (D, R, P)$. WATCHMAN then tries to add each dependency $\beta$ in $D$ to the full dependency graph. If $\beta$ has not been loaded (or installed), WATCHMAN first determines the version to be loaded based on constraint $C_{\alpha^x \to \beta}$ (recorded in $R$) following `pip`'s installation rules (Line 8). $N$ and *Queue* are then updated accordingly (Line 9). If $\beta$ has already been added to the FDG, WATCHMAN will retrieve the loaded version (Line 11). A new edge $\langle \alpha^x, \beta^y \rangle$ is then added to $E$. The algorithm uses another queue *VisitedEdges* to record the order in which the edges are traversed (Line 13). WATCHMAN also sets the version constraint of this edge (Line 14), which can be retrieved from $R$. After traversing all the dependencies in *Queue*, the full dependency graph of a library is completely constructed.

**DC Issue Detection.** WATCHMAN detects DC issues by analyzing the full dependency graph $FDG(\zeta^v)$ for each project $\zeta^v \in L_{af}$ in the following steps. First, WATCHMAN traverses $FDG(\zeta^v)$ and

**Algorithm 2:** Constructing FDG via Breath-First Search

**Input:** $\zeta^v$ and $\mathcal{G}$
**Output:** $FDG(l^v) = (N, E, FR)$
1 $N \leftarrow \{\zeta^v\}$; $E \leftarrow \{\}$; $FR \leftarrow \{\}$;
2 $Queue.add(\zeta^v)$; $Loaded \leftarrow \{\zeta\}$; $VisitedEdges \leftarrow \{\}$;
3 **while** $!Queue.isEmpty()$ **do**
4 $\quad \alpha^x \leftarrow Queue.pop()$; $Loaded \leftarrow Loaded \cup \{\alpha\}$ ;
5 $\quad G(\alpha^x) \equiv (D, R, P) \leftarrow$ getMetadata$(\alpha^x, \mathcal{G})$ ;
6 $\quad$ **foreach** $\beta \in D$ **do**
7 $\quad\quad$ **if** $\beta \notin Loaded$ **then**
8 $\quad\quad\quad \beta^y \leftarrow$ getToLoadVersion$(\beta, C_{\alpha^x \to \beta})$ ;
9 $\quad\quad\quad N \leftarrow N \cup \{\beta^y\}$; $Queue.add(\beta^y)$;
10 $\quad\quad$ **else**
11 $\quad\quad\quad \beta^y \leftarrow$ getLoadedVersion$(\beta, N)$ ;
12 $\quad\quad E \leftarrow E \cup \{\langle \alpha^x, \beta^y \rangle\}$;
13 $\quad\quad VisitedEdges.add(\langle \alpha^x, \beta^y \rangle)$;
14 $\quad\quad FR(\langle \alpha^x, \beta^y \rangle) \leftarrow C_{\alpha^x \to \beta}$;

locates those nodes with multiple incoming edges. A node has multiple incoming edges when there are multiple projects that directly depend on the library represented by the node. Next, for each such node $\alpha^x$, WATCHMAN analyzes the set of its incoming edges, denoted $E_\alpha$. Note that there is one edge $e$ in $E_\alpha$ that is traversed first when constructing $FDG(\zeta^v)$ and $x$ is the latest version number of the library $\alpha$ that satisfies the constraint $FR(e)$. To detect DC issues, WATCHMAN checks $x$ against the set of constraints associated with other edges, i.e., $\{FR(e') \mid e' \in E_\alpha \backslash \{e\}\}$. If $x$ violates any such constraints, WATCHMAN will report a DC issue to the project $\zeta$.

## 4.5 Predictive Analysis for DC Issues

The constructed FDGs by WATCHMAN can further enable us to perform predictive analysis for proactive prevention of DC issues via detecting the two types of smells discussed in **Findings 3–4**.

**Type 1.** *Restricting a dependency to a specific version.* If a project restricts a dependency to a specific version, its downstream projects may suffer from DC issues. Specifically, DC issues may arise if the following conditions hold:

(1) There is a version $v$ of project $\zeta$, denoted $\zeta^v$, that restricts its direct dependency $\alpha$ to a specific version $x$.

