Bloat beneath Python's Scales: A Fine-Grained Inter-Project Dependency Analysis

ANONYMOUS AUTHOR(S)

Modern programming languages promote software reuse via package managers that facilitate the integration of inter-dependent software libraries. Software reuse comes with the challenge of *dependency bloat*, which refers to unneeded and excessive code that is incorporated into a project through reused libraries. The presence of bloated dependency code exhibits security risks and maintenance costs, increases storage requirements, and slows down application load times. In this work, we conduct a large-scale, fine-grained analysis for understanding bloated dependency code in the PyPI ecosystem. Our analysis is the first to focus on different granularity levels, including bloated dependencies, bloated files, and bloated methods. This allows us to identify the specific parts of a library that contribute to the bloat. To do so, we analyze the source code of 1,302 popular Python projects and their 3,232 transitive dependencies. For each project, we employ a state-of-the-art static analyzer and incrementally construct the *fine-grained project dependency graph (FPDG)*, a representation that captures all inter-project dependencies at method-level.

Our reachability analysis on the FPDG enables the assessment of bloated dependency code in terms of several aspects, including its prevalence in the PyPI ecosystem, its impact on software security, its root causes, and developer perception. Our key finding suggests that the PyPI ecosystem exhibits significant resource underutilization: nearly 50% of dependencies are bloated. This rate gets worse when considering bloated dependency code at a more subtle level, such as bloated files and bloated methods. Our fine-grained analysis also indicates that there are numerous vulnerabilities that reside in bloated areas of utilized packages (15% of the defects existing in PyPI). Other major observations suggest that bloated code primarily stems from omissions during code refactoring processes and that developers are willing to de-bloat their code (22 out of 36 contacted project teams de-bloated a total of 23 dependencies). We believe that our findings can help researchers and practitioners come up with new de-bloating techniques and development practices to detect and avoid bloated code, ensuring that dependency resources are utilized efficiently.

1 INTRODUCTION

Over the past decade, software reuse [Krueger 1992; Mohagheghi and Conradi 2007; Naur and Randell 1969] has experienced a significant rise, mainly due to the widespread adoption of open-source software and package managers. Open-source software libraries are hosted in centralized repositories and are made accessible through package managers, such as Python's `pip`, Java's Maven, or JavaScript's `npm` [Cox 2019; Spinellis 2012]. Developers specify project dependencies in a textual file, allowing package managers to automate the process of fetching the required library versions from the repositories [Boldi and Gousios 2020]. While software reuse offers the benefits of reduced development and maintenance costs [Mohagheghi and Conradi 2007], it also introduces security and reliability risks [Bavota et al. 2015; Cox 2019; Gkortzis et al. 2019; Salza et al. 2020], or license compatibility issues among dependencies [German et al. 2010; van der Burg et al. 2014].

This study focuses on an emerging challenge originating from code reuse, that is, the presence of *bloated dependency code*. Introduced in 2021 by Soto-Valero et al. [2021b], this concept refers to the incorporation of unused code into a software project via dependencies (reused libraries). Research indicates that bloated dependencies introduce (1) dependency conflicts [Patra et al. 2018; Wang et al. 2020], (2) increased maintenance effort [Jafari et al. 2022; Soto-Valero et al. 2021b], and (3) potentially exploitable vulnerable code [Gkortzis et al. 2019]. The security implications of bloated dependency code are particularly noteworthy. Even if some vulnerable dependency code is unreachable, it can still increase the application's attack surface, i.e, attackers may invoke the corresponding code via arbitrary code execution [cve 2022, 2023; OWASP 2020; Ponta et al. 2021].

To illustrate the negative effects of bloated dependency code, consider the case of a Python project named `Zope`. This project declares an unused dependency, which breaks the corresponding

build process due to incompatibility issues with other declared dependencies. To fix this issue, the developers of Zope simply remove the bloated dependency from their codebase [Gmach 2021].

In spite of its importance, bloated dependency code has been only touched by a few studies. A handful of them have addressed bloated dependencies in various software ecosystems such as Maven [Ponta et al. 2021; Soto-Valero et al. 2021a] and Rust's Cargo [Hejderup et al. 2022], with others exploring dependency-related issues, including bloated dependencies in Python [Cao et al. 2023] and JavaScript [Jafari et al. 2022]. Surprisingly, all those studies quantify the prevalence of bloat *only* at package level (e.g. number of bloated dependencies). This can result in false estimation of bloated dependency code, as the actual code reuse occurs at the level of methods [Boldi and Gousios 2020] or functions. For example, a programmer might import a certain dependency, but not actually invoke any of its code. At the same time, reports show that 95% of vulnerable code comes from transitive dependencies [Endor Labs 2023]. All the above considerations demand a more fine-grained analysis for quantifying bloated dependency code. This analysis should not only operate at the method level, but also delve deep into the dependency tree of a project.

In this work, we perform the first *fine-grained inter-project dependency analysis* of bloated dependency code in Python projects. Python, being one of the most popular programming languages [Cosentino et al. 2017; GitHub 2022], has experienced a 22.5% growth in 2022 according to GitHub's annual statistics [GitHub 2022]. Additionally, Python Package Index (PyPI) facilitates one of the largest software ecosystems, hosting more than 450k projects and their 9M Python releases. The extensive software reuse in PyPI emphasizes the importance of studying and addressing dependency bloat in this ecosystem. Our study seeks answers to the following research questions.

RQ1 (Prevalence) How prevalent is bloated dependency code in the PyPI ecosystem? What are the qualitative characteristics of bloated dependency code at different granularities (i.e., package, file, method)? (Section 3.1)

RQ2 (Security) What is the impact of bloated dependency code on software security? Do vulnerabilities reside in bloated code regions? (Section 3.2)

RQ3 (Causes) What are the main causes of bloated PyPI dependencies? What is the frequency of these causes? (Section 3.3)

RQ4 (Developer perception) To what extent are developers willing to remove bloated PyPI dependencies? Does the cause of bloated code affect developer decision and responsiveness? (Section 3.4)

To answer these questions, we design and implement a large scale, fine-grained analysis on popular Python projects. We select 1,302 GitHub Python projects from a well-established dataset [Alfadel M 2020]. For each project in our dataset, we build the *fine-grained project dependency graph (FPDG)*, which captures the *entire* dependency network of a project at the level of methods. To do so, we fetch the source code of every project along with the source code of its dependencies (including direct and transitive ones). Following the approach of Keshani [2021] for practical and large-scale analyses, we then employ PyCG [Salis et al. 2021], the state-of-the-art static analyzer for Python, to construct the (partial) call graph of every Python project and dependency. Through a process called *stitching*, we merge all partial call graphs and derive the final FPDG. Stitching essentially connects an external method in a partial call graph (client code) with its counterpart found in the dependent call graph (library code).

Running standard reachability analyses on the resulting FPDGs allows the assessment of bloated dependency code. We evaluate the identified bloated code in terms of several aspects, including its prevalence, its impact on software security, its main causes, and developer perception.

Contributions: Our work makes the following contributions.

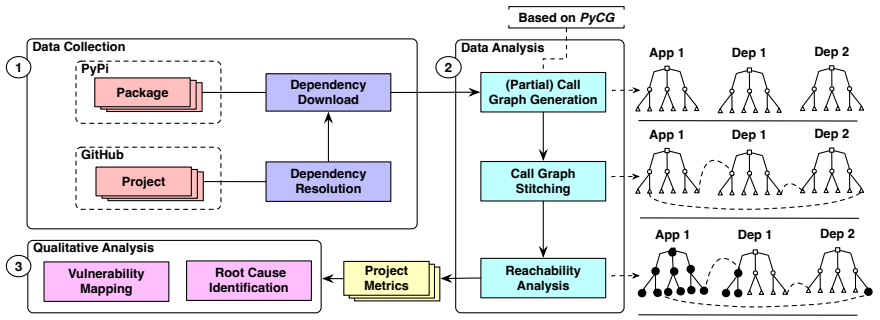


Fig. 1. The overview of our approach for studying bloated code in PyPI ecosystem.

- We conduct the first large-scale analysis for constructing fine-grained dependency networks in the PyPI ecosystem. We provide a corresponding reference dataset containing dependency networks that come from 1,302 Python projects and 3,232 PyPI releases (Section 2).
- We quantify bloat in the PyPI ecosystem, and provide a thorough assessment of its prevalence, its security impact, its main causes, and developer perception (Section 3).
- We enumerate the implications of our findings, and discuss potential future directions in software dependency management (Section 4).

