PyPitfall: Dependency Chaos and Software Supply Chain Vulnerabilities in Python

Jacob Mahon
Computer Science Department
New Jersey Institute of Technology
Newark, New Jersey, USA
jpm233@njit.edu

Chenxi Hou
Computer Science Department
New Jersey Institute of Technology
Newark, New Jersey, USA
ch395@njit.edu

Zhihao Yao
Computer Science Department
New Jersey Institute of Technology
Newark, New Jersey, USA
zhihao.yao@njit.edu

***Abstract*—Python software development heavily relies on third-party packages. Direct and transitive dependencies create a labyrinth of software supply chains. While it is convenient to reuse code, vulnerabilities within these dependency chains can propagate through dependencies, potentially affecting downstream packages and applications. PyPI, the official Python package repository, hosts many packages and lacks a comprehensive analysis of the prevalence of vulnerable dependencies. This paper introduces PyPitfall, a quantitative analysis of vulnerable dependencies across the PyPI ecosystem. We analyzed the dependency structures of 378,573 PyPI packages and identified 4,655 packages that explicitly require at least one known-vulnerable version and 141,044 packages that permit vulnerable versions within specified ranges. By characterizing the ecosystem-wide dependency landscape and the security impact of transitive dependencies, we aim to raise awareness of Python software supply chain security.**

***Index Terms*—Software Supply Chain, Python, Dependency Analysis**

1. Introduction

Modern software engineering relies heavily on third-party packages, creating complex software supply chains. While this practice accelerates development by avoiding reinventing the wheel, it also introduces security risks. Vulnerabilities in one package can propagate through its dependencies, potentially affecting downstream packages and applications [1].

Python, first released in 1991 as Python 0.9.0 [2], has a rich ecosystem of packages contributed to and maintained by a large community of developers. At the time of writing, Python Package Index (PyPI), the official Python package repository, hosts 627,810 projects and over 6 million releases [3]. An empirical study in 2019 reported 178,592 packages in PyPI and 76,997 contributors, with 156,816,750 import statements [4]. The proliferation of packages and their dependencies has led to increased complexity and security concerns [5], [6]. We will take a closer look at PyPI and its ecosystem in §2.1.

Understanding the nature and extent of these dependencies is the first step in addressing their security risks. A package may have direct dependencies (packages that are directly required) and transitive dependencies, which are indirectly needed for the package due to nested dependencies. The chain of dependencies can be long and complex, as shown in §2.3. A single package may depend on hundreds of others, each with its dependencies, forming a deep software supply chain labyrinth. Unfortunately, vulnerabilities can exist anywhere within this structure and affect the entire chain.

Existing tools, such as `pip-audit` [7] and `in-toto` [6], focus on detecting known vulnerabilities in installed packages or during the Continuous Integration and Continuous Delivery (CI/CD) pipeline. Existing studies have also focused on detecting malware in PyPI [8] or characterizing the PyPI ecosystem [4], but not on analyzing the security dependency labyrinth of the entire ecosystem.

In this paper, we present PyPitfall, a quantitative analysis of vulnerable dependencies in the PyPI ecosystem. While we did not discover new vulnerabilities, we focused on analyzing the existing dependencies and the prevalence of dependencies on specific versions of packages known to be vulnerable. We analyze the dependency metadata of 378,573 PyPI packages and identify 4,655 packages that explicitly require a vulnerable package version and another 141,044 packages that allow for a vulnerable version in their dependency constraints. For those that require a vulnerable version, the package would not work if the vulnerable version was not installed or unavailable.

Through our ecosystem-wide study, we quantitatively analyze the dependency relationships among Python packages and the security risks of dependencies on stale packages with known vulnerabilities. Our work aims to raise awareness of the security implications of complex transitive dependencies in Python software supply chains. The main contributions of this paper are listed as follows:

- We present a comprehensive analysis of the PyPI ecosystem, including 378,573 packages, including their direct and transitive dependencies.
- We analyze the impact of transitive dependencies on the security of Python packages and identify 4,655 packages that explicitly require other packages with known vulnerabilities.

- We provide ecosystem-wide insights into Python software supply chain security. We have responsibly disclosed our findings to the Python Packaging Authority, which maintains PyPI [9].

2. Background

To understand the complexity of Python's dependency ecosystem and the associated security risks, we first provide an overview of the PyPI and its dependency model, and survey the existing works on Python supply chain security.

2.1. Python Package Index (PyPI)

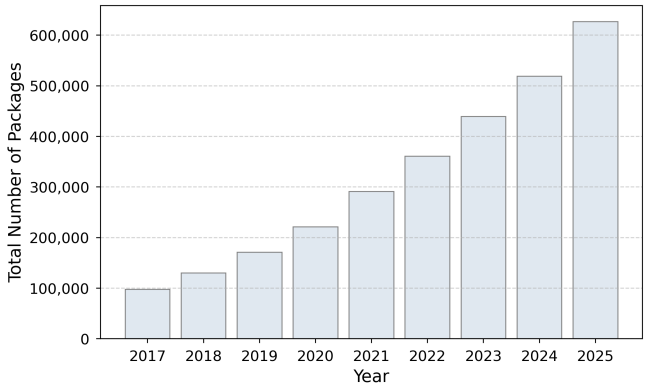


Figure 1: The number of packages in PyPI over the years.

PyPI serves as Python's official repository for third-party software libraries, hosting 627,810 projects totaling 27.0 TB of release files as of the time of writing [3], [10]. It enables developers worldwide to share and distribute their Python code, enabling a collaborative ecosystem for efficient software development. We looked up the number of packages in PyPI by looking at the cached versions of the PyPI website [3] since 2017 (the earliest date we could find) and show the trend in Figure 1.

When developers write codes that `import` other packages, *Direct Dependencies* are formed. *Transitive Dependencies* are formed when the dependent packages themselves require additional ones. As packages build upon one another, complex dependency relationships are often created without developers' awareness, sometimes spanning multiple layers of transitive dependencies. While the software supply chain facilitates rapid development through code reuse, its complexity introduces a fundamental trade-off: the convenience gained may be counterbalanced by the dependency maintenance efforts. Ensuring the security and availability of applications requires careful management and vetting of their dependencies, as well as keeping up with the latest versions and security advisories.

Another essential aspect of PyPI is that it is a community-driven platform, where anyone who passes basic registration and email verification can publish [11]. Malware has been found in PyPI packages [8], [12], [13], and several tools have been developed to detect malware in PyPI packages, such as Microsoft's OSSGadget [14] and Bandit4Mal [15]. Malware that is intentionally named to resemble legitimate packages (also known as typosquatting [16]) poses a primary risk to software supply chain security because it can be inadvertently imported. Although malware is out of the scope of this paper, the presence of security issues in PyPI packages can further complicate the dependency landscape, as any vulnerabilities in a package can propagate to its dependencies.