(2) There is a version $y$ of a downstream project $\beta$ that depends on both $\zeta$ and $\alpha$, and $\zeta^v$ and $\alpha^x$ are the installed library versions for $\beta^y$ at the time of analysis.

Let $DP$ be the set of downstream projects (e.g., $\beta$) such found. The larger $|DP|$ is, the more likely that DC issues can arise. This is because each project in $DP$ independently sets its own version constraints on $\alpha$. If $\zeta^v$ only accepts the version $x$ of $\alpha$, the possibility that the constraint $\langle = x \rangle$ conflicts with other constraints on $\alpha$ set by the projects in $DP$ is high, especially when $DP$ is a large set. In our experiments, we will warn the project $\zeta$ developers, if $|DP|$ is larger than a threshold value, which is set empirically.

Figure 6(a) gives an illustrative example. In project $\texttt{C}^{2.0}$, the constraint for library `A` is restricted to $\langle = 2.0 \rangle$. In addition, `C`'s downstream project $\texttt{B}^{5.0}$ depends on both $\texttt{C}^{2.0}$ and $\texttt{A}^{2.0}$. In such a case, it is very likely that the restrictive constraint `C` sets on `A` would cause conflicts for `B` (e.g., when `A` gets updated on `PyPI`).

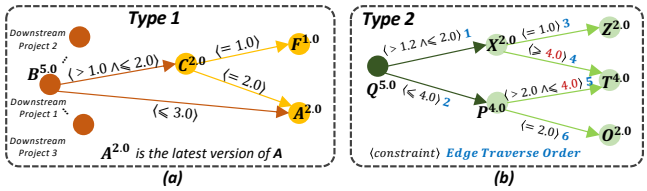



**Figure 6: Illustrative examples of potential DC issues**

The risk of conflicts gets higher if we find more such downstream projects. Watchman will find such cases and suggest project C's developers to relax its constraint on A, to avoid potential DC issues.

**Type 2.** *The installed version of a library is close to the upper bound specified in the version constraint.* If the installed version of a library satisfies the concerned version constraint but is close to the upper bound specified in the constraint, build failures can easily occur when the library evolves. Watchman considers the project $\zeta^v$ has a potential DC issue, if the following conditions hold:

(1) In $FDG(\zeta^v) = (N, E, FR)$, there exists a node $\alpha^u$ with multiple incoming edges, where $\alpha$ is a dependency of $\zeta^v$ and $u$ is the installed version of $\alpha$. Let $E_\alpha$ be the set of incoming edges to $\alpha^u$.

(2) The version constraint of the first traversed edge $e$ in $E_\alpha$ does not specify an upper bound on $\alpha$ (e.g., $\langle \geq y \rangle$) or the specified upper bound is greater than $\alpha$' latest version $z$ on PyPI. In this case, any updates of $\alpha$ on PyPI will affect the version of $\alpha$ to be installed.

(3) There exists another edge $e'$ in $E_\alpha$ ($e' \neq e$), of which the associated constraint $FR(e')$ specifies an upper bound on the version of $\alpha$ (e.g., $\langle \leq x \rangle$) and the upper bound is greater than or equal to the latest version of $\alpha$, i.e., $z$.

Figure 6(b) gives an illustrative example. In the FDG of project $Q^{5.0}$, there are two incoming edges to project $T^{4.0}$, one from project $X^{2.0}$ and the other from project $P^{4.0}$. Suppose that the former edge is traversed before the latter. Since the constraint that $X^{2.0}$ sets on T has no upper bound, the latest version 4.0 of T will be installed. There is no dependency conflict at the time of analysis. However, since the constraint associated with latter edge, i.e., $\langle > 2.0 \wedge \leq 4.0 \rangle$, restricts T to a version range, build failures may occur if developers release a newer version (e.g., 4.1) of T on PyPI.

## 5 EVALUATION

In this section, we study the following two research questions:

- RQ3 (**Effectiveness**): How effective is Watchman in detecting real DC issues and predicting potential ones?
- RQ4 (**Usefulness**): Can Watchman monitor DC issues in PyPI and provide useful diagnostic information?