Summary of findings: We find that the PyPI ecosystem exhibits severe dependency under-utilization (RQ1). On average, a Python project consists of ten bloated dependencies. Bloated dependency code becomes more prevalent when measuring bloat at the level of files or methods. In particular, 87% of the dependency source files and 96% of the external methods in a Python project are bloated. Further, our fine-grained analysis highlights the security concerns related to bloated dependency code (RQ2): We identify several vulnerabilities in bloated areas of utilized packages (15% of the total defects in PyPI). Such defects act as "time-bombs" that can get activated after a slight code change. Furthermore, bloated dependencies are mainly introduced by pervasive changes in developers' code base (e.g., feature removals) (RQ3). Finally, developers tend to remove bloated dependencies, especially when they are aware of their main causes (RQ4).

Implications: Through our analysis, we have identified and reported 42 bloated dependencies found in 36 Python projects. The development teams of 22 projects have already fixed 23 bloated dependencies, demonstrating the practicality of our work. We believe that our study can help researchers and practitioners to build appropriate linters and automated refactoring tools to detect and eliminate bloated dependency code in Python projects.

2 METHODOLOGY

Figure 1 summarizes our approach for studying bloated code within the PyPI ecosystem. As a starting point (①), we select a set of popular and well-established Python projects based on a specific criteria (Section 2.2). Then, for each selected project, we resolve its direct and transitive dependencies (Section 2.2), and download the source code of each dependency for further analysis.

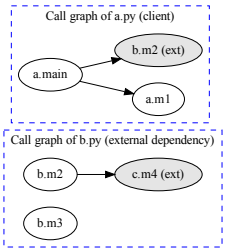
The *data analysis* step (②) begins with the construction of the partial (stand-alone) call graph of every individual project using the state-of-the-art static analysis tool PyCG [Salis et al. 2021]. Next, we merge all partial call graphs through a process we call stitching (Section 2.3.2). Stitching connects the external calls of each partial call graph with the corresponding nodes that appear in the call graphs of the callee dependencies. We call the resulting (stitched) call graph, the *fine-grained project dependency graph (FPDG)* of a project. The FPDG captures the caller-callee relationships within a project's entire dependency tree. Following the construction of the FPDG, we perform a reachability analysis to identify the packages, files, or methods that are accessible within each

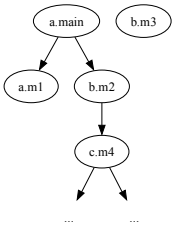
```python
# a.py (client)
from b import m2

def main():
  m1()
  m2()
def m1():
  pass

# b.py (external dependency)
from c import m4

def m2():
  m4()
def m3():
  pass
```





(a) Python program (b) Partial call graphs (c) FPDG graph

Fig. 2. An example demonstrating a Python program with its direct and transitive dependencies (a), the corresponding partial call graphs (b), the resulting fine-grained project dependency graph (c). A gray node in a partial call graph represents an external method whose definition can be found in the dependency.

application. All unreachable methods are marked as *bloated*, and serve the basis for detecting bloated files and bloated dependencies according to a set of formal definitions (Section 2.1).

In addition to the quantitative analysis, we also perform a qualitative analysis (③) on our bloated code results. The goal of our qualitative analysis is twofold. First, we identify the root causes of code bloat. Second, we map previously-reported Python vulnerabilities taken from GitHub's advisory database [gad 2023] to the individual methods these vulnerabilities come from, and see whether the vulnerabilities involve bloated dependency code.

2.1 Preliminary Definitions

Before we describe in detail each step of our approach, we define (1) our representation for capturing inter-project dependencies, and (2) bloated dependency code.

Fine-Grained Project Dependency Graph: A fine-grained project dependency graph (FPDG) captures the method-level relationships of a Python project across its different dependencies. Formally, we define an FPDG as $FPDG = (V, E)$, where a node $v \in V = Method$ corresponds to a method defined in a project or a dependent PyPI release (external dependency). Each method $\langle m, n \rangle \in Method$ is described by a tuple, where m is the method name and n is the namespace where the method is defined. For example, consider a Python module file named `mymodule.py` with the following contents:

```python
class A:
  def m1(x):
    def m2(y):
      pass
    pass
```

The namespace of method `m1` is `mymodule.A`, while the namespace of method `m2` is `mymodule.A.m1`. Since Python does not support method overloading (i.e., having multiple methods of the same name within the same scope), the combination of namespaces and method names is sufficient to uniquely identify each method.

Finally, an edge $e \in E \subseteq V \times V$ in an FPDG represents a caller-callee relationship. For example, the edge (m_1, m_2) indicates that there is a direct method call from method m_1 to method m_2: a method call or a constructor call. A complete example of a Python project, and its corresponding FPDG is shown in Figure 2.

Bloated dependency code: We focus on *unused* code originating from third-party libraries integrated into a Python's application ecosystem. This unused code stems from two major sources: (1) *direct* dependencies, explicitly declared by the developers, or (2) *transitive* dependencies, automatically resolved by the package manager (e.g., `pip`). We now formalize the notion of software bloat in project dependencies at different granularity levels.

In our setting, we represent each project as a triple of the form $\langle n, F, D\rangle \in Project = ProjectName \times \mathcal{P}(File) \times \mathcal{P}(Project)$. Essentially, a project consists of its name, a set of its source files, and another set that contains its direct project dependencies. Each file is a pair $\langle f, M\rangle \in File = FileName \times \mathcal{P}(Method)$. The first element of the pair corresponds to the file name, while the second element stands for the set of methods defined in the source file. For what follows, the function $Paths(g, s, t)$ returns the set of paths that appear in a graph g, between the source node s and the target node t. Function $Methods : Project \longrightarrow \mathcal{P}(Methods)$ gives the set of all *internal* methods included in a project $p \in Project$ as follows:

$$Methods(\langle n, F, D\rangle) = \bigcup_{(f,M)\in F} M$$

Function $Deps : Project \longrightarrow (Project)$ gives all direct and transitive dependencies of a project:

$$Deps(\langle n, F, D\rangle) = D \cup (\bigcup_{d\in D} Deps(d))$$

Definition 2.1 (Bloated Method). Consider a project $p = (n, F, D) \in Project$, an FPDG g, and a project dependency $d \in Deps(p)$. A method $m \in Methods(d)$ is considered a *bloated method (BM)*, if $\forall\, m' \in Methods(p).\ Paths(g, m', m) = \emptyset$.

This definition says that a method m of a dependency is marked as bloated, when m is *not* reachable by any method defined in the current project.

Definition 2.2 (Bloated File). Consider a project $p = (n, F, D) \in Project$, an FPDG g, and a project dependency $(n', F', D') \in Deps(p)$. A file $(f, M) \in F'$ is considered a *bloated file (BF)*, when $\forall m' \in M.\ m'$ is a bloated method.

In essence, a dependency involves a bloated file f when none of the methods defined in f are used by the project.

Definition 2.3 (Bloated Dependency). Consider an application $p = (n, F, D) \in Project$ and an FPDG g. A dependency $(n', F', D') \in Deps(p)$ is considered a *bloated dependency (BD)*, when $\forall f \in F'.\ f$ is a bloated file.

According to the above definition, a dependency d is marked as bloated, when the current project does not invoke any code included in the source files of d. Based on the aforementioned definitions, we now explain each step of our approach for detecting and studying software bloat..

2.2 Project Selection and Dependency Resolution

Selecting Python projects: Our study focuses on the impact of bloated dependency code from the perspective of an *end-user*. Thus, our dataset contains actual user applications. Specifically, we chose to use a dataset of Python GitHub projects provided by the work of Alfadel et al. [2023]. The dataset [Alfadel M 2020] includes 2,224 Python projects, all of which use PyPI for their dependency management, aligning directly with the focus of our research. Each project in the dataset has at least 10 stars, indicating community interest [Dabic et al. 2021]. In addition, all projects (1) have a minimum of 100 commits, (2) involve at least two contributors, and (3) demonstrate recent activity, with the latest commit pushed after June 1, 2021. Furthermore, all projects are original, i.e., they are

Table 1. The evolution of our initial dataset [Alfadel M 2020] after applying each step of our data collection and data analysis approach. The final dataset consists of 1,302 Python GitHub projects. We have resolved 21,787 dependencies in total corresponding to 3,232 unique PyPI releases. Each GitHub project contains 17 dependencies on average.

Step	Operation	Total GitHub projects	Resolved deps	PyPI releases	Avg. deps
Data Collection	Initial dataset of Python GitHub projects	2,224	-	-	-
	Filtering inaccessible projects	2,215	-	-	-
	Dependency resolution	1,644	34,821	5,611	21
Data Analysis	Partial call graph construction	1,302	21,787	3,232	17
	Call graph stitching	1,302	21,787	3,232	17
	Reachability analysis	1,302	21,787	3,232	17

not forked from other projects. Notably, our selection criteria are consistent with standards utilized in other studies [Abdalkareem et al. 2020; Chowdhury et al. 2022; Kalliamvakou et al. 2014].