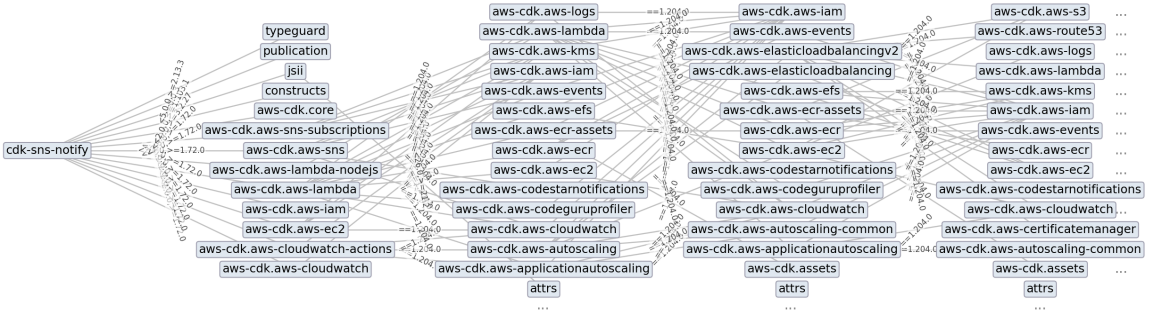
2.2. PyPI Dependency Model

Python package management is standardized through Python Enhancement Proposals 508 (PEP 508) [17], which defines a standard format for specifying direct dependencies. When a package is installed, `pip` resolves the dependencies recursively, downloading and installing the required packages. Interestingly, the resolution process is not always successful, as packages may have infeasible (e.g., in Figure 3) or conflicting version requirements. Following PEP 508 and PEP 440 [17], [18], a package may use logical operators such as ==, !=, >=, <=, >, and < to specify versions of a dependency that are required. Packages that require outdated dependencies can conflict with others that require newer versions of the same packages [19]–[21].

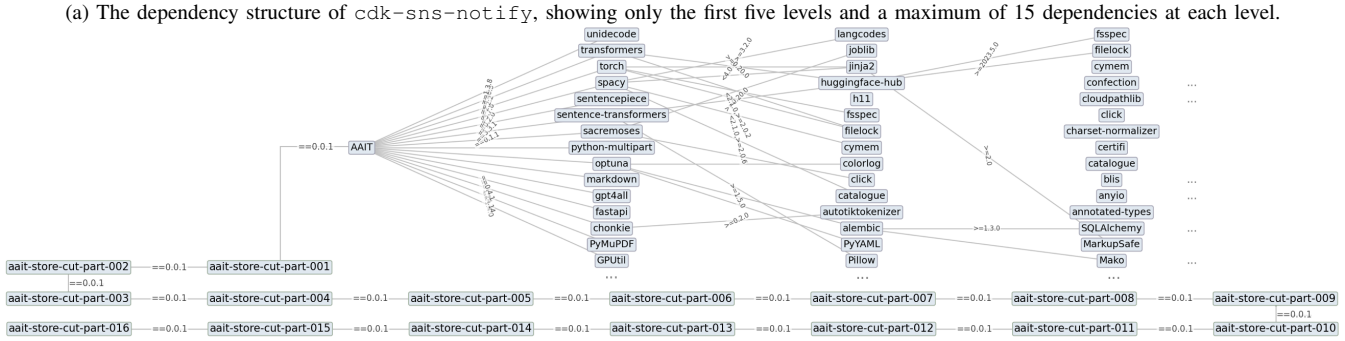
In Figure 2a, we showcase the dependency structure of `cdk-sns-notify` [22], a package related to cloud software engineering, which has a total of 49 dependencies spanning 22 levels of depth. The dependency is the second-longest acyclic dependency structure we found in our study, with the deepest one having 23 levels of depth but appearing to be using the Python versioning constraint system to implement the logic of Sudoku (see Figure 2b and §3.1).

The complexity has led to a well-known phenomenon called "dependency hell" [23], where developers struggle to resolve version conflicts and maintain compatibility among Python packages. Given that packages are often updated independently, a package that works well today may break tomorrow due to an update in one of its dependencies. Developers struggle to keep their code up-to-date with the latest versions of their dependencies, leading to a situation where they are forced to choose between using outdated packages or risking compatibility issues with newer versions [24], [25]. If a developer opts for the former, they may miss important security updates.

To cope with dependency conflicts, `pip` uses a backtracking resolver to find a compatible set of package versions by trying "every possible combination" of dependencies [26]. SmartPip has modeled dependency resolution as constraint satisfaction problems and proposed solving them using a Satisfiability Modulo Theories (SMT) solver [27]. UPCY [28] utilizes a graph-based algorithm to update outdated dependencies safely. Recent advancements of Large Language Models (LLMs) have also been applied to dependency resolution [29]. However, these approaches do not study or address the underlying issue of dependency complexity.



(a) The dependency structure of `cdk-sns-notify`, showing only the first five levels and a maximum of 15 dependencies at each level.



(b) The dependency structure of `aait-store-cut-part-016` (at the bottom-left corner of the figure), showing only the first 20 levels.

Figure 2: Simplified dependency structures of the two Python packages with the deepest dependency structures in the PyPI ecosystem. Boxes represent packages, and edges represent dependency relationships. Arrows are omitted for simplicity. The numbers on the edges indicate the version constraints of dependencies (if any).

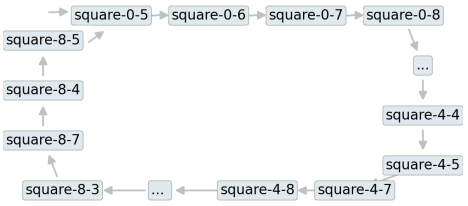


Figure 3: The dependency structure of `square-0-5`, showing a circular dependency that causes an infinite loop.

2.3. Software Supply Chain Security

The complex network of dependencies in PyPI packages described in §2.2 forms a software supply chain that is as strong as its weakest link. No matter how deeply nested a vulnerable package is, the entire chain is at risk if it exists. Software supply chain attacks have become a significant concern in recent years [6], [30]–[33]. Unfortunately, there is no perfect solution to this problem as each package is maintained (or lacks maintenance) by different developers. A study in 2019 [4] analyzed the PyPI ecosystem for framework, operating system, development status, license, and other metadata, but it did not focus on the security aspects of dependencies. Recent works have studied software supply chain security [34], [35], but a comprehensive analysis of the entire PyPI ecosystem is still lacking.

3. Motivations and Assumptions

3.1. Preliminary Analysis of PyPI Dependency

Given the PyPI ecosystem's reliance on third-party packages and their complex dependencies, this work aims to empirically analyze the security of the Python software supply chain, especially the risk exposed through transitive dependencies. To motivate our study, we conducted a preliminary analysis of the dependency structure of several packages available on PyPI. Figures 2a and 2b, respectively, show the (simplified) two longest acyclic dependency structures that we found in PyPI: Package `aait-store-cut-part-016` [36] has the longest acyclic dependency chain with a total of 117 dependencies spanning 23 levels, whereas package `cdk-sns-notify`'s dependencies span across 22 levels.

Moreover, we observed packages with unresolvable dependencies, where the dependency structure is not a Directed Acyclic Graph (DAG), resulting in circular dependencies. Figure 3 illustrates a case where the package `square-0-5` depends on itself after 75 jumps, causing the `pip install` command to run into an infinite loop. Upon further study, we find that the package names and dependency constraints were creatively utilized to enforce Sudoku rules [37].

3.2. Motivations

Our preliminary analysis of the PyPI ecosystem revealed that many packages depend on specific versions of other packages, which may be vulnerable. The sheer scale of the PyPI ecosystem, with over 627,810 packages, makes it impractical to analyze each package and its dependencies manually. Existing tools focus on scanning for known vulnerabilities in installed packages or during the CI/CD pipeline [6], [7]. A comprehensive, ecosystem-wide study is needed to navigate the Python supply chain security landscape. In this study, we aim to answer the following research questions:

- **RQ1:** What is the scale of the dependency complexity in the PyPI ecosystem?
- **RQ2:** To what extent do Python packages depend on packages with known vulnerabilities?
- **RQ3:** How do transitive dependencies affect the security of Python packages?