For RQ3, we replayed the evolution history of all libraries on PyPI from 1 Jan 2017 to 30 Jun 2019. We first constructed a metadata repository for PyPI's snapshot on 1 Jan, 2017, and then conducted incremental analysis to extract daily updates of all libraries until 30 Jun 2019. For each library update, we applied Watchman to detect DC issues and predict potential ones via identifying dependency smells. Since we have the whole evolution history, we could evaluate Watchman's effectiveness by checking whether the detected DC issues have been resolved and whether the predicted ones have indeed evolved into real issues subsequently.

For RQ4, we deployed Watchman to monitor PyPI since 1 Jul 2019, and configured it to detect new DC issues of **Patterns A.a** and **A.b**, as well as potential ones that could be induced by smells of **Type 1** and **Type 2**. Note that issues of **Pattern B** can hardly be detected since they are affected by developers' local environments, on which we have no knowledge.

**Table 2: Basic information of experimental subjects**

| | Period 1 | Period 2 | Period 3 | Period 4 | Period 5 |
|---|---|---|---|---|---|
| Project# | 1,454 | 1,535 | 2,279 | 2,398 | 2,673 |
| Release# | 11,759 | 13,202 | 18,418 | 18,984 | 19,746 |
| Commits# | 530 | 646 | 338 | 740 | 694 |

We then consolidated the detected DC issues and filed reports to the concerned projects' issue tracking systems if (1) the detected issues have not been reported or fixed in the unreleased master branches of the concerned projects and (2) the concerned projects have maintenance records in the last two years (still active). The two criteria allow us to obtain developers' feedback (e.g., whether they will confirm our reported issues or merge our pull requests) to evaluate the usefulness of Watchman. In each issue report, we will point out the conflicts and explain how they arise. Such diagnostic information can be easily provided by Watchman since it simulates the build process of each analyzed project. The report also includes a list of fixing suggestions, which are generated by Watchman based on our summarized relationships between issue manifestation patterns and common fixing strategies.

### 5.1 RQ3: Effectiveness

**Data collection.** A project's evolution history provides useful information about how DC issues were manifested (and fixed). To ease experiments, we divided the whole period from 1 Jan, 2017 to 30 Jun, 2019 into five sub-periods, including: (1) *Period 1:* 1 Jan 2017 - 30 Jun 2017, (2) *Period 2:* 1 Jul 2017 - 31 Dec 2017, (3) *Period 3:* 1 Jan 2018 - 30 Jun 2018, (4) *Period 4:* 1 Jul 2018 - 31 Dec 2018, and (5) *Period 5:* 1 Jan 2019 - 30 Jun 2019. For each sub-period, we collected open-source Python projects satisfying the following two criteria as our experimental subjects: (1) having more than five release versions during this sub-period (active), and (2) having more than 300 commits during this sub-period (well-maintained). Table 2 lists the basic information of these subjects, which involves on average 16,421 releases of 2,067 projects for each sub-period.

**Metrics.** Watchman detects DC issues of **Patterns A.a** and **A.b**, and predicts potential ones that could be induced by smells of **Type 1** (Type 1 issues) and **Type 2** (Type 2 issues), during each sub-period on a daily basis. To evaluate Watchman's effectiveness, we define two metrics, ***resolving ratio*** and ***lasting time***, as follows:

- For each detected issue of **Patterns A.a** and **A.b**, we checked whether it had been resolved (fixed) in the latest version of the project released on PyPI, up to the validation date (20 Jul, 2019). The ***resolving ratio*** measures the proportion of resolved ones in Watchman's detected DC issues. Higher resolving ratios indicate better effectiveness of Watchman.
- The ***lasting time*** measures the gap between the detection time of a DC issue and the fixing time of this DC issue. A longer lasting time indicates wider side effects caused by a DC issue on the concerned downstream projects.

