LastPyMile: Identifying the Discrepancy between Sources and Packages

Duc-Ly Vu
ducly.vu@unitn.it
University of Trento
Italy

Fabio Massacci
fabio.massacci@ieee.org
University of Trento
Italy
Vrije Universiteit Amsterdam
Netherlands

Ivan Pashchenko
ivan.pashchenko@unitn.it
University of Trento
Italy

Henrik Plate
henrik.plate@sap.com
SAP Security Research
France

Antonino Sabetta
antonino.sabetta@sap.com
SAP Security Research
France

ABSTRACT

Open source packages have source code available on repositories for inspection (e.g. on GitHub) but developers use pre-built packages directly from the package repositories (such as npm for JavaScript, PyPI for Python, or RubyGems for Ruby).

Such convenient practice assumes that there are no discrepancies between source code and packages. These differences pose both operational risks (e.g. making dependent projects unable to compile) and security risks (e.g. deploying malicious code during package installation) in the software supply chain.

Our empirical assessment of 2438 popular packages in PyPI with an analysis of around 10M lines of code shows several differences in the wild: modifications cannot be just attributed to malicious injections. Yet, scanning again all and whole 'most likely good but modified' packages is hard to manage for FOSS downstream users.

We propose a methodology, LastPyMile, for identifying the differences between build artifacts of software packages and the respective source code repository. We show how it can be used to extend current package scanning practices for malware injection (which only covers less than 1% of the code of deployed packages).

CCS CONCEPTS

• **Software and its engineering** → **Software configuration management and version control systems**; • **Security and privacy** → **Software security engineering**;

KEYWORDS

Open source software, software supply chain, Python, PyPI

ACM Reference Format:
Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. LastPyMile: Identifying the Discrepancy between Sources and Packages. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468592

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468592

1 INTRODUCTION

The expression *software supply chain* refers to "anything that goes into or affects your code from development, through your CI/CD pipeline, until it gets deployed into production" [22]. In the past decade, Free and Open-Source Software (FOSS) has become an integral part of the software supply chain: as much as 99% of codebases contain open-source code [43], and 85% [41] to 97% [47] of enterprise codebases comes from open source.

One benefit of FOSS is that source code and additional metadata is publicly available for audit, review, and even modification. Developers rely on this information (e.g., number of GitHub stars, number of downloads from `libraries.io`) to decide whether to add a FOSS project as a software dependency into their projects [25, 35]. Organizations with high security requirements, e.g., government organizations or vendors of commercial enterprise software, commonly establish vetting processes to ensure the quality and security of 3rd party software and services [9, 31]. In the case of FOSS, this evaluation is performed mostly by manual reviews and automated scans of the source code repository of each dependency [10].

In theory, once code is checked, developers could download software dependencies as source files in tarballs, and build them in-house. Yet, this process can be time-consuming and requires knowledge of the build systems [21].

In practice, developers download pre-built packages from repositories (such as npm for JavaScript, PyPI for Python, or RubyGems for Ruby) under *the comfortable assumption that no discrepancies are introduced in the last mile between the source code and their respective packages.* Yet, such discrepancies might be introduced by manual or automated build tools (e.g., metadata, Python bytecode files) [18] or for evil purposes. For example, a backdoor was inserted into the PyPI package `ssh-decorate` to collect the users' SSH credentials and exfiltrate them to a remote server [7].

Reproducible builds could be a solution. For it to be practical, modifications need to be the exception rather than the norm. Unfortunately, the opposite is true on the field. Indeed, in the npm

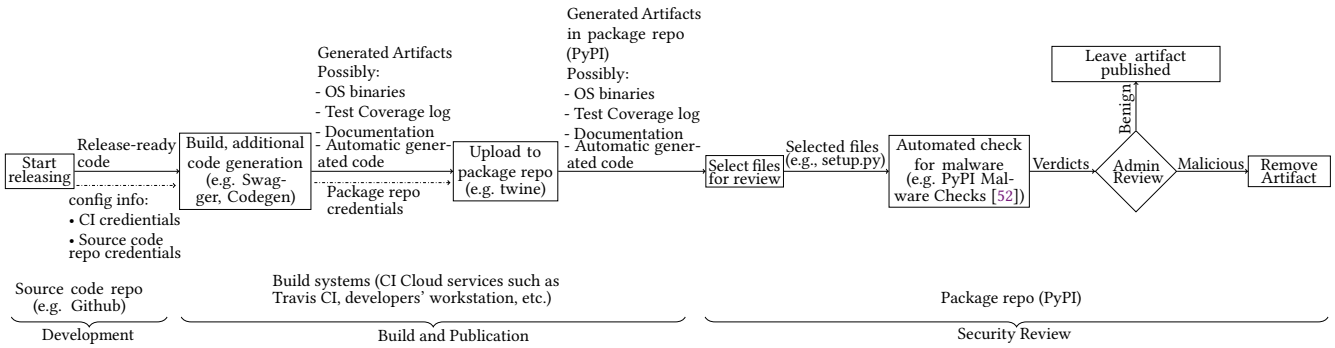


Figure 1: Current Development, Build, Publication and Security Review pipeline of PyPI packages

ecosystem, packages are not easily reproducible from the source code [18]. The same applies to the PyPI ecosystem (see Section 6).