3.3. Trust Assumptions

In this study, we assume that Python package developers are not malicious and do not intentionally introduce vulnerabilities into their packages. However, as the number of dependencies grows, developers may not be aware of the known security issues in their dependent third-party packages, causing software security to be temporal [38]. Failure to update direct dependencies or unawareness of vulnerabilities in transitive dependencies leads to potential exposures to known bugs. Malware, typosquatting, and the intentional introduction of vulnerabilities are outside the scope of this work.

4. Design

We designed PyPitfall to systematically analyze the entire PyPI ecosystem to identify and assess the exposure of packages to known vulnerabilities in package dependencies. The analysis pipeline is shown in Figure 4. The pipeline consists of four main components: data collection, dependency resolution, version constraint calculation, and comparison of vulnerable versions.

4.1. Data Collection

The first step in our data collection process is to obtain the names of all PyPI packages using the official index [39] specified by PEP 503 [40]. As our study focuses on the current state of the PyPI ecosystem, we used the latest version of all available packages. Because PyPI does not provide a direct way to obtain the dependency metadata of a package, we had to use tools to collect this information. Indeed, `pip` needs to use a backtracing algorithm to resolve each layer of transitive dependencies [26]. We reuse this mechanism to dry-run the installation of each package through a third-party tool called `Johnnydep` [41]

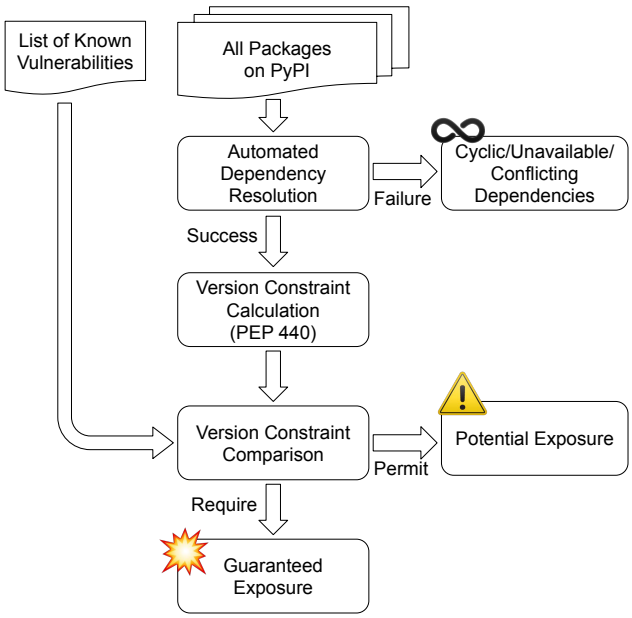


Figure 4: The architecture of PyPitfall.

and record the dependency metadata (including names and version specifiers). Collecting this data and its analysis helps us understand **RQ1**.

Vulnerability information typically comes from the National Vulnerability Database (NVD) [42] or other security advisories. We are aware of the Python Packaging Advisory Database [43] that provides an extensive list of known vulnerabilities in various Python packages. However, due to the sheer number of packages in the PyPI ecosystem, we decided to focus on a curated list of known vulnerabilities that affect Python libraries to showcase and raise awareness of the issue of using vulnerable packages in the PyPI ecosystem and leave the analysis of the larger known vulnerability dataset for future work. We will provide more details in §5.

For each vulnerability, we record the affected package name, version, severity, and the range(s) of vulnerable versions. Some ranges are concise, such as `<1.3.0`, while others involve logic operators, such as (`>=2.0.0` $\wedge$ `<2.0.6`) $\vee$ (`<1.26.17`), as seen in CVE-2023-43804.

4.2. Dependency Constraint Calculation

As we utilize the `pip` to resolve the dependencies, we rely on it to determine whether a package's dependency constraint can be satisfied. As shown in Figure 4, the ones that `pip` fails to resolve will be recorded as `unresolvable` dependencies and excluded from further analysis. In §2.2, we have shown a case where `pip` runs into an infinite loop due to the circular dependencies.

Constraint Aggregation: We also need to aggregate the version constraints of each package's dependencies to avoid false positives. A package P may depend on another package D via multiple paths. Each path imposes different version constraints. For example, $P \rightarrow A \rightarrow D_{\geq 1.5}$ and

$P \rightarrow B \rightarrow D_{\geq 1.7}$. The *effective* constraint on D required by P is the intersection of all constraints imposed along all paths. The effective constraint becomes $D_{\geq 1.7}$ in this example. We calculate the effective constraint set, denoted as S, for every dependency D of the package P.

PEP 440 Versioning Standard: PEP 440 [18] defines the versioning standard for Python packages, which is widely adopted but not universally followed by all packages. We perform fault-tolerant parsing to convert all version strings into logical structures for comparison, such as normalizing version strings (e.g., treating 1.0 and 1.0.0 as equivalent) and handling pre- and post-release tags (e.g., `beta`, `rc`, `dev`). Our design utilizes Python's standard `packaging` library to ensure compatibility with PEP 440.

4.3. Vulnerable Version Comparison

Let S be the set of versions of dependency D allowed by package P's *effective* constraints, and let V be the set of versions of D known to be vulnerable, we compute the intersection $I = S \cap V$.

We define a **Guaranteed Exposure** as the condition where the entire set of allowed versions falls within the vulnerable set.

$$\text{Guaranteed Exposure if } S \subseteq V$$

Any successful installation of P will *inevitably* result in a vulnerable version of D being installed. The installation will fail if the vulnerable version is yanked (removed) from PyPI.

We define a **Potential Exposure** if the intersection of the two sets is non-empty, and the required set is not fully contained in the vulnerable set.

$$\text{Potential Exposure if } (I \neq \emptyset) \wedge (S \not\subseteq V)$$

The dependency D may be installed in a vulnerable version, depending on how `pip` resolves the dependencies based on other packages' constraints. In this case, the dependency constraints *allow for* the installation of both vulnerable and non-vulnerable versions. Although a *Potential Exposure* is not as severe as a *Guaranteed Exposure*, it remains a concern because the dependency resolution process will not update the version of D to a non-vulnerable version if the vulnerable version exists in the environment [26].

The comparison outputs the list of packages guaranteed or potentially exposed. This analysis helps us understand **RQ2** and **RQ3** by quantifying known vulnerabilities' exposure in the PyPI ecosystem based on documented vulnerabilities and explicit dependency constraints.

5. Implementation Details

In this section, we detail the implementation of PyPitfall, which analyzes the known vulnerabilities in the Python package dependencies.

5.1. Data Collection

5.1.1. PyPI Package List. We used the PyPI Simple Index [39] to retrieve a comprehensive list of packages available on PyPI. We obtained 616,266 valid package names out of 627,810 packages claimed on PyPI (98.2%). We estimate that the remaining packages are either invalid or unavailable for download.