For the predicted issues, we checked whether they had turned into real ones. There are two cases: (1) the predicted issue indeed arose (reported) in history due to library updates, and was fixed by developers in subsequent releases; (2) the predicted issue was not reported in history but developers still had fixed it for avoiding certain undesirable consequences. Accordingly, the ***resolving ratio***

**Table 3: Results of DC issues reported by WATCHMAN from 1 Jan 2017 to 30 Jun 2019**

| | Period 1 | Period 2 | Period 3 | Period 4 | Period 5 | Summary |
|---|---|---|---|---|---|---|
| **Pattern A** | 56 | 42 | 84 | 72 | 115 | 369★ |
| Fixed | 56 | 42 | 84 | 72 | 115 | 369★ |
| ***resolving ratio*** | 100% | 100% | 100% | 100% | 100% | 100%♮ |
| ***lasting time*** | 25.2 | 27.3 | 25.0 | 20.8 | 31.6 | 26.0♮ |
| **Type 1** | 10 | 13 | 12 | 11 | 15 | 61★ |
| **Type 2** | 16 | 18 | 19 | 21 | 21 | 95★ |
| *Case (1)* | 2 | 2 | 3 | 4 | 2 | 13★ |
| *Case (2)* | 22 | 25 | 26 | 26 | 31 | 130★ |
| ***resolving ratio*** | 92.3% | 87.1% | 93.5% | 93.8% | 91.7% | 91.7%♮ |
| ***lasting time*** | 101.6 | 77.1 | 100.8 | 51.0 | 63.9 | 78.9♮ |

♮ denotes the average value while ★ denotes the sum.

measures the proportion of predicted DC issues that belong to either cases. The ***lasting time*** measures the gap between an issue was predicted and it was reported for those issues that belong to *Case (1)*, and the gap between an issue was predicted and it was resolved by developers for those issues that belong to *Case (2)*.

**Results.** Table 3 presents the statistics of WATCHMAN's detected DC issues for the five sub-periods, as well as *Resolving ratio* and *Lasting time* measurement results.

For all five sub-periods, Watchman detected a total of 369 DC issues of **Patterns A.a** and **A.b** (merged in table), and all of them had been fixed by developers (i.e., *resolving ratio* = 100%). This result strongly suggests that WATCHMAN can precisely detect DC issues. WATCHMAN also predicted a total of 156 **Type 1** and **Type 2** issues, and 143 of them had been resolved by developers, resulting in a satisfactory average *resolving ratio* of 91.7% (i.e., = (13 + 130) / (61 + 95)). The *resolving ratio* for different periods ranges from 87.1% to 93.8%, which are generally satisfactory. This result suggests that WATCHMAN is also effective in predicting potential DC issues. Besides, we observe that all detected 369 DC issues were resolved by developers within a month (on average, 26 days). WATCHMAN can potentially help reduce this delay since it can detect DC issues timely (it performs analysis on a daily basis) and report them to developers along with fixing suggestions. If developers fix the issues in due course, the side effect of these issues on downstream projects will be largely diminished.

As at 20 Jul 2019, 13 (8.7%) of the 156 DC issues predicted by WATCHMAN had not evolved into real ones. By further analyzing the concerned projects, we found that the dependencies introducing WATCHMAN's predicted issues were no longer active. For instance, in project `finance-dl` [5], there exist multiple version constraints for library `idna`. The version `idna` 2.8 installed is equal to the upper bound of the constraint $\langle \geq 2.5 \wedge \leq 2.8 \rangle$ introduced by the latest version of library `selenium-requests`. However, this potential DC issue (**Type 2**) did not evolve into a real one, since `selenium-requests` has stopped its update on `PyPI` (so far).

## 5.2 RQ4: Usefulness

WATCHMAN detected and predicted a total of 189 DC issues since we started our online-monitoring on 1 Jul 2019. We filtered out 23 issues that had been reported in the corresponding projects' issue tracking systems and 49 issues whose associated projects had no maintenance record in the last two years. After filtering, we reported the remaining 117 DC issues to developers. As shown in Table 4, 63 issues (53.8%) were confirmed by developers as real DC issues within a few days. 38 out of the 63 confirmed issues (60.3%) were quickly fixed, and 25 confirmed issues (38.7%) are in the process of being fixed. The remaining 54 issues are still pending, mainly due to the inactive maintenance of the associated projects. We provided a detailed analysis in the following.

### 5.2.1 *Feedback on reported issues.*

For 64 detected issues of **Pattern A** caused by library updates, WATCHMAN got a higher confirmation rate (60.9% = 39/64), which is within our expectation. For these 39 confirmed DC issues, developers agreed that the detected conflicts would lead to build failures, and invited us to submit patches to help resolve them. In particular, in issue #70 [27] of project `Osmedeus`, its developer indeed encountered our reported DC issue when deploying the project to a new environment, and left a comment "*I also get that error when installing the project but my server works fine. Just submit a PR and I will review the patch*" on our issue report.