We have been able to successfully download the source code of all GitHub projects, with the exception of nine projects that have been either deleted or (potentially) set to private on GitHub. Consequently, we have access to the source code of 2,215 GitHub projects (Table 1).

Resolving project dependencies: For each downloaded project, we look for standardized configuration files [Cannon et al. 2016; Python Packaging Authority 2023] that control the installation of Python dependencies. This includes the traditional setup scripts (`setup.py`), the `pyproject.toml` files, or the `requirements.txt` file. When we detect at least one of the three files, we employ `pip`, which is the official package installer for Python, to install dependencies in a fresh and isolated environment. To do so, for every individual GitHub project, we run `pip install` in a new virtual environment created through the `virtualenv` utility.

Based on the aforementioned steps, we successfully resolve a set of dependencies and obtaine their corresponding releases (uniquely identified by the `package_name:package_version` pattern) for 1,644 out of the 2,215 projects. The failed scenarios (571 in total) can be attributed to two main factors. First, some projects do not contain any configuration files for dependency resolution. Therefore, we are unable to accurately determine the corresponding dependencies. Second, during the installation process, we encountered some unexpected errors due to either dependency conflicts or missing dependencies that are not explicitly specified. Projects without resolved dependencies, are excluded from our analysis. As a final step, we download the source code of these releases using the `pip download` command.

2.3 Construction of Fine-Grained Project Dependency Graph

Our dependency graph generation method is inspired by the practical approach proposed by Keshani [2021]. This proposed method addresses the scalability challenges of ecosystem-wide analyses by generating call graphs in an incremental manner. Specifically, unlike other traditional methods that employ whole-program analyses, the method of Keshani involves the construction of partial call graphs for individual projects or dependencies (Section 2.3.1). Through a stitching process (Section 2.3.2), these partial call graphs are combined together to derive a universal call graph, that is, FPDG. (Section 2.1). We now present the technical details behind the construction of the FPDG.

2.3.1 Partial Call Graph Construction. The process of constructing an FPDG starts with the generation of partial call graphs for every Python project and its dependencies. We use PyCG [Salis et al. 2021], the state-of-the-art static analyzer for Python. The input of PyCG is a set of Python source files, while its output is a call graph represented in a JSON format. Given a Python project $p = \langle n, F, D \rangle \in \mathit{Project}$, we feed the set of source files F to PyCG and store the resulting call graph. Then, we do the same for the source code of every individual dependency of project p, that is $d \in D$.

PyCG has succeeded in generating the partial call graph for 1,502 out of the 1,644 GitHub projects included in our dataset. The remaining 142 projects are excluded from our analysis because they are

Algorithm 1: Stitching Algorithm

```
1  fun stitch(callGraphs)=
2      g ← ∅
3      for cg ∈ callGraphs do
4          for v ∈ nodes(cg) do
5              addNode(g, v)
6          for (s, t) ∈ edges(cg) do
7              if isExternal(t) then
8                  cg' ← get call graph of dependency based on t
9                  t ← resolveExternalNode(t, cg')
10             if t ≠ nil then addEdge(g, s, t)
11     return g
```

written in Python 2, which PyCG does not support. The 1,502 projects processed by PyCG include a total of 4,968 unique PyPI releases as dependencies. PyCG managed to generate the partial call graph for 4,789 releases, while the remaining 175 releases are written in Python 2.

Based on the PyCG results, we perform an extra filtering step to identify all projects and releases that depend on at least one release for which PyCG produces no results. This leads to the exclusion of 196 additional Python projects from our dataset. This filtering procedure is an important step, as it filters out projects with incomplete analysis results. Incomplete analysis results can introduce a substantial number of false positives and false negatives in the corresponding call graphs. Overall, we were able to build the partial call graph for 1,302 projects and 3,232 PyPI releases. On average, each project contains 17 resolved dependencies (see Table 1).

Example: Figure 2a shows an example Python module named `a` that invokes a number of methods that are both internal and external. Figure 2b demonstrates the partial call graphs of module `a` and its dependency. External methods are denoted with gray nodes. External methods have *no* outgoing edges, because the source code of these methods is not available to PyCG.

2.3.2 Stitching of Call Graphs. Having generated the partial call graph of every project and dependency, we develop a method, which we call *stitching*, for combining individual partial call graphs into an FPDG. The key idea behind our method is to merge a list of call graphs by connecting an external node (method) with its counterpart found in the partial call graph of the dependency. To do so, we have developed a tool named `pycg-stitch`, which accepts a list of partial call graphs as input and follows the stitching procedure as outlined in Algorithm 1.

Algorithm 1 iterates over every given call graph and proceeds as follows. Initially, the algorithm adds every orphan node (i.e., a node with no incoming or outgoing edges) of a partial call graph into the FPDG (lines 4, 5). Next, for every edge (s, t) in a given call graph, the algorithm checks whether the edge reaches an external node (method). If this is the case, the algorithm resolves the target method t based on the call graph of the dependency where method t is defined (see method resolveExternalNode, lines 8, 9). When this resolution process ends, the algorithm creates a new edge in the FPDG, consisting of source node s and the resolved method t. Once the algorithm iterates over all the given call graphs, it returns graph g (line 11), which stands for the fine-grained project dependency graph of the project.

Example: An example of the stitching procedure is shown on Figure 2b and Figure 2c. Notice how the internal method `b.m2` of the first call graph (dependency `b`) is matched against the external method `b.m2 (ext)` of the second call graph (project `a`). The algorithm merges these nodes and adds all the incoming edges of `b.m2 (ext)` to node `b.m2` (see Figure 2c).

Resolving external nodes: External calls represent interactions between a node in the current graph and a node in another partial call graph. However, our stitching procedure faces two major

```
1  # modA.py (client code)
2  from ext import B
3  def main():
4    obj = B()
5    obj.m()
6  ...
7  # ext/module.py
8  class B:
9    def m():
10     pass
11 class C:
12   def m():
13     pass
14 # ext/__init__.py
15 from b.module import *
```

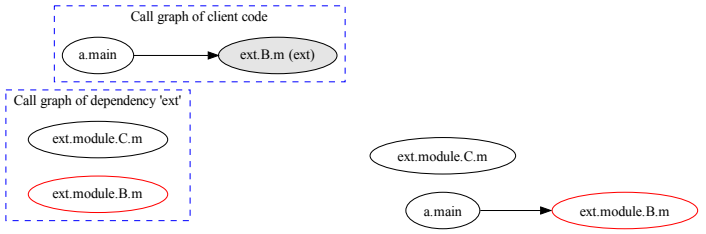


Fig. 3. An example Python program where the access path of an external method is different from its definition namespace (see external method `ext.module.B.m`). Our stitching process matches the access path at the call site (`ext.B.m`) with the corresponding definition node in the dependent call graph (`ext.module.B.m`).

Algorithm 2: Resolving external methods

```
1 fun resolveExternalNode(v, cg)=
2   if v ∈ cg then return v
3   (n, m) ← v
4   cands ← ∅
5   for (n′, m′) ∈ nodes do
6     if m′ = m then
7       d ← levenshteinDistance(n, n′)
8       cands ∪ {(d, (n′, m′))}
9   return Find node with the smallest Levenshtein distance in cands
```

challenges associated with the resolution of external calls. First, the official distribution name of a package might be different from its import name. For example, the PyPI package `beautifulsoup4` is imported as `bs4` in the code. To tackle this issue, Algorithm 1 (line 8) consults the `top_level.txt` file that accompanies every package download. This file outlines the top-level directories or modules included in every package. Based on this information, we construct a dictionary that maps every module to the package it belongs to. For example, when the algorithm encounters an external call of the form `bs4.parse_html`, the algorithm is able to map the module `bs4` to the dependency `beautifulsoup4`, and retrieve its corresponding partial call graph.

The second challenge lies in correctly associating method identifiers with their corresponding internal nodes in the dependent partial call graphs. This is because of the Python's import system, where an external callable can be imported and accessed in multiple ways, depending on the contents of Python `__init__.py` files. Notably, an `__init__.py` file contains code that is executed once a specific package is imported.

To illustrate this, consider the code fragment in Figure 3. The code imports a class `B` from an external dependency named `ext`. In turn, it creates an object of `B` and calls the instance method `m` (lines 4, 5). At use site, the method `m` is accessed through the namespace (`ext.B.m`), which is different from the namespace where the method is defined in the external dependency (i.e., `ext.module.B.m`, line 9). This is because of the contents of file `__init__.py`, which imports all contents of module `ext.module.py` beforehand (line 13). Since PyCG does not have access to the source code of the dependency `ext`, it becomes impossible for us to determine the namespace where method `m` is actually defined. To address the aforementioned issue, the function resolveExternalNode (Algorithm 1, line 9) takes an external node t, and a call graph cg, and tries to match the node t with a corresponding internal node found in cg.