5.2. Known Vulnerability List

Common Vulnerabilities and Exposures (CVE) is a standardized method of identifying vulnerabilities in software and is maintained in established public databases, such as NVD and MITRE [42], [44]. We searched NVD and MITRE databases for the term "Python library" to find known vulnerabilities in Python libraries. Each identified CVE entry was manually curated to ensure: (1) the vulnerability is related to a Python package in PyPI (excluding built-in Python libraries), and (2) the entry provided sufficient information regarding the affected package name and vulnerable version ranges. As discussed in §4.1, we focused on CVEs that affect Python libraries (thus the search term "Python library"). We curated 67 CVE entries that met our criteria, with 26 not seen in the Python Packaging Advisory Database [43] as of the time of writing [43]. We have suggested including these CVEs in the Python Packaging Authority maintainers.

5.3. Dependency Retrieval

5.3.1. Tool Selection. The `Johnnydep` [41] tool was selected for resolving package dependencies. Under the hood, `Johnnydep` uses the `pip` API to "dry-run" the installation of each package and triggers dependency resolution without actually downloading the package. As discussed in §3.1 and 4.2, the dependency resolution process relies on correct definitions of dependencies by individual packages. Our workflow treats the dependency resolution as a black box, and we do not attempt to resolve any issues that arise during the process. If the package fails to resolve, we record it as `unresolvable` and exclude it from the next steps. `Johnnydep` outputs a tree structure of the dependencies, which is then parsed into `JSON` format to represent the dependency information, as described below.

5.3.2. Automated Data Collection Workflow. A Python script was developed to automate the following steps for dependency collection:

- **Input Handling:** The partitioned (for parallelism) package lists were read from a file, and each package name was processed sequentially.
- **Dependency Extraction:** For each package name, the script invoked `Johnnydep` as a process and waited for its completion.
- **Error Handling:** A `try-except` block within a `while` loop managed potential errors during `Johnnydep` execution.

- **Output:** The raw dependency structure outputs generated by `Johnnydep` for each successfully processed package were captured and stored.
- **Output Formatting:** The raw outputs were parsed into a structured `JSON` format for the next steps.

The parallelized data collection was mainly distributed across four machines running Ubuntu 22.04, with varying hardware configurations (4 - 16 CPU cores, 8 - 32 GB RAM, and wired/Wi-Fi connections). The dependency collection process's total runtime was approximately 17 days, accounting for the time taken to process the 616,266 packages and occasional interruptions due to system crashes.

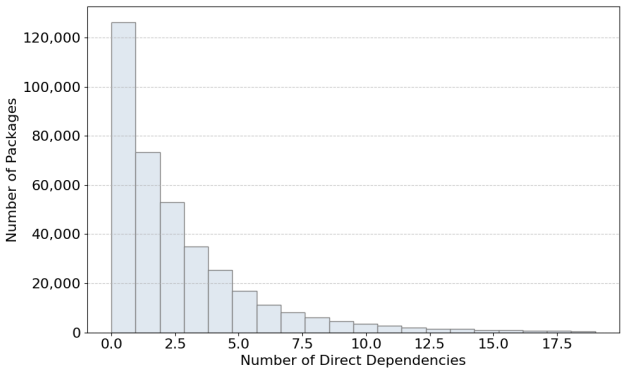


Figure 5: Distribution of the number of direct dependencies per package.

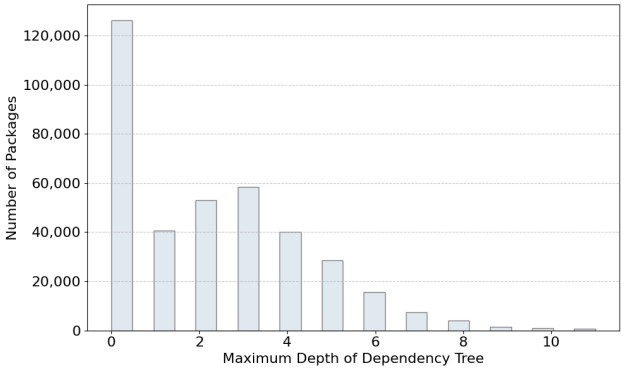


Figure 6: Distribution of the depth of dependency chains.

5.3.3. Challenges in Dependency Collection. Despite the automated workflow, several challenges were faced during the data collection process. First, circular dependencies have caused the dependency resolution to enter infinite loops (see §3.1), and in some cases, the system would crash due to excessive resource consumption. Detecting circular dependencies is challenging, as they can occur at any level of transitive dependencies. We rely on two methods to mitigate these issues: (1) `Johnnydep` has a built-in mechanism to detect previously-visited nodes and break out of the loop, and print `<circular dependency marker>` in the output, and (2) our workflow uses a simple timeout mechanism to terminate the process or manually recover a machine that crashed.

Second, some packages were not resolvable due to incorrect or stale dependencies or compatibility issues with the system. Certain libraries require a specific version of Python. For example, analysis cannot be conducted when resolving the dependency for the package `snakemake-interface-report-plugins`, as Johnnydep needs a version of Python greater than 3.11 to work. The system used Python 3.10, preventing the package's dependencies from being resolved. Other libraries require a specific CPU architecture or operating system environment, which can lead to unexplained permission errors and resolution failures. Direct or transitive dependencies on non-existent packages, including the ones that never existed or were yanked, can cause the resolution to fail. Deprecated package names can also lead to failures, such as using 'sklearn' instead of the canonical 'scikit-learn' [45]. These factors contributed to 237,693 (38.6%) packages being unresolvable, and 378,573 packages' dependencies were successfully resolved.

5.3.4. Vulnerability Matching and Classification. After collecting dependency data, we compared the dependencies against the known vulnerabilities in our curated CVE list using a Python program that we developed. Due to the large size of the dependency dataset (about 32 GB), we used `ijson` [46], an iterative `JSON` parser, to load the data in a memory-efficient manner. The vulnerability data was also loaded into memory as a map, with package names as keys, for efficient lookups for each encountered dependency.

We use `packaging.version.parse`, which supports PEP 440 versioning standards, to compare the version strings. We also handle potential logical operators in vulnerability ranges (see §4.3) as multiple comparisons. An iterative Depth-First Search (DFS) algorithm (implemented using `collections.deque` as a stack) was used to traverse the nodes in each dependency structure in our dataset. Each item that is pushed onto the stack represents a node to visit, which contains the dependency package name, its associated data (e.g., any further transitive dependencies), the traversal path from the top-level package as a list of package names, and the *direct* version constraints imposed on this dependency by its parent in the current traversal path.

We checked if the dependency package name was a key in the vulnerability map for each dependency node popped from the stack. If it does, we iterate through each CVE associated with the package name and each corresponding vulnerability constraint set (V). We then compare the direct dependency constraints (S_{direct}, representing the constraints from the parent node in the current path) against the vulnerability constraint set (V). The algorithm ensures low memory usage and efficient processing by only storing the current path and the direct constraints for each dependency node.