For 21 predicted DC issues of **Type 1**, 11 of them have already been spotted by developers and resolved in the master branches of the projects (but not released on `PyPI`) before we reported them. For instance, project `MycroftAI adapt` voluntarily relaxed its version constraint on library `six` from $\langle = 1.10.0 \rangle$ to $\langle \geq 1.10.0 \rangle$ via commit `7eeadeb` [1] with a log "*to avoid incompatibility with downstream projects `adapt-parser` and `jsonschema`*". Therefore, we reported only the remaining 10 issues of **Type 1**, and 4 of them were confirmed by developers. Encouragingly, in the issue #182 of project `dynamic-prefer -ences`, we got the following comment from developers after they resolved the issue: "*It is a hazzle to keep track of all the frozen versions of some dependencies, especially for larger projects. I think it would be good to get an automatic notification as maintainer somehow, if one of your dependencies has locked its own libraries on a specific version.*"

For 43 predicted DC issues of **Type 2**, 20 of them have been confirmed by developers although these issues may not cause build failures immediately. We observed that in issues #295 [15] of `sherlock` and #2729 of `plaso` [14], WATCHMAN's warnings quickly caught developers' attention, and they added labels "bug" and "deployment problem" to these two issue reports.

Among the 63 confirmed DC issues (including both detected and predicted ones), developers resolved 54 (85.7%) of them following our suggested solutions. For example, in issue #9 [29], a build failure was introduced into project `webinfo`, due to an issue of **Pattern A.a**. We provided four solutions, and developer finally chose the one WATCHMAN generated based on **Strategy 2** to resolve this conflict. An encouraging comment "*thanks for recommending, the solution really meaningful, that's awesome!*" has been left by the developers. For the remaining 9 confirmed issues, for which our fixing solutions have not been adopted by developers, we found that these projects can be sensitive to certain library upgrades or downgrades and our suggested changes may introduce other side effects, such as vulnerability or compatibility issues into their projects (e.g., issue #16 in project `kindred`). In the future, we plan to further improve the quality of the fixing solutions generated by WATCHMAN.

### 5.2.2 *Feedback on WATCHMAN.*

Besides confirming and fixing our reported DC issues, some developers expressed interest in our tool WATCHMAN.

For example, a developer left the following comment in the PR #71 [34] for issue #70 of project `arxiv-submission-core`:

"*A better mechanism of maintaining the dependency constraints among projects on PyPI like what you did, is much-needed!*"