The body of `resolveExternalNode` is shown in Algorithm 2, which we describe using the example of Figure 3. In this example, we want to resolve node `ext.B.m` found in the call graph of the client module `modA.py`. The algorithm first checks whether this node appears in the call graph of the dependency. Since this is not the case, the algorithm iterates over the given call graph and identifies a set of candidate nodes that can potentially match with the input node `ext.B.m`. Specifically, each element in this candidate set represents a method in *cg* whose name is the same as the input node (see lines 5, 6). In our example, the candidate set is {`ext.module.C.m`, `ext.module.B.m`}. The algorithm then picks and returns the element in the candidate set whose namespace is the most similar to that of the input node (i.e., `ext.B`). To do so, our algorithm leverages the Levenshtein [1966] metric, which is the distance between two sequences. Our intuition is that the namespace of a callable at use-site is typically very similar to the namespace of method at its definition site (e.g., see `ext.B` vs. `ext.module.B`).

Back to our example, the input node `ext.B.m` finally resolves to `ext.module.B.m`. Among candidate nodes, namely `ext.module.B.m` and `ext.module.C.m`, the namespace `ext.module.B` is the closest one to the namespace of the callable `ext.B.m`. When resolveExternalNode returns, our stitching procedure finally proceeds with the creation of the FPDG as explained in Algorithm 1.

2.3.3 Reachability Analysis. Once we generate the FPDG for every project, we perform a standard reachability analysis through a Breadth-First Search (BFS) algorithm to compute the set of reachable methods in every project dependency. These reachable methods are directly or transitively called by the project. Based on the results of our reachability analysis, we identify the set of bloated methods, bloated files, and bloated dependencies according to definitions 2.1, 2.2, and 2.1 respectively.

2.4 Analyzing Reachability Results

We now explain how we analyze our reachability results to answer each of our research questions, and better understand bloated dependency code and its implications.

RQ1: Prevalence of software bloat: To quantify the prevalence of code bloat (RQ1), we measure the number and the size (in terms of lines of code — LoC) of bloated methods, bloated files, and bloated dependencies respectively. To do so, we identify every bloated method and measure its code size using metadata provided by PyCG. Regarding the size of bloated files and bloated dependencies, we aggregate the size of every enclosing bloated method.

RQ2: Impact of software bloat on security: We examine the GitHub Advisory Database [gad 2023] and collect all reviewed vulnerabilities reported on PyPI packages. We exclude all vulnerabilities marked with the *"withdrawn"* label, which typically signs that the vulnerability is duplicate or invalid. Through this process, we have collected 1,930 unique PyPI vulnerabilities. Next, we extract the package versions that are affected by every vulnerability, and create a set V, containing all vulnerable PyPI releases. We have identified which of the PyPI releases (i.e., dependencies) of our dataset are included in V. In total, our dataset contains 76 vulnerable PyPI releases. We then leverage our reachability analysis findings to detect how many GitHub projects within our dataset depend on at least one of these 76 vulnerable releases. As a final step, to make our analysis more fine-grained, we also perform a vulnerability mapping: we manually examine all CVEs associated with our vulnerable dependencies, and identify which specific method in the dependency is vulnerable. Again, our reachability analysis results shed light on whether a Python project reaches at least one vulnerable method, or the vulnerability resides in a bloated method (Definition 2.1).

RQ3: Root cause analysis: In this question, we aim to study the main causes of bloated dependencies. Since the manual analysis of each bloated dependency is costly, it is infeasible for us to study each bloated dependency in the population. Therefore, we take a random sample of 50 bloated dependencies that stem from the outcome of our reachability analysis according

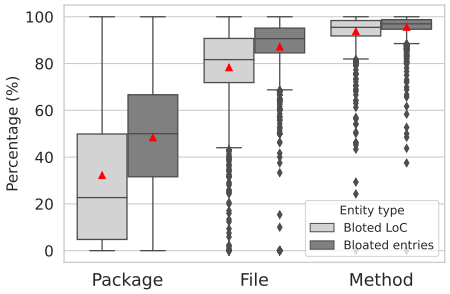


Fig. 4. The distribution of bloat metrics per granularity. Each entry indicates the percentage of bloated entities (e.g., files, methods), and the size of bloated dependency code compared to the overall LoC.

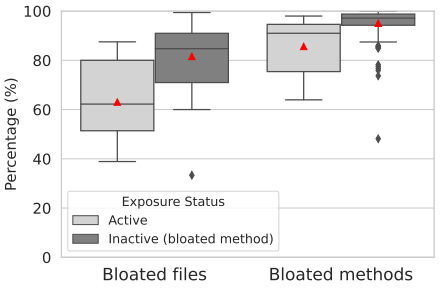


Fig. 5. The distribution of bloat metrics per vulnerability exposure. Each entry indicates the relationship between a vulnerable package, and a project that uses it. For example, 82% of the files in an inactive vulnerable package are bloated.

to Definition 2.3. All the selected dependencies are *direct* dependencies, meaning the they are explicitly declared in the configuration files (e.g., `setup.py`) of a project. We chose to study the root cause of bloated direct dependencies because developers have typically a better control over direct dependencies. Therefore, it is easier for us to explain the causes of bloated code.

For each bloated dependency, we proceed as follows. We use the `git blame` command to identify the commit where this dependency was introduced. We then examine the message of this introductory commit, and if the commit is linked with a pull request (PR), we also inspect the corresponding PR comments. Examining commit messages and developers' discussion helps us uncover the reasons why the dependency is introduced. Finally, we employ `git` to find the commit when developers stopped using the dependency, and why. Based on this information, one author independently assigns every bloated dependency to a root cause category . The final categorization has been validated by an additional researcher.

RQ4: Developer perception on software bloat: We have provided fixes for 42 out of the 50 bloated dependencies for which we have performed root cause analysis (see RQ3). We have submitted our fixes to developers along with insights originating from our previous root cause analysis. We did not submit any pull request for eight out of the 50 bloated dependencies, because although they are unused, these dependencies constrain the resolved versions of some other transitive dependencies included in the project. Developers employ this pattern for ensuring compatibility with an older library version, or avoiding a vulnerable version (see Section 3.3 for more details). Finally, based on our provided fixes and cause insights, developers respond to our pull requests. We then examine their feedback.

3 RESULTS

We now present the answers of our research questions.

3.1 RQ1: How prevalent is bloated dependency code in the PyPI ecosystem?

Our reachability analysis leverages our definitions on bloated dependency code (Section 2.1) and identifies the set of bloated dependencies, bloated files, and bloated methods. At each granularity level, we compute (1) the number of bloated entities (e.g., files, methods), and (2) the size of bloated entities in terms of lines of code (LoC). Figure 4 presents the quantitative characteristics of bloated dependency code. The red marker in each box plot represents the mean of the distribution.

Package level: We observe that half of the dependencies (48%) in a typical PyPI project are bloated. On average, a Python project contains 10 bloated dependencies. Furthermore, we observe

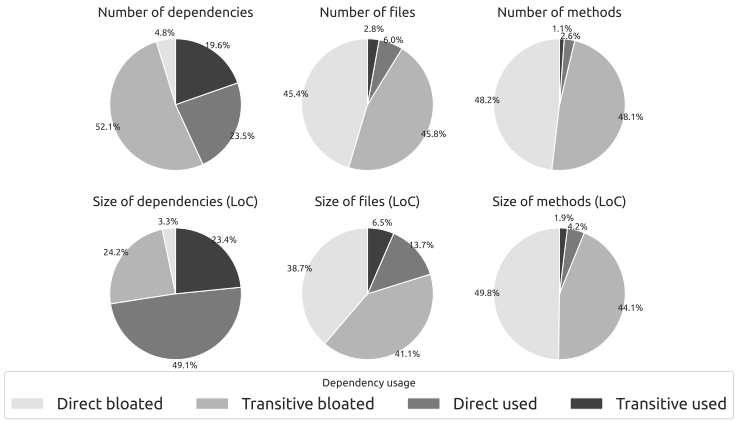


Fig. 6. The distribution of the usage status of dependencies, dependency files, and dependency methods, along with their size in LoC, as aggregated from our dataset.

diverse dependency usage practices: for 25% of the examined projects, at most 32% of project dependencies are bloated. However, for another 25% of the studied projects, this ratio exceeds 67%. Such a high ratio could be attributed to several factors including: (1) developers declaring unused direct dependencies, or (2) projects using only a subset of a direct dependency's functionalities, leading to bloated transitive dependencies.