As discussed in §4.3, our comparison algorithm identifies two types of vulnerabilities: potential exposure and guaranteed exposure. It first determined

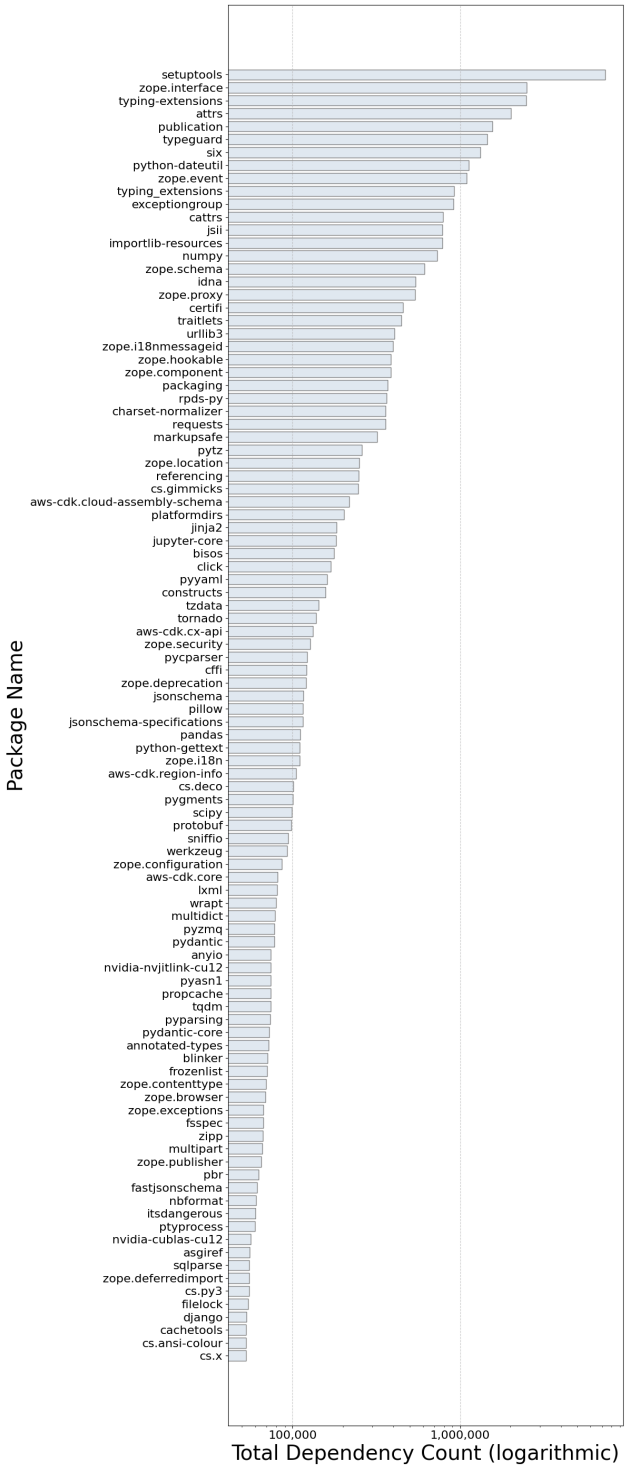


Figure 7: Number of dependency occurrences in the entire PyPI ecosystem (including both direct and transitive) among the top 100 most depended-upon packages.

if any overlap existed between the version set defined by S_{direct} and the set V by calculating lower and upper version bounds for both sets and utilizing

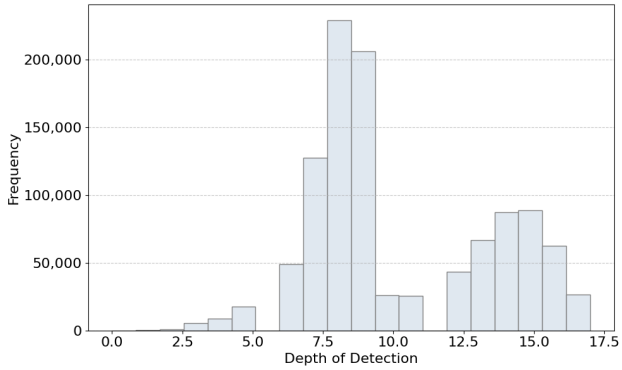


Figure 8: Distribution of detection depths of circular dependencies among all 1,075,559 detected occurrences of circular dependencies.

`packaging.specifiers.SpecifierSet` for checking specific version constraints (==). If an overlap is found, the algorithm checks whether S_{direct} is entirely contained within the vulnerability set V. The check compares the version bounds of S_{direct} and V. We ran the matching algorithm on a single machine with 8 CPU cores and 32 GB of RAM, which took approximately 7 days to complete. Finally, we save the findings in a `JSON` file for further analysis.

6. Result Analysis

We present the results of our analysis of the PyPI ecosystem, focusing on the dependency structures of 378,573 packages, which we successfully resolved for dependencies. Our vulnerability matching algorithm (see §5.3.4) matched 4,655 packages with *Guaranteed Exposures* to known vulnerabilities, and another 141,044 packages with *Potential Exposures*.

6.1. Dependency Complexity

Our study revealed a highly interconnected web of dependencies in the PyPI ecosystem in these 378,573 packages (which we refer to as the top-level packages). There are 57,767 unique packages and 47,974,375 dependency nodes identified in the dependency structures of these packages. On average, each top-level package has 2.6 direct dependencies and 129.6 (non-unique) transitive dependencies that span an average of 2.3 levels of depth.

Figure 7 shows the number of occurrences in the entire PyPI ecosystem (including direct and transitive dependencies) among the top 100 most depended-upon packages. The most depended-upon package is `setuptools`, a standard library for packaging Python projects, with 7,329,798 occurrences. The second most depended-upon package is `zope.interface`, a library for defining interfaces in Python, with 2,501,533 occurrences among all dependency structures. Figure 5 illustrates the distribution of direct dependencies per package. Despite most packages having a