In the absence of reproducible builds, a vetting process must be extended to cover the risk of malicious code injection in the last mile. Since applications have many direct and transitive dependencies, and because every new version has to be verified, scalability and integration with existing security review pipelines are key.

These requirements clash with the resources at hand for FOSS repositories: less than ten PyPI administrators oversee 400 000 package owners. At the time of writing, for every new upload, PyPI's vetting pipeline only checks a script called `setup.py` for malicious code that would execute upon package installation [52]. Although `setup.py` is commonly targeted by attackers, malicious code is also injected other locations. Other approaches also require a significant effort to reduce false positives [13] and to improve the quality of hand-crafted signatures [32]. While suspicious packages or updates might be flagged, too many false alerts are generated for benign packages [29]. In 2020, the administrators had to evaluate 1874 new updates *per day*, with an average of 3500 files generated by more than 76 997 developers [6]. Thus, the cost of even a single false positive in the evaluation must be multiplied by those numbers.

A key observation is that in code injection attacks, only a minimal part of the codebase is modified [49]. One could simply focus on the last mile differences between the source code and the submitted packages. Hence our first question:

- **RQ1:** *Can we effectively and efficiently identify differences?*

A basic solution already exists: `git log`. For each line in an artifact, we check whether it is (or at least was) in the repository at some point. By iterating over all commits (revisions), we ensure that we collect everything in the source code repository, and we eliminate the need for identifying the pair of Git release/tag and PyPI release to be compared. Unfortunately, that does not scale as `git log` needs to loop over all revisions and spawns a heavy git process each time it is invoked. We could also use diffing techniques [2, 18], but they require a mapping of each PyPI release onto the corresponding Git tag or release, which does always exist.

Our algorithm LastPyMile is a feasible alternative to this problem. By cleverly combining package scraping and artifact hashing, we can extract these differences in a scalable way. Then, we can analyze how big is the gap in the field:

- **RQ2:** *How big are the 'normal' differences between source code and package repositories?*

We show that for more than 2000 popular packages in the PyPI ecosystem, such differences are pervasive. If a package code differs from the published source code, one cannot assume that it has been maliciously modified. Differences are too many (65% of artifacts and 22% of files in our sample) and too diverse for reproducible builds to be a solution. Yet, only few modifications happen in Python source files (2.6% of files) so that vetting might be a feasible alternative.

Finally, we can try to determine whether this solution can make a difference on the end goal: improving the vetting and coverage of scanners while keeping the number of false alerts manageable for PyPI maintainers given the imbalance ratio between the PyPI maintainers and the number of packages [50].

- **RQ3:** *Can LastPyMile be combined with package scanners while keeping the number of alerts manageable by a human?*

To be effective in the field, we should allow developers and development organizations to use the same tools to scan the source code repository of a package as part of their vetting process. Without protecting their investments in licenses, workflows, and developer education, an excellent technical solution would be doomed to fail. We show that such an approach is possible with LastPyMile.

2 TERMINOLOGY

Source code are human-readable instructions that others could check to understand the functionality of a software project.

Artifact is a software entity that contains all necessary items (e.g., files) to run the software and can be installed or directly used by project consumers. Typically, they are produced by the build process [24]. In Python, built distributions (e.g., `Wheels`) are generated from the source distributions (e.g., tarballs).[1]

Package 'exists to be installed (or deployed)'[2], and is a collection of pre-built and versioned artifacts for one or more target environments that is made available to consumers as an entity.

Repository is a cloud provider with a versioning system to store and access several versions of a software project. A *source code repository* stores and maintains the project source code, and a *package repository* distributes pre-built packages to consumers.

[1] https://packaging.python.org/glossary/#term-Source-Distribution-or-sdist
[2] https://packaging.python.org/overview/

An artifact present in a package available from a package repository is a *published artifact.*

Phantom is a software entity (e.g., files, lines of code) present in an artifact but does not match the one submitted to the source code repository. We use *phantom lines* to refer to lines of code and *phantom files* to refer to entire files.

3 BACKGROUND

3.1 The Last Mile from Source to Package

Figure 1 shows a typical package process of releasing a Python package in PyPI that consists of three main stages: development, build and distribution, and security review.

The primary activities of the *development* stage mainly happen at a