Regarding lines of code, 32% of the dependency code is bloated, on average. This suggests that while many dependencies might be unused, their code contribution is not as extensive. However, in sheer volume, an average project in our dataset still carries a significant number of bloated lines of code, that is, 94,554 LoC, on average. However, there are still projects in our dataset that diverge from the aforementioned trend. Specifically, approximately 13.6% (177 projects) have optimally managed dependencies with no bloat. On the other hand, 93 projects (~7%) do not use any of their declared dependencies, indicating potential areas for better dependency management.

File level: Moving to bloated files (Figure 4), on average, 87% of the dependency source files are bloated. A Python project contains on average 694 bloated files. When examining the size of these bloated files, the bloated dependency code contains 245,988 LoC, on average. Worse, Figure 4 suggests that over half of the projects are burdened with a bloated LoC rate of more than 91%, meaning that less than 10% of dependency source files are actually used. This emphasizes that while the package-level analysis highlighted unused dependencies as a concern, the file-level analysis indicates that even within seemingly used dependencies a substantial amount of code is unused. This uncovers an additional layer of software bloat that go beyond bloated packages.

Method level: At the method level, our measurements exhibit an even more significant increase of dependency code bloat. Specifically, within a Python project, 96% of its external methods (i.e., 10,652 methods) are bloated, on average. Considering the size of these external methods, we find a similar picture: 94% of the dependency code within an application is bloated. This translates to 185,469 lines of bloated code per project.

Direct vs. transitive dependencies: For completeness, we have also examined the degree to which direct dependencies contribute to the bloat compared to transitive dependencies. Figure 6 presents the distribution of dependency usage statuses across different granularities. In particular, at the package level, it seems that the bloated dependency code mainly stems from transitive dependencies, as more than half of the project dependencies (52.1%) are transitive and unused. However, when examining LoC, we see that only one quarter (24.2%) of the overall project size

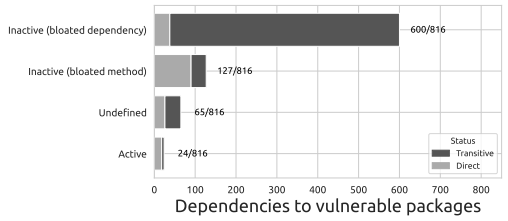


Fig. 7. Number of different usage statuses of vulnerable PyPI dependencies in Github projects.

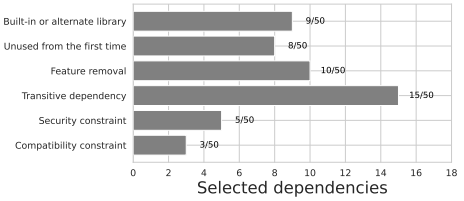


Fig. 8. The distribution of causes leading to bloated-direct PyPI dependencies.

is bloated because of unused transitive dependencies. As granularity refines to file and method levels, an overwhelming majority of dependency code, both in sheer numbers and LoC, is unused. Interestingly, direct and transitive dependencies contribute almost equally to this software bloat.

Overall, as we make our analysis more fine-grained (i.e., from dependencies to files and then to methods), we observe that bloat becomes more prevalent: nearly half of the dependencies in a typical Python project are bloated. The amount of bloated code gets worse at the file and method levels. Our fine-grained analysis helps us uncover which specific code parts contribute to the bloat, and allows for targeted improvements. For example, understanding which specific dependency code is bloated enables developers to refactor or rewrite their code more efficiently.

3.2 RQ2: What is the impact of bloated dependency code on software security?

Our dataset contains 76 vulnerable packages. Our reachability analysis results indicate that out of the 1,302 Python projects of our dataset, 599 projects depend on *at least one* vulnerable package. This translates to 816 unique dependencies vulnerable packages.

Figure 7 illustrates how these 816 vulnerable dependencies are used within the analyzed Python projects. Nearly three quarter of the vulnerabilities (600/816) are discovered in bloated dependencies. This raises an alarming fact, because although these vulnerabilities are not triggered under regular use, they are all latent threats that can be exploited using techniques such as arbitrary code execution [cve 2022, 2023].

Furthermore, we discover that in 15.5% (127 out of 816) of cases, although projects invoke a certain vulnerable package, the specific vulnerable function or class within that package remains unused (see “bloated method in used package”). Our fine-grained analysis captures the distinction between completely unused vulnerabilities and those that are just a step away from activation. This is further exemplified by the fact that among these 127 inactive exposures, 29 of them (23%) appear in *non-bloated* files. This means that when a vulnerable package is in use, a slight code change (e.g., replacing a method call with another one) in the project could trigger the execution of vulnerable code. This further emphasizes the need for developers to stay alert for such “time-bombs”.

Another usage category, accounting for 8% (65/816) consists of vulnerabilities located within non-Python dependency files. Given our Python-centric methodology, the invocation status of such defects remains uncertain. Yet, their existence within the PyPI ecosystem suggests the necessity for broader security evaluations that consider multiple languages and file types. When examining active security threats, we find that a mere 3% (24 out of 816) of the vulnerable functions are invoked by the projects. This means that there is still a non-negligible number of urgent situations, where a Python project is exposed to vulnerabilities that are directly exploitable.

Furthermore, our distinction between direct and transitive dependencies in Figure 7 shows that direct dependencies are mainly responsible for vulnerabilities that appear in invoked methods (i.e., active exposure status) or bloated methods found in used packages. This contradicts with the (1)

"inactive—bloated dependencies" category, and (2) other recent reports [Endor Labs 2023], where vulnerabilities primarily stem from transitive dependencies.

To examine bloated methods and files in vulnerable packages, we need to take into account that several projects may depend upon a certain vulnerable package. That is, in a situation where a project p_1 employs a vulnerable package d, and a project p_2 also uses the same package, bloat metrics may differ depending on the usage. To address this challenge we consider all relationships between a vulnerable package and a project that uses it, and measure the corresponding bloat. The results are illustrated in Figure 5 where we present the distribution of bloated code in terms of files and functions. We focus on two categories, (1) packages that include vulnerable code invoked by the projects (active exposure status), and (2) used packages with unused vulnerable code (inactive exposure status). On average, 63% of the files in an active vulnerable package are bloated. The percentage increases to 86% in the case of methods. In contrast, for inactive vulnerable dependencies, an average of 82% of files and 95% of methods are bloated. Notice that the bloat rates for the dependencies with an active exposure status are still high but not as high as in other cases.

Overall, our results show that a significant number (89%) of vulnerabilities in PyPI dependencies are found within bloated code sections (i.e., either in bloated dependencies or bloated methods).

3.3 RQ3: What are the main causes of bloated PyPI dependencies?

To delve into the root causes of bloated dependency code, we manually inspected 50 bloated dependencies picked at random as explained in Section 2.4. Our manual analysis has resulted in six categories as shown in Figure 8.

Replacement with built-in or alternate library: This root cause arises when developers shift from a third-party library to a built-in or a different library, but neglect to remove the former from their declared dependencies. For example, the `ewels/MultiQC` project replaced `simplejson` library with the Python's internal module `json`, but failed to remove `simplejson` from its dependency list. Such cases account for nine instances in our study.

Feature removal: As software evolves, certain features or code sections may be deprecated or removed. Such removals could render some dependencies redundant. Our investigation identifies ten instances, where some dependencies become bloated, after developers remove certain implementation features from their code base. For example, the `HadrienG/InSilicoSeq` project uses the `future` library to bridge compatibility between Python 2 and Python 3. However, at a later stage, the project decides to drop support for Python 2, removing the corresponding source code and making the dependency to package `future` redundant.

Unused from the first time: We have discovered instances where dependencies are introduced in a project dependency list without any corresponding usage in the source code. The reasons for these introductions often remain unclear. For example, in one case the developers of `PSLmodels/Cost-of-Capital-Calculator` project may have added the `psutil` library for potential benchmarking, but was never used in the code base of `PSLmodels/Cost-of-Capital-Calculator`. We have identified eight bloated dependencies that lie in this category. This highlights that some developers tend to introduce extraneous dependencies, possibly during exploratory development phases. Caution during code reviews and cleanup routines can help prune these unused inclusions.

Redundant declaration of a transitive dependency: When installing a Python project, `pip` resolves all transitive dependencies without requiring them to be declared in project's dependency list. We have observed that 15 projects redundantly list their transitive dependencies as direct. Such a practice possesses risks: for example, consider a package p_1 that stops relying on package p_2. All projects that redundantly depend on both p_1 and p_2 will still retain a dependency to p_2. This can make projects maintain redundant transitive dependencies, and worse face version conflicts

Table 2. The status of our pull requests.

PR status	Number of PRs	Number of bloated dependencies removed
Merged	22	23
Approved (pending merge)	1	1
Positive feedback (pending decision)	2	2
Rejected	1	1
Pending	10	15
Total	**36**	**42**

due to discrepancies between directly and transitively included dependencies. By eliminating such redundancies, projects have exclusive control over the dependencies they directly invoke.