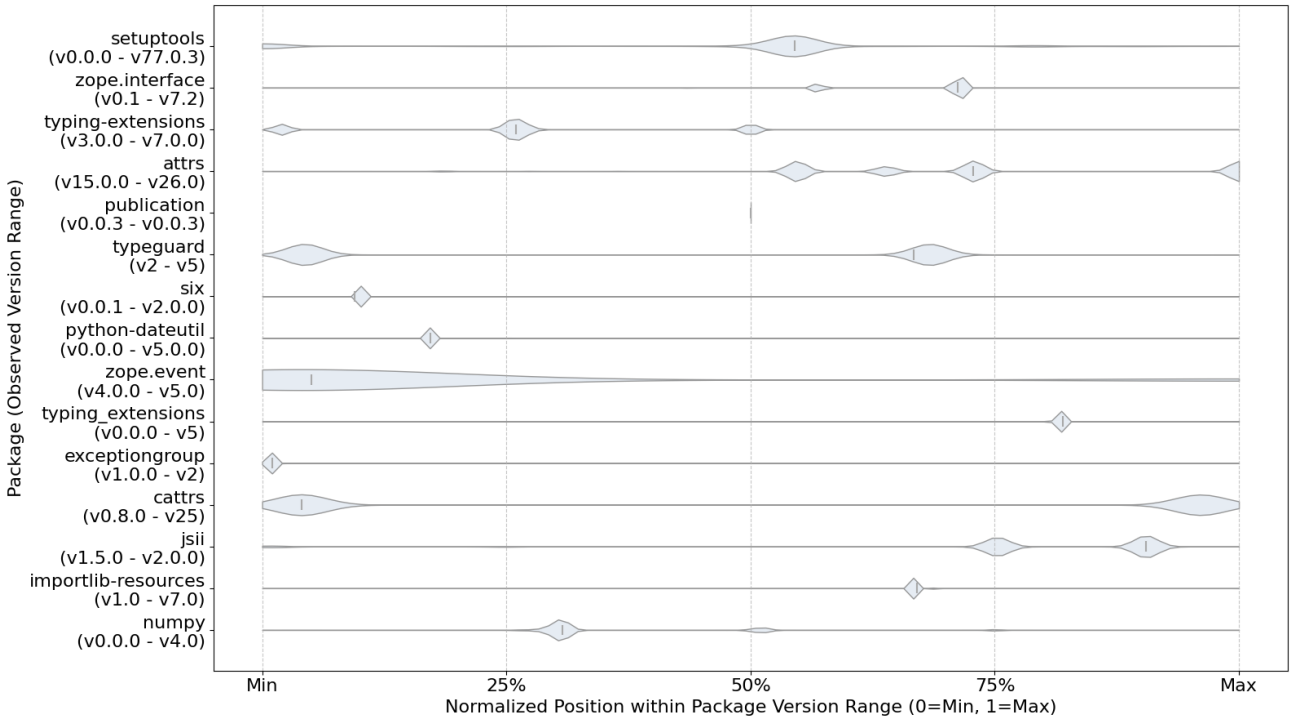


Figure 9: Version density of the top 15 most dependent packages. Each violin plot shows the distribution of the requested versions of the package by its dependents across the PyPI ecosystem. The width of the violin plot indicates the density of the version requests.

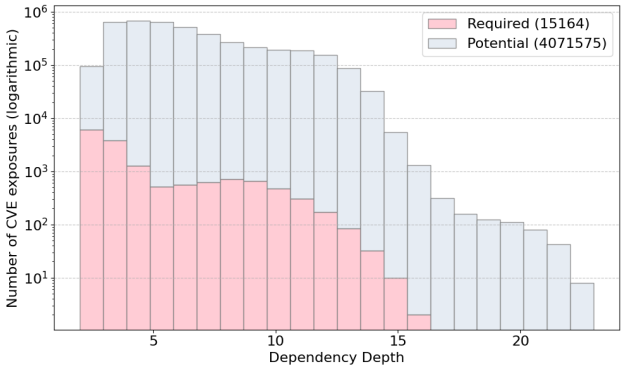


Figure 10: Number of guaranteed exposures ('Required' dependencies on vulnerable versions, shown in red), and potential exposures ('Potential' dependencies, shown in blue).

small number of direct dependencies (fewer than 2), a diminishing number of packages have many direct dependencies. Figure 6 illustrates the distribution of dependency structure depths per package. The graph shows a right-skewed bell curve with the most common depth being 1, indicating that most packages have shallow dependency structures, but a few have deep dependency structures (up to 23 levels of depth, as described in §3.1).

6.2. Circular Dependency Analysis

1,075,559 circular dependencies were detected by the dependency resolver and were excluded from the statistics above. Figure 8 shows a bimodal distribution of detection depths of circular dependencies. The most frequent value is around 8.5 levels, and the second peak is around 14.5 levels of depth. On average, circular dependencies span 10.3 levels of depth, showing that they are not trivial to resolve and are hard to detect. Compared with the average depth of PyPI packages (2.3), the circular dependencies are more likely to occur at deeper levels.

6.3. Version Analysis

We conducted a version study of the package dependencies across the PyPI ecosystem. Despite the uniformity in the versioning label format (PEP 440), we found that the assignments and stepping of version numbers are arbitrary. For example, the `setuptools` package (ranked #1 among the most depended-upon packages) has version ranges from `0.6` to `79.0`, whereas the `publication` package (ranked #5) has version ranges from `0.0.1` to `0.0.3`.

The version requirements of a package are also multi-faceted: for example, the `setuptools` package has 628 unique sets of version constraints, and `zope.interface` has 59. Figure 9 visualizes the requested version density of the top 15 most depended-upon packages. The x-axis

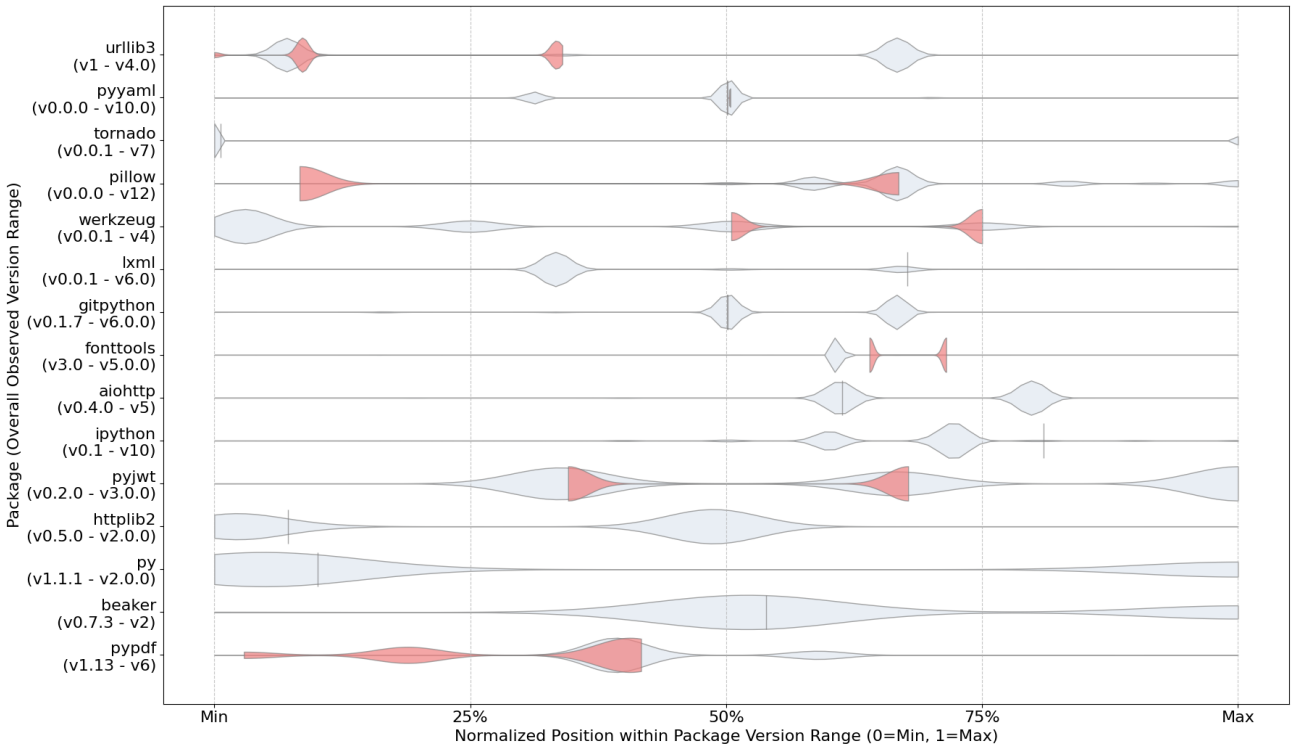


Figure 11: The version density distribution of the required exposures for the top 15 most depended-upon vulnerable packages is shown. The x-axis shows the normalized version number. Grey violin plots indicate the density of version requests by dependents, and red plots indicate the density of vulnerable versions.

displays the *normalized* version number, calculated by dividing the version number by the maximum version range requested by its dependents. Each violin plot's width indicates the density of its version requests. A wider section of the violin plot indicates more requests for that version range. This figure shows that most packages have a focused and small number of version requests.