**Table 4: Results of 117 DC issues reported by WATCHMAN from July 11, 2019 to August 16, 2019**

| Manifestation | Issue reports |
|---|---|
| Pattern A.a | Issue#1, aucome; Issue#110, crypto; Issue#1, OrcaSong; Issue#2, pypmml-spark; Issue#138, toolium; Issue#26, GatewayFramework; Issue#56, Airbnb-data Issue#2, Runcible; Issue#95, identification; Issue#96, identification; Issue#1813, tasking-manager; Issue#356,Archery; Issue#325, bocadillo; Issue#21, crema; Issue#4, what-digit-you-write; Issue#9, webinfo-crawler; Issue#35, zarp; Issue#4, open-helpdesk; Issue#5, languagecrunch; Issue#103, account-creator; Issue#9, jawfish; Issue#212, openpose-plus; Issue#16, kindred; Issue#13, Generator-GUI; Issue#3, tabular; Issue#5, whats-bot; Issue#65, armory-bot; Issue#39, derrick; Issue#16, Historical-Prices; Issue#688, dxr; Issue#18526, erpnext; Issue#1, scrapy-qtwebkit; Issue#4778, InstaPy; Issue#2, api-indotel; Issue#145, cert-issuer; Issue#146, django; Issue#4, pymacaron; Issue#1, mgz-db; Issue#1, twitterbots; Issue#2, gremlin; Issue#17, AWSBucketDump; Issue#198, fabric-cli; Issue#1, BlockCluster; Issue#3, gateway; Issue#2, beauty_image; Issue#1389, Indy-node; Issue#130, swapi; Issue#279, explorer; Issue#34 footmark; Issue#3, driver-acs; Issue#56, driver-napi; Issue#11, simulator; Issue#9, Friends-Finder; Issue#1, chatbot-template; Issue#545, djangopackages; Issue#2048, cadasta-platform; Issue#122, adminset; Issue#45, Wallpaper; Issue#21, ltiauthenticator; Issue#28, cryptography; |
| Pattern A.b | Issue#243, bakerydemo; Issue#4, pytools; Issue#70, Osmedeus; Issue#101, aldryn-search; |
| Type 1 | Issue#182, dynamic-preferences; Issue#20, ldapdomaindump; Issue#326, py-cluster; Issue#986, faker; Issue#717, newspaper; Issue#120, mixer; Issue#3, client-python; Issue#75, PyInquirer; Issue#953, compressor; Issue#26, certstream; |
| Type 2 | Issue#8, AutoCrawler; Issue#31, BBScan; Issue#492, pywb; Issue#8, ct-exposer; Issue#71, EagleEye; Issue#1179, mythril; Issue#1, frida-util; Issue#34, python-urwid; Issue#4, SecurityManageFramwork; Issue#295, sherlock; Issue#2077, freqtrade; Issue#36, trains; Issue#298, glastopf; Issue#5, Machine-Learning-with-Python; Issue#569, kalliope; Issue#98, bless; Issue#70, arxiv-submission-core; Issue#2729, plaso; Issue#17, oauth-dropins; Issue#303, ripping-machine; Issue#27, ChannelBreakoutBot; Issue#167, tldextract; Issue#183, messytables; Issue#9, kuberdock-platform; Issue#42, python-weixin; Issue#25, NoDB; Issue#146, Photon; Issue#911, pyspider; Issue#7, fan; Issue#126, historical; Issue#49, stephanie-va; Issue#979, subliminal; Issue#56, WPSeku; Issue#3, zhihu-crawler; Issue#38, network-topology; Issue#647, marathon-lb; Issue#9, Konan; Issue#181, JBOPS; Issue#962, hangoutsbot; Issue#41, GyoiThon; Issue#120, automation-tools; Issue#4, start-vm; Issue#10, ahmia-index; |

Status 1: The issues had already been fixed using our suggested solutions; Status 2: The issues had already been fixed using other solutions; Status 3: The issue was confirmed and being fixed using our suggested solutions in progress. Status 4: The issue was confirmed as DC issues and being fixed using other solutions in progress. Status 5: The issues were pending.
We do not present the link of these issues due to page limit. The detailed information of them can be found on WATCHMAN's homepage (**http://www.watchman-pypi.com/buglist**).

In issue #492 of project `pywb`, we received an encouraging feedback from an experienced lead developer who is also the founder of `webrecorder` community [4]:

"*Are you an 'automation' written by Github community to help resolve dependency conflict issues for Python projects? If so, a piece of nice work! I'd say this is a good approach, a nice friendly bot to inform of potential dependency issues.*"

We also received other positive comments. Such feedback indicates that monitoring library updates and detecting/predicting dependency conflicts is indeed important to, and welcomed by, real-world Python developers. The information provided by WATCHMAN is also useful to help developers diagnose DC issues in practice.

## 6 DISCUSSIONS

**Threats to validity.** Keyword search can introduce irrelevant issues into our dataset. Such noises pose a threat to the validity of our study results. Another threat is the errors in our manual analysis of the DC issues. To reduce these threats, three co-authors independently investigated all our collected DC issues and crossed-validated their analysis results.

**Limitations.** Our work has three limitations. First, we focus on DC issues that cause build failures. However, in some cases, the conflicts may lead to semantic inconsistencies, runtime errors or other consequences in Python projects. Second, the rules adopted in the predictive analysis can only help find a subset of all possible DC issues that may be induced by the two types of dependency smells. They are designed based on our observed real cases in the empirical study. Third, WATCHMAN currently is not able to detect all patterns of DC issues observed in our empirical study. We will address these limitations in future work.