In general, these redundancies can be attributed to misunderstandings about how the dependency resolution process of `pip` works. Specifically, through our interaction with developers (RQ4), we have observed that some development teams are unaware that `pip` also retrieves all of the indirect dependencies of a project.

Security constraint: This category contains cases where developers intentionally constrain the versions of transitive dependencies for security reasons, e.g., to avoid resolving a vulnerable transitive dependency. Since those dependencies are not invoked directly, we consider them direct-bloated. We have run into five such instances. This phenomenon occurs when a dependent library fails to update its own dependencies and mitigate the security risk.

Compatibility constraint: Developers occasionally enforce version constraints on transitive dependencies to ensure compatibility and avoid potential conflicts with certain functionalities or components [Patra et al. 2018; Wang et al. 2020]. In our study, we have detected three such instances. As an example, the `20c/vaping` project constrains the version of package `Werkzeug` between 2.0.0 and 2.1.0. This declaration is accompanied by the following comment:

```
FIXME: werkzeug >2.1.0 breaks static file serving
```

This comment is a sign of a compatibility issue: versions of `Werkzeug` greater than 2.1.0 would break a specific functionality in the project.

Overall, the primary factor contributing to bloated-direct PyPI dependencies is mistakes and omissions during code refactoring (34%). Additionally, another 30% of the bloated dependencies arise from unnecessary listing of transitive dependencies, while 16% comes from dependencies that are introduced without ever being used. Our results uncover several key areas for consideration, including (1) the significance of maintaining updated configuration files after code refactoring, (2) better control over transitive dependencies avoiding the declaration of redundant dependencies, (3) avoiding the declaration of redundant dependencies.

3.4 RQ4: To what extent are developers willing to remove bloated PyPI dependencies?

In this research question, we aim to investigate how developers react to bloated PyPI dependencies. To do so, we have opened pull requests that fix 42 bloated dependencies detected by our reachability analysis (Section 2.4). Table 2 presents the outcomes of these pull requests. Overall, around half of our fixes (22/42) have been accepted and integrated into developers' code base. In one case, developers have approved our fixes, but they have not merged the corresponding pull requests yet. Developers often expressed gratitude for our pull requests, with many highlighting the value of the proposed changes using phrases, such as *"nice catch"* or *"thanks for catching that"*. This positive feedback indicates that developers understand the risks of bloated dependencies, and the importance of removing them. Below, we discuss two interesting cases that come from the our interaction with developers.

Example 1: In project `EasyPost/easypost-python`, we encountered diverse feedback from the development team. A member of the development team initially rejected our fix with the

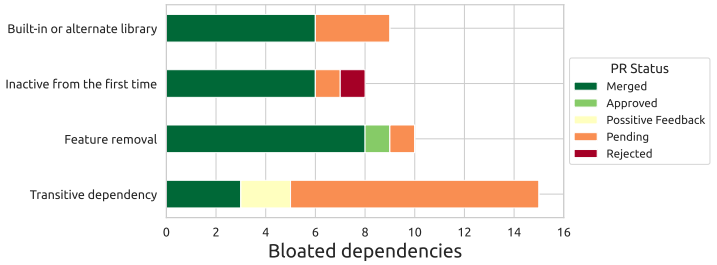


Fig. 9. The distribution of PR statuses per root cause.

following comment: *"If the dependency is unused, that is a mistake as we want to implement typing in this library. Recommend to reject."*. However, the lead developer answered with *"This dependency is unused at this time and has been since its introduction over a year and a half ago. I'm down to remove it. If we ever need it in the future we can easily reintroduce it."* Subsequently, the opposing developer approved and performed the merge.

Example 2: In project `ocrmypdf/OCRmyPDF`, we proposed the removal of the `deprecation` dependency. However, the developer declined the change, pointing out that while `deprecation` is currently unused, they sporadically employ it to mark API sections as deprecated. The developer expressed that the consistent inclusion of `deprecation` simplifies matters, as the developers do not need to notify maintainers for every API change. This example also demonstrates that some developers prioritize potential future utility and maintenance convenience over code simplification.

Figure 9 also shows how the underlying root cause of a bloated dependency affects the status of our pull requests. 20 accepted fixes are linked to direct dependencies that fall into two categories: (1) mistakenly declared dependencies, and (2) dependencies that are no longer used due to code refactoring, such as feature removal. Interestingly, there are only three accepted fixes related to redundant transitive dependencies. This pattern of responses could be attributed to the clarity of root causes. For example, removing unnecessary dependencies is often part of a code cleanup process following pervasive code changes. Unfortunately, developers sometimes overlook this cleanup, which makes them quickly address and resolve issues that stem from such omissions.

4 KEY TAKEAWAYS AND SUGGESTIONS

We now discuss several implications of our work, and how our findings can serve as a basis for future research endeavors in de-bloating the PyPI ecoysystem.

The PyPI ecosystem exhibits considerable resource underutilization. We have shown that a substantial portion of Python dependency code is bloated (Section 3.1). This indicates a need for tailored solutions within the Python domain. Researchers and practitioners should consider the development of Python-specific de-bloating techniques that effectively address the widespread bloat in the PyPI ecosystem.

Dependency code bloat becomes more prevalent via a fine-grained analysis. Figure 4 reveals that the majority of the dependency code bloat can only be found only when examining inter-project dependencies at file- or method-level. Removing unused packages will contribute to reducing bloat to some extent [Soto-Valero et al. 2021b], however a significant amount of bloat hidden within used dependencies will continue to exist. Hence, effective de-bloating techniques should consider file and method bloat within dependencies that are partially used by the developers.

"Time-bombs" in bloated dependency code. Our results indicate that there is a relationship between code bloat and security vulnerabilities in the PyPI ecosystem. A significant 89% of these vulnerabilities are identified within bloated code regions (Figure 7). Interestingly, 15% of them are marked as "time-bombs", i.e., the corresponding code is not used at the moment, but this

may change in the future and trigger the vulnerability. Another layer of complexity is added by a non-trivial number of vulnerabilities (8%) found in non-Python files. Developers and security professionals should proactively address bloat in their software design and maintenance practices. Addressing the identified cross-language vulnerabilities (e.g., via a technique that goes beyond Python source files) could further reduce the attack surface of Python projects.

Code refactoring introduces bloated dependencies. Figure 8 shows that bloated dependencies are introduced after extensive modifications in the developers' code base, such as feature removal, replacement of algorithms, etc. This suggests that developers might be less careful when tidying up post-feature deprecations or refactoring. To mitigate this issue, existing linters and automated refactoring tools, such as `pylint` or `flake8`, should be extended with additional functionalities, Such functionalities should check and update (if necessary) the configuration files of a project so that the declared list of dependencies contains only used ones.

Developers are willing to remove bloated dependency code especially when they are aware of its major causes. Our interaction with development teams indicates that developers are willing to de-bloat their code (Section 3.4). This finding contradicts with the findings of a previous study on bloated dependencies [Cao et al. 2023], where the authors found that the majority of Python developers are not willing to remove bloated dependencies. This can be attributed to the fact that the reports of Cao et al. [2023] are not accompanied by a detailed, well-justified description containing the corresponding causes. Based on this observation, future linters and de-bloating tools should clearly enumerate and document the reasons why a certain bloated dependency code should be removed. Finally, our findings suggest that future tools should prioritize warnings based on the causes of bloated code. For example, developers are more likely to accept de-bloating code that stems from code refactoring processes.

5 THREATS TO VALIDITY

Internal validity: One threat to the internal validity of our work is related to the resulting FPDG. Because of the dynamic nature of Python, our static FPDG might contain false negatives and false positives. This is unavoidable even when dealing with statically-typed programming languages, such as Java [Soto-Valero et al. 2021a,b]. We build FPDG using PyCG [Salis et al. 2021], the de-facto static analyzer of Python, which has been employed in other empirical studies [Simon et al. 2023; Venkatesh et al. 2023]. Another threat is the representativeness of the selected Python projects. We chose a carefully crafted dataset [Alfadel et al. 2023; Alfadel M 2020] that contains real-world Python applications from various domains, including web technology (e.g., REST API clients, servers), astronomy, high-performance computing, biomedical tools, robotics, and many others. Each application in the dataset has at least 10 stars, a minimum of 100 commits, involves at least two contributors, and demonstrates recent activity. Finally, another internal threat to validity relates to our stitching process and the resolution of external calls (Section 2.2). First, we consulted the `top_level.txt` file to associate every external module with its package distribution. This is the standard practice employed by other studies [Cao et al. 2023]. Second, we leveraged a well-established metric (the Levenshtein [1966] distance) to compare the similarity between the callable namespace and the definition namespace of an external method. Based on the similarity score, we associated every external method in a call graph with its corresponding node in the dependent call graph. This relies on the assumption that the access path of callable is typically similar to the its namespace at the definition site.