6.4. Python Software Supply Chain Security

Using the 67 curated CVEs, we found 4,655 guaranteed exposures and 141,044 potential exposures. The average depth of the dependency paths for guaranteed exposures is 4.1, and the average depth of potential exposures is 6.2. The exposure depths are higher than the average depth of all packages (2.3), potentially indicating that the vulnerable packages are more likely to be at deeper levels of the dependency chains. As shown in Figure 10 (in logarithmic scale), we noticed that the number of guaranteed exposures diminishes faster than the number of potential exposures, and there is no guaranteed exposure starting from a depth of 17. As discussed in §5.2, the CVEs we used for our analysis are not exhaustive, as we focused on Python libraries, and a larger set of CVEs may yield more findings. We leave this for future work.

6.5. Vulnerable Version Density

Figure 11 shows the violin plots for the required and vulnerable versions of the top 15 most depended-upon vulnerable packages. Similar to Figure 9, version numbers were normalized. Note that a package may have multiple disjoint vulnerable version ranges due to various CVEs or multiple version ranges associated with a single CVE. Overlapping violin plots indicate that the requested versions are vulnerable, exposing them to known vulnerabilities in the software supply chain. Even a small overlap can lead to a large number of transitively dependent packages being affected. For example, the `urllib3` package's vulnerability, CVE-2024-37891, has 2,169 guaranteed exposures. Motivated by this finding, we conducted a case study on the `urllib3` package in §6.6.

6.6. Case Study: `urllib3`

`Urllib3` is a widely adopted Python HTTP library. It is a dependency of many popular packages, including another HTTP library, `requests`, which is among the top 100 most depended-upon packages (see Figure 7). Our study found that `urllib3` occurs 407,333 times in the dependency chains, with the deepest occurrences at 22 levels.

As shown in Table 1, our curated vulnerability list includes several CVEs in `urllib3`. Note that these are not all

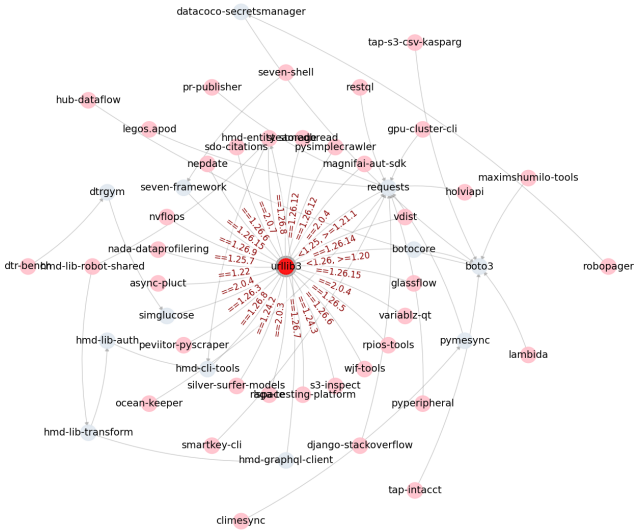


Figure 12: Selected guaranteed exposure dependency paths for the `urllib3` package. Nodes are packages. The edges indicate the dependency paths and required version constraints. For simplicity, we only show the last-level version constraints.

CVE ID	Avg. Depth	Potential Pkgs	Required Pkgs	Severity
CVE-2020-7212	6.3	99277	60	High
CVE-2021-28363	6.2	99666	46	Medium
CVE-2023-43804	6.2	100180	1732	High
CVE-2023-45803	6.2	100185	1755	Medium
CVE-2024-37891	6.2	100210	1906	Medium

TABLE 1: Summary of CVEs affecting `urllib3`, showing the number of affected top-level packages.

CVEs in `urllib3`, but only the ones we studied. Among these vulnerable versions of `urllib3`, we matched 1,926 unique top-level packages with a guaranteed exposure and 100,213 packages with a potential exposure. The exposures are respectively 41.4% and 71.1% of our total guaranteed and potential exposure findings.

We selectively visualized the dependency paths that introduced guaranteed exposures to `urllib3` in Figure 12. The dark red node indicates the `urllib3` package. The light red nodes indicate the top-level packages that depend on it through the transitive dependencies (shown as blue nodes). The graph shows that many popular packages propagate the CVEs in `urllib3` down the dependency chains.

7. Related Work

7.1. Software Supply Chain Security

Ellis et al. [1] first proposed risk analysis for the software lifecycle and externally-sourced software, which is later known as the software supply chain. The in-toto project [6] focused on cryptographic provenance to ensure integrity of software supply chains in different development stages. Studies by Enck et al. [30] and Hammi et al. [35] emphasized the management of software supply chains, including the build and deployment stages. Ladisa et al. [47] proposed a taxonomy for software supply chain attacks. OSV-SCALIBR [48] analyzed software composition to scan for known vulnerabilities. Our work shares a similar goal of improving software supply chain security.

7.2. Malicious Package Detection and Mitigation

Malicious packages infiltrate the software supply chain through various means. Typosquatting and combosquatting attacks, where attackers publish packages with names similar to legitimate ones, have been studied extensively in the Python ecosystem [8], [13], [16], [32], [47]. ZTD-JAVA [33] used permission control to prevent malicious dependencies from affecting other software. PyPitfall complements these studies by analyzing the risks posed by dependencies on packages with known vulnerabilities, which are unintentionally included in the dependency paths of legitimate packages.

7.3. Programming Analysis

Static analysis is one of the most common techniques for identifying and mitigating software bugs [49], [50]. Ruohonen et al. [51] conducted a large-scale static analysis of PyPI packages to identify common security issues. Dynamic analysis, on the other hand, attempts to identify vulnerabilities by executing the code and has been used to analyze Python, Java, and C/C++ programs [52]–[54]. PyPitfall is not a static or dynamic analysis tool; instead, it focuses on the inter-package relationships and the security implications arising from dependency declarations (metadata).

8. Limitations

As our study relies on the dependency resolution provided by `pip` (via `Johnnydep`), inaccuracies or omissions in the dependency resolution process may lead to an underestimation of the actual risk. We successfully resolved dependencies for 378,573 packages (60.3%), but the remaining unsuccessful packages can hide additional software supply chain risks. Moreover, our vulnerability analysis utilized a curated list of 67 CVEs, specifically targeting Python libraries. Using a broader vulnerability dataset, such as the full Python Packaging Advisory Database [43], would likely reveal more exposures.

9. Conclusion

This paper presents PyPitfall, a comprehensive analysis of the PyPI ecosystem's dependency landscape. By analyzing the dependency metadata of 378,573 PyPI packages, we quantified the extent to which packages rely on versions with known vulnerabilities. Our study reveals that 4,655 packages have guaranteed dependencies on known vulnerabilities, and

141,044 packages allow for the use of vulnerable versions. Our findings underscore the need for enhanced security awareness in the Python software supply chain.

Acknowledgements

This work was supported in part by the New Jersey Institute of Technology new faculty startup fund.

Use of AI-based tools: Google Gemini 2.5 was utilized to revise §1, §4, correcting any grammar and phrasing issues, and to assist with the formatting of Figure 9, 11.

References

[1] R. J. Ellison, J. B. Goodenough, C. B. Weinstock, and C. Woody, "Evaluating and mitigating software supply chain security risks," *Software Engineering Institute, Tech. Rep. CMU/SEI-2010-TN-016*, 2010.

[2] G. Van Rossum, "Python programming language." in *USENIX annual technical conference*, vol. 41, no. 1. Santa Clara, CA, 2007, pp. 1–36.