## 7 RELATED WORK

**Dependency conflict.** Pradel et al. [44] studied the dependency conflicts among JavaScript libraries and proposed a detection strategy. Suzaki et al. [38] conducted an extensive case study of conflict defects, including conflicts on resource access, conflicts on configuration data, and interactions between uncommon combinations of packages. Soto-Valero et al. [46] studied the problem of multiple versions of the same library co-existing in `Maven Central`, and presented empirical evidence about how the immutability of artifacts in `Maven Central` supports the emergence of natural software diversity. Wang et al. [48] conducted an empirical study to characterize dependency conflicts in Java projects and developed RIDDLE to generate tests to collect crashing stack traces to facilitate DC issue diagnosis [49]. To the best of our knowledge, there is no previous work focusing on characterizing and detecting DC issues in the Python world.

**Studies of software ecosystem.** Software ecosystem research has been rapidly growing in recent years. Serebrenik et al. [45] perform a meta-analysis of the difficult tasks in software ecosystem research and identified six types of challenges, e.g., how to scale the analysis to the massive amount of data. Mens [42] further studied software ecosystem from the socio-technical view on software maintenance and evolution. Zimmermann et al. [51] studied the security risks in the `npm` ecosystem by analyzing data such as dependencies between packages and publicly reported security issues. Another study by Lertwittayatrai et al. [40] used network analysis techniques to study the topology of the JavaScript package ecosystem and extracted insights about dependencies and their relations. Our work studies software ecosystem from a novel perspective by taking into account the interference between the version constraints of upstream and downstream projects. We also propose a technique to continuously monitor dependency conflicts for Python projects.

## 8 CONCLUSION AND FUTURE WORK

DC issues are common in Python projects. In this work, we first conducted an empirical study on 235 real DC issues to understand the manifestation patterns and fixing strategies of DC issues. Motivated by our empirical findings, we then designed a technique,WATCHMAN, to continuously monitor the dependency conflicts for the `PyPI` ecosystem. Evaluation results show that WATCHMAN can effectively detect DC issues with a high precision and provide useful diagnostic information to help developers fix its detected issues. In the future, we plan to further improve the detection capability of WATCHMAN and generalize our technique to other Python library ecosystems such as `Anaconda` to make it accessible to more developer communities.