External validity: A threat to the external validity is associated with the generalizability of our findings to other software ecosystems. PyPI is one of the largest software ecosystems, hosting more than 450k projects and their 9M releases. Given its widespread use, a study dedicated to PyPI is essential on its own. Notably, some of our findings (e.g., RQ1—prevalence) are also consistent with

the findings of other empirical studies that target other software ecosystems, such as Maven [Soto-Valero et al. 2021b]. The representativeness of the bloated dependencies selected for answering RQ3 and RQ4 is another threat to the external validity. To mitigate this threat, we picked a random sample of 50 bloated dependencies, which is in line with other studies on bloated dependencies. For example, Cao et al. [2023] have manually analyzed the causes of 127 bloated dependencies. Following the reasoning outlined in the work of Mastrangelo et al. [2019] and Chaliasos et al. [2021], when working with a random sample of 50 bloated dependencies, there is an approximately 8% chance of missing a cause category with a relative frequency of at least 5%. Finally, our manual cause analysis might be subjective. To mitigate this threat, the proposed categorization has been validated by an additional researcher, following the best practices in manual qualitative analyses [Brereton et al. 2007; Chaliasos et al. 2021; Kotti et al. 2023; Paixao et al. 2017].

6 RELATED WORK

Software bloat: Research has focused on reducing the code size of source programs (written in C) by adopting program transformation and syntax-aware techniques [Regehr et al. 2012; Sun et al. 2018], or reinforcement learning [Heo et al. 2018]. Other de-bloating techniques operate on C/C++ binaries to reduce their attack surface [Qian et al. 2019; Quach et al. 2018; Sharif et al. 2018]. In recent years, there has been a growing interest in de-bloating Java applications [Bruce et al. 2020; Jiang et al. 2016; Macias et al. 2020; Soto-Valero et al. 2023, 2021b]. The proposed tools employ static and dynamic analysis that works on Java bytecode. Beyond traditional computer programs, software de-bloating has been applied to diverse domains, including the Chromium web browser [Qian et al. 2020], Android applications [Jiang et al. 2018], Docker containers [Rastogi et al. 2017], web applications written in JavaScript or PHP [Azad et al. 2019; Vázquez et al. 2019], or shared binary libraries [Agadakos et al. 2020]. Surprisingly, despite its popularity, there has not yet been any development of de-bloating techniques for Python.

Bloated dependencies: The concept of "bloated dependencies" is first introduced by Soto-Valero et al. [2021b] in their in-depth study that focuses on the Maven ecosystem. By statically analyzing Java bytecode files, they identify and remove unused dependencies from Maven POM files. Their study reveals a significant 75% bloat rate in dependency relationships, primarily stemming from transitive dependencies. In comparison, our findings indicate a bloated dependency rate of only 48%, probably indicating a better dependency utilization in the PyPI Ecosystem. In their subsequent study [Soto-Valero et al. 2021a], the authors find that the vast majority (89%) of bloated dependencies in Maven increase over time, with the addition of new dependencies identified as the primary root cause. Our qualitative analysis identifies omissions during code refactoring processes as the main origin of bloated direct PyPI dependencies (RQ3). None of these Maven studies are as fine-grained as our work, as they only consider bloated dependency code at package level.

Further extending the exploration of bloated dependencies, Hejderup et al. [2022] leverage static call graphs to generate the call-based dependency network of the Rust Ecosystem. They discover that ~60% of the resolved dependencies within packages are bloated. Jafari et al. [2022] identifies a similar trend in the `npm` ecosystem.

The closest study to our work is the one of Cao et al. [2023], who investigate *"dependency smells"* across 132 Python projects. The term "dependency smells" describes a set of dependency-related issues, including bloated dependencies. To identify bloated dependencies, the authors parse configuration files (e.g., `setup.py`) and analyze import statements in the source code. The authors characterize each bloated dependency by its prevalence, evolution, causes, and developer perceptions. The main findings of this study show that bloated direct dependencies appear in 86% of projects, with 75% of these dependencies coming from new dependency declarations without subsequent source code use. Contrary to this study, our approach to detecting bloated code is more

fine-grained: we quantify bloat at both the file- and method-level. Additionally, our analysis is able to distinguish transitive dependencies from direct ones. This is particularly evident in our root cause analysis. For instance, our fine-grained analysis allows us to know whether a dependency declaration corresponds to a transitive dependency. Without this knowledge, we would have misclassified such cases as *"unused from the first time"*.

Furthermore, Cao et al. [2023] pinpoint that developers tend to be hesitant when it comes to removing bloated dependencies (0/10 reported issues fixed, 4/10 rejected). In contrast, we show that developers are willing to remove unused direct dependencies, even in cases where the dependencies are used transitively. As described in Section 4, this difference could be explained by our well-justified pull requests that incorporate the results of our cause analysis.

Software ecosystem analysis: Software ecosystems are experiencing significant growth over time [Decan et al. 2019; Kikas et al. 2017; Wittern et al. 2016]. Kikas et al. [2017] analyze the ecosystems of JavaScript, Ruby and Rust and find that the removal of a single package can impact around one third of projects within these ecosystems. Decan et al. [2019] perform an empirical comparison of seven software ecosystems. Their results show that software packages tend to be fragile due to the constant increase of transitive dependencies. Highlighting the significance of a fine-grained analysis, Hejderup et al. [2018] introduce the idea of using call graphs to represent actual calling relationships within software ecosystems. Building on this method, Mir et al. [2023] design a fine-grained analysis for the Maven ecosystem, and reveal that while one third of the packages are vulnerable due to transitive dependencies, only 1% of them has a reachable call to a vulnerable dependency method. This is in line with the results of our RQ2.

Working on the PyPI ecosystem, another study [Abdalkareem et al. 2020] shows that the size of 10% of the packages on PyPI is less than 35 LoC. Another empirical analysis on PyPI [Alfadel et al. 2023] comments that the average number of vulnerabilities affecting a PyPI package increases over time, with each vulnerability taking three years to be discovered. Our work provides methods, impetus, and guidance for counter-regressive software changes involving dependencies.

7 CONCLUSION

We have presented the first fine-grained study of bloated dependency code in the PyPI ecosystem. We have constructed large, inter-project dependency graphs that capture the relationship between 1,032 Python projects and 3,232 PyPI releases at the method level. Using standard graph reachability algorithms, we study bloated dependency code at different granularities. We find that the PyPI ecosystem is full of bloat: around half of the dependencies in a typical Python project are entirely unused. Worse, the bloat considerably increases when considering the granularity of files and methods. Further, through our fine-grained analysis, we have identified a number of "time-bombs"(15%), i.e., vulnerable code that is not invoked by the different projects, yet it exists in packages that the projects use. Moreover, bloated dependency code mainly originates from pervasive code changes (e.g., feature removals). Finally, developers are willing to remove their bloated dependencies. Thanks to our analysis, the developers of 22 open-source projects have already fixed 23 issues related to bloated dependencies.

We believe our work can guide the design of future bloat detection and prevention tools for Python. Such tools should effectively address bloat at granularities that go beyond the package level, thereby significantly reducing the attack surface of Python applications. Finally, bloat prevention tools need to provide informative and actionable messages to developers, and prioritize warnings based on the root cause of bloat.

8 DATA AVAILABILITY

We have incorporated the data supporting our results within our submission.

REFERENCES

2022. CVE-2022-24439. https://nvd.nist.gov/vuln/detail/CVE-2022-24439. [Online; accessed 25-September-2023].

2023. CVE-2023-36188. https://nvd.nist.gov/vuln/detail/CVE-2023-36188. [Online; accessed 25-September-2023].

2023. *GitHub Advisory Database*. https://github.com/advisories [Online; accessed 11-September-2023].

Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: an empirical case study on npm and PyPI. *Empirical Software Engineering* 25, 2 (01 Mar 2020), 1168–1204. https://doi.org/10.1007/s10664-019-09792-9

Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2020. Large-Scale Debloating of Binary Shared Libraries. *Digital Threats* 1, 4, Article 19 (dec 2020), 28 pages. https://doi.org/10.1145/3414997

Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in Python packages. *Empirical Software Engineering* 28, 3 (25 Mar 2023), 59. https://doi.org/10.1007/s10664-022-10278-4

Shihab E Alfadel M, Costa DE. 2020. *Empirical Analysis of Security Vulnerabilities in Python Packages*. https://doi.org/10.5281/zenodo.5645517

Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1697–1714. https://www.usenix.org/conference/usenixsecurity19/presentation/azad

Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20, 5 (01 Oct 2015), 1275–1317. https://doi.org/10.1007/s10664-014-9325-9

Paolo Boldi and Georgios Gousios. 2020. Fine-Grained Network Analysis for Modern Software Ecosystems. *ACM Trans. Internet Technol.* 21, 1, Article 1 (dec 2020), 14 pages. https://doi.org/10.1145/3418209

Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from Applying the Systematic Literature Review Process within the Software Engineering Domain. *J. Syst. Softw.* 80, 4 (apr 2007), 571–583. https://doi.org/10.1016/j.jss.2006.07.009

Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-Depth Investigation into Debloating Modern Java Applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/3368089.3409738

Brett Cannon, Nathaniel Smith, and Donald Stufft. 2016. *PEP 518 – Specifying Minimum Build System Requirements for Python Projects*. PEP 518. Python Software Foundation. https://www.python.org/dev/peps/pep-0518/

Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. 2023. Towards Better Dependency Management: A First Look at Dependency Smells in Python Projects. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1741–1765. https://doi.org/10.1109/TSE.2022.3191353

Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 123 (Oct. 2021), 30 pages. https://doi.org/10.1145/3485500

Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. 2022. On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages. *IEEE Transactions on Software Engineering* 48, 8 (2022), 2695–2708. https://doi.org/10.1109/TSE.2021.3068901

Valerio Cosentino, Javier L. Cánovas Izquierdo, and Jordi Cabot. 2017. A Systematic Mapping Study of Software Development With GitHub. *IEEE Access* 5 (2017), 7173–7192. https://doi.org/10.1109/ACCESS.2017.2682323

Russ Cox. 2019. Surviving Software Dependencies. *Commun. ACM* 62, 9 (aug 2019), 36–43. https://doi.org/10.1145/3347446

Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 560–564. https://doi.org/10.1109/MSR52588.2021.00074

Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. *Empirical Software Engineering* 24, 1 (01 Feb 2019), 381–416. https://doi.org/10.1007/s10664-017-9589-y

Endor Labs. 2023. *The state of dependency management*. https://www.endorlabs.com/state-of-dependency-management [Online; accessed 26-September-2023].

Daniel M. German, Massimiliano Di Penta, and Julius Davies. 2010. Understanding and Auditing the Licensing of Open Source Software Distributions. In *2010 IEEE 18th International Conference on Program Comprehension*. 84–93. https://doi.org/10.1109/ICPC.2010.48

GitHub. 2022. The State of the Octoverse: Top Programming Languages 2022. https://octoverse.github.com/2022/top-programming-languages. Online: accessed 24 September 2023.

Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2019. A double-edged sword? Software reuse and potential security vulnerabilities. *Lecture Notes in Computer Science* (2019), 187–203. https://doi.org/10.1007/978-3-030-22888-0_13

Jürgen Gmach. 2021. *Remove unused Sphinx dependency.* https://github.com/zopefoundation/Zope/pull/968 [Online; accessed 26-September-2023].

Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios. 2022. Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering* 27, 5 (30 May 2022), 102. https://doi.org/10.1007/s10664-021-10071-9

Joseph Hejderup, Arie van Deursen, and Georgios Gousios. 2018. Software Ecosystem Call Graph for Dependency Management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results* (Gothenburg, Sweden) *(ICSE-NIER '18)*. Association for Computing Machinery, New York, NY, USA, 101–104. https://doi.org/10.1145/3183399.3183417

Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2022. Dependency Smells in JavaScript Projects. *IEEE Trans. Softw. Eng.* 48, 10 (oct 2022), 3790–3807. https://doi.org/10.1109/TSE.2021.3106247

Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android Application Redundancy Customization Based on Static Analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. 189–199. https://doi.org/10.1109/ISSRE.2018.00029

Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 12–21. https://doi.org/10.1109/COMPSAC.2016.146

Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 92–101. https://doi.org/10.1145/2597073.2597074

Mehdi Keshani. 2021. Scalable Call Graph Constructor for Maven. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 99–101. https://doi.org/10.1109/ICSE-Companion52605.2021.00046

Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and Evolution of Package Dependency Networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 102–112. https://doi.org/10.1109/MSR.2017.55

Zoe Kotti, Rafaila Galanopoulou, and Diomidis Spinellis. 2023. Machine Learning for Software Engineering: A Tertiary Study. *ACM Comput. Surv.* 55, 12, Article 256 (mar 2023), 39 pages. https://doi.org/10.1145/3572905

Charles W. Krueger. 1992. Software Reuse. *ACM Comput. Surv.* 24, 2 (jun 1992), 131–183. https://doi.org/10.1145/130844.130856

Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, Vol. 10. Soviet Union, 707–710.

Konner Macias, Mihir Mathur, Bobby R. Bruce, Tianyi Zhang, and Miryung Kim. 2020. WebJShrink: A Web Service for Debloating Java Bytecode. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1665–1669. https://doi.org/10.1145/3368089.3417934

Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. 2019. Casting about in the Dark: An Empirical Study of Cast Operations in Java Programs. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 158 (Oct. 2019), 31 pages. https://doi.org/10.1145/3360584

Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. 2023. On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 201–211. https://doi.org/10.1109/SANER56733.2023.00028

Parastoo Mohagheghi and Reidar Conradi. 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering* 12, 5 (01 Oct 2007), 471–516. https://doi.org/10.1007/s10664-007-9040-x

Peter Naur and Brian Randell. 1969. Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7th-11th october 1968. (1969).

OWASP. 2020. Deserialization of Untrusted Data. https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data Online; accessed 25 September 2023.

Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2017. Are developers aware of the architectural impact of their changes?. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 95–105. https://doi.org/10.1109/ASE.2017.8115622

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* (Gothenburg, Sweden) *(ICSE '18)*.

Association for Computing Machinery, New York, NY, USA, 741–751. https://doi.org/10.1145/3180155.3180184

Serena Elisa Ponta, Wolfram Fischer, Henrik Plate, and Antonino Sabetta. 2021. The Used, the Bloated, and the Vulnerable: Reducing the Attack Surface of an Industrial Application. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 555–558. https://doi.org/10.1109/ICSME52107.2021.00056

Python Packaging Authority. 2023. Pip v23.1.2 Documentation: Build System Interface. https://pip.pypa.io/en/stable/reference/build-system/# Accessed: July 9, 2023.

Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 461–476. https://doi.org/10.1145/3372297.3417866

Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886. https://www.usenix.org/conference/usenixsecurity18/presentation/quach

Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 476–486. https://doi.org/10.1145/3106237.3106271

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. *SIGPLAN Not.* 47, 6 (jun 2012), 335–346. https://doi.org/10.1145/2345156.2254104

Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

Pasquale Salza, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Filomena Ferrucci. 2020. Third-party libraries in mobile apps. *Empirical Software Engineering* 25, 3 (01 May 2020), 2341–2377. https://doi.org/10.1007/s10664-019-09754-1

Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/3238147.3238160

Sebastian Simon, Nikolay Kolyada, Christopher Akiki, Martin Potthast, Benno Stein, and Norbert Siegmund. 2023. Exploring Hyperparameter Usage and Tuning in Machine Learning Research. In *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*. 68–79. https://doi.org/10.1109/CAIN58948.2023.00016

César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021a. A Longitudinal Analysis of Bloated Java Dependencies *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1021–1031. https://doi.org/10.1145/3468264.3468589

César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. 2023. Coverage-Based Debloating for Java Bytecode. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 38 (apr 2023), 34 pages. https://doi.org/10.1145/3546948

César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021b. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Software Engineering* 26, 3 (25 Mar 2021), 45. https://doi.org/10.1007/s10664-020-09914-8

Diomidis Spinellis. 2012. Package Management Systems. *IEEE Softw.* 29, 2 (mar 2012), 84–86. https://doi.org/10.1109/MS.2012.38

Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 361–371. https://doi.org/10.1145/3180155.3180236

Sander van der Burg, Eelco Dolstra, Shane McIntosh, Julius Davies, Daniel M. German, and Armijn Hemel. 2014. Tracing Software Build Processes to Uncover License Compliance Inconsistencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 731–742. https://doi.org/10.1145/2642937.2643013

Hernán Ceferino Vázquez, Alexandre Bergel, Santiago Vidal, JA Díaz Pace, and Claudia Marcos. 2019. Slimming JavaScript applications: An approach for removing unused functions from JavaScript libraries. *Information and Software Technology* 107 (2019), 18–29. https://doi.org/10.1016/j.infsof.2018.10.009

A. Shivarpatna Venkatesh, J. Wang, L. Li, and E. Bodden. 2023. Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 391–401. https://doi.org/10.1109/SANER56733.2023.00044

Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 125–135. https://doi.org/10.1145/3377811.3380426

Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 351–361. https://doi.org/10.1145/2901739.2901743