[3] "PyPI - the Python Package Index," https://pypi.org/, 2025.

[4] E. Bommarito and M. Bommarito, "An empirical analysis of the python package index (pypi)," *arXiv preprint arXiv:1907.11073*, 2019.

[5] K. Gao, R. He, B. Xie, and M. Zhou, "Characterizing deep learning package supply chains in pypi: Domains, clusters, and disengagement," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–27, 2024.

[6] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in *Proc. USENIX Security 2019*, 2019, pp. 1393–1410.

[7] "pip-audit," https://pypi.org/project/pip-audit/, 2025.

[8] D. Vu, Z. Newman, and J. S. Meyers, "Bad snakes: Understanding and improving python package index malware scanning," in *Proc. IEEE/ACM ICSE 2023*, 2023, pp. 499–511.

[9] "Python Packaging Authority," https://www.pypa.io/, 2023.

[10] "Statistics - PyPI," https://pypi.org/stats/, 2025.

[11] "Packaging Python Projects - Python Packaging User Guide," https://packaging.python.org/en/latest/tutorials/packaging-projects/, 2020.

[12] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, "An empirical study of malicious code in pypi ecosystem," in *Proc. IEEE/ACM ASE 2023*, pp. 166–177.

[13] W. Jiang, B. Çakar, M. Lysenko, and J. C. Davis, "Detecting active and stealthy typosquatting threats in package registries," *arXiv preprint arXiv:2502.20528*, 2025.

[14] "GitHub - Microsoft OSSGadget," https://github.com/microsoft/OSSGadget, 2025.

[15] "GitHub - Bandit4Mal," https://github.com/lyvd/bandit4mal, 2022.

[16] D. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *2020 ieee european symposium on security and privacy workshops (euros&pw)*. IEEE, 2020, pp. 509–514.

[17] "PEP 508 - Conditional Dependencies," https://peps.python.org/pep-0508/, 2015.

[18] "PEP 440 – Version Identification and Dependency Specification," https://peps.python.org/pep-0440/, 2014.

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

[20] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, and L. Wang, "Towards better dependency management: A first look at dependency smells in python projects," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1741–1765, 2022.

[21] X. Jia, Y. Zhou, Y. Hussain, and W. Yang, "An empirical study on python library dependency and conflict issues," in *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2024, pp. 504–515.

[22] "cdk-sns-notify - PyPI," https://pypi.org/project/cdk-sns-notify/, 2024.

[23] "The Nine Circles of Python Dependency Hell," https://medium.com/knerd/the-nine-circles-of-python-dependency-hell-481d53e3e025, 2015.

[24] "Breaking Changes, Breaking Trust," https://www.endorlabs.com/learn/breaking-changes-breaking-trust, 2024.

[25] F. Reyes, B. Baudry, and M. Monperrus, "Breaking-good: Explaining breaking dependency updates with build analysis," in *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, 2024, pp. 36–46.

[26] "pip resolver," https://pip.pypa.io/en/stable/topics/dependency-resolution, 2025.

[27] C. Wang, R. Wu, H. Song, J. Shu, and G. Li, "smartpip: A smart approach to resolving python dependency conflict issues," in *Proc. IEEE/ACM ASE 2022*, pp. 1–12.

[28] A. Dann, B. Hermann, and E. Bodden, "Upcy: Safely updating outdated dependencies," in *Proc. IEEE/ACM ICSE 2023*, pp. 233–244.

[29] F. Lukas and K. Jens, "Automatically fixing dependency breaking changes," in *Proc. ACM FSE 2025*.

[30] W. Enck and L. Williams, "Top five challenges in software supply chain security: Observations from 30 industry and government organizations," *IEEE Security & Privacy*, vol. 20, no. 2, pp. 96–100, 2022.

[31] D. Wermke, J. H. Klemmer, N. Wöhler, J. Schmüser, H. S. Ramulu, Y. Acar, and S. Fahl, ""always contribute back": A qualitative study on security challenges of the open source supply chain," in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2023.

[32] P. Ladisa, S. E. Ponta, N. Ronzoni, M. Martinez, and O. Barais, "On the feasibility of cross-language detection of malicious packages in npm and pypi," in *Proceedings of the 39th Annual Computer Security Applications Conference*, 2023, pp. 71–82.

[33] P. C. Amusuo, K. A. Robinson, T. Singla, H. Peng, A. Machiry, S. Torres-Arias, L. Simon, and J. C. Davis, "Ztd _{$JAVA$}: Mitigating software supply chain vulnerabilities via zero-trust dependencies," *arXiv preprint arXiv:2310.14117*, 2023.

[34] C. J. Alberts, A. J. Dorofee, R. Creel, R. J. Ellison, and C. Woody, "A systemic approach for assessing software supply-chain risk," in *2011 44th Hawaii International Conference on System Sciences*. IEEE, 2011, pp. 1–8.

[35] B. Hammi and S. Zeadally, "Software supply-chain security: Issues and countermeasures," *Computer*, vol. 56, no. 7, pp. 54–66, 2023.

[36] "aait-store-cut-part-016 - PyPI," https://pypi.org/project/aait-store-cut-part-016/, 2024.

[37] "square-0-5 - PyPI," https://pypi.org/project/square-0-5/, 2024.

[38] Z. Yao, "Formal trust and threat modeling using large language models," in *2024 Annual Computer Security Applications Conference Workshops (ACSAC Workshops)*. IEEE, 2024, pp. 232–239.

[39] "PyPI Simple Index," https://pypi.org/simple/, 2025.

[40] "PEP 503 – Simple Repository API," https://peps.python.org/pep-0503/, 2015.

[41] "johnnydep - PyPI," https://pypi.org/project/johnnydep/, 2024.

[42] "National Vulnerability Database," https://www.nist.gov/programs-projects/national-vulnerability-database-nvd, 2017.

[43] "GitHub - Advisory database for Python packages published on pypi.org," https://github.com/pypa/advisory-database, 2025.

[44] "MITRE - Common Vulnerabilities and Exposures," https://cve.mitre.org, 2024.

[45] "The 'sklearn' PyPI package is deprecated," https://github.com/facebookresearch/CrypTen/issues/512, 2024.

[46] "ijson - PyPI," https://pypi.org/project/ijson/, 2024.

[47] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2023.

[48] "GitHub - OSV-SCALIBR: A library for Software Composition Analysis," https://github.com/google/osv-scalibr, 2025.

[49] B. Chess and G. McGraw, "Static analysis for security," *IEEE security & privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[50] S. M. Seyed Talebi, Z. Yao, A. Amiri Sani, Z. Qian, and D. Austin, "Undo Workarounds for Kernel Bugs," in *Proc. USENIX Security Symposium*, 2021.

[51] J. Ruohonen, K. Hjerppe, and K. Rindell, "A large-scale security-oriented static analysis of python packages in pypi," in *2021 18th International Conference on Privacy, Security and Trust (PST)*. IEEE, 2021, pp. 1–10.

[52] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1974–1993.

[53] H. Peng, Z. Yao, A. Amiri Sani, D. J. Tian, and M. Payer, "GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation," in *Proc. USENIX Security Symposium*, 2023.

[54] W. Li, H. Yang, X. Luo, L. Cheng, and H. Cai, "Pyrtfuzz: Detecting bugs in python runtimes via two-level collaborative fuzzing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1645–1659.