# REFERENCES


[1] 2019. Commit 7eeadeb. https://github.com/MycroftAI/adapt/commit/7eeadeb4744b7e2dd7a9aa61e0350c4e22350eba. (2019). Accessed: 2019-08-02.
[2] 2019. conda. https://conda.io/. (2019). Accessed: 2019-08-02.
[3] 2019. Dependency specification for Python. https://www.python.org/dev/peps/pep-0508/. (2019). Accessed: 2019-08-02.
[4] 2019. An experienced developer. https://github.com/ikreymer. (2019). Accessed: 2019-08-02.
[5] 2019. finance-dl. https://github.com/jbms/finance-dl. (2019). Accessed: 2019-08-02.
[6] 2019. Issue #1277. https://github.com/django/channels/issues/1277. (2019). Accessed: 2019-08-02.
[7] 2019. Issue #152. https://github.com/django/channels_redis/issues/152. (2019). Accessed: 2019-08-02.
[8] 2019. Issue #1586. https://github.com/crossbario/crossbar/issues/1586. (2019). Accessed: 2019-08-02.
[9] 2019. Issue #1824. https://github.com/allenai/allennlp/issues/1824. (2019). Accessed: 2019-08-02.
[10] 2019. Issue #2077. https://github.com/freqtrade/freqtrade/issues/2077. (2019). Accessed: 2019-08-02.
[11] 2019. Issue #2195. https://github.com/allenai/allennlp/issues/2195. (2019). Accessed: 2019-08-02.
[12] 2019. Issue #2483. https://github.com/allenai/allennlp/issues/2483. (2019). Accessed: 2019-08-02.
[13] 2019. Issue #25487. https://github.com/pandas-dev/pandas/issues/25487. (2019). Accessed: 2019-08-02.
[14] 2019. Issue #2729. https://github.com/log2timeline/plaso/issues/2729. (2019). Accessed: 2019-08-02.
[15] 2019. Issue #295. https://github.com/sherlock-project/sherlock/issues/295. (2019). Accessed: 2019-08-02.
[16] 2019. Issue #3118. https://github.com/pypa/pipenv/issues/3118. (2019). Accessed: 2019-08-02.
[17] 2019. Issue #32. https://github.com/ARMmbed/valinor/issues/32. (2019). Accessed: 2019-08-02.
[18] 2019. Issue #3404. https://github.com/docker/compose/pull/3404. (2019). Accessed: 2019-08-02.
[19] 2019. Issue #36. https://github.com/rianhunter/dbxfs/issues/36. (2019). Accessed: 2019-08-02.
[20] 2019. Issue #407. https://github.com/wandb/client/issues/407. (2019). Accessed: 2019-08-02.
[21] 2019. Issue #409. https://github.com/PyCQA/bandit/issues/409. (2019). Accessed: 2019-08-02.
[22] 2019. Issue #4431. https://github.com/docker/compose/issues/4431. (2019). Accessed: 2019-08-02.
[23] 2019. Issue #4669. https://github.com/psf/requests/pull/4669. (2019). Accessed: 2019-08-02.
[24] 2019. Issue #4674. https://github.com/psf/requests/pull/4674. (2019). Accessed: 2019-08-02.
[25] 2019. Issue #4675. https://github.com/psf/requests/pull/4675. (2019). Accessed: 2019-08-02.
[26] 2019. Issue #66. https://github.com/z4r/pythoncoveralls/issues/66. (2019). Accessed: 2019-08-02.
[27] 2019. Issue #70. https://github.com/j3ssie/Osmedeus/issues/70. (2019). Accessed: 2019-08-02.
[28] 2019. Issue #740. https://github.com/ARMmbed/yotta/issues/740. (2019). Accessed: 2019-08-02.
[29] 2019. Issue #9. https://github.com/lubosson/webinfocrawler/issues/9. (2019). Accessed: 2019-08-02.
[30] 2019. Librariesio. https://libraries.io/. (2019). Accessed: 2019-08-02.
[31] 2019. pip. https://pypi.org/project/pip/. (2019). Accessed: 2019-08-02.
[32] 2019. pip documentation. https://pip.pypa.io/en/stable/reference/pip_install. (2019). Accessed: 2019-08-02.
[33] 2019. pipenv. https://docs.pipenv.org/. (2019). Accessed: 2019-08-02.
[34] 2019. PR #71. https://github.com/arXiv/arxiv-submission-core/pull/71. (2019). Accessed: 2019-08-02.
[35] 2019. PyPI. https://pypi.org/. (2019). Accessed: 2019-08-02.
[36] 2019. virtualenv. https://virtualenv.pypa.io/en/latest/. (2019). Accessed: 2019-08-02.
[37] Pietro Abate and Roberto Di Cosmo. 2011. Predicting upgrade failures using dependency analysis. In *2011 IEEE 27th International Conference on Data Engineering Workshops*. IEEE, 145–150.
[38] Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2012. Why do software packages conflict?. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 141–150.
[39] Alexandre Decan, Tom Mens, and Maelick 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*. ACM, 21.
[40] Nuttapon Lertwittayatrai, Raula Gaikovina Kula, Saya Onoue, Hideaki Hata, Arnon Rungsawang, Pattara Leelaprute, and Kenichi Matsumoto. 2017. Extracting insights from the topology of the javascript package ecosystem. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 298–307.
[41] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 1013–1024.
[42] Tom Mens. 2016. An ecosystemic and socio-technical view on software maintenance and evolution. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 1–8.
[43] Fabio Nelli. 2015. Python data analytics. *Berkeley: Apress* (2015).
[44] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: Finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 741–751.
[45] Alexander Serebrenik and Tom Mens. 2015. Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*. ACM, 40.
[46] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. 2019. The Emergence of Software Diversity in Maven Central. In *MSR 2019-16th International Conference on Mining Software Repositories*. 1–11.
[47] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 644–655.
[48] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 319–330.
[49] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could I Have a Stack Trace to Examine the Dependency Conflict Issue?. In *Proceedings of the 41th International Conference on Software Engineering (ICSE)*.
[50] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*.
[51] Markus Zimmermann, Cristianalexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. *arXiv: Cryptography and Security* (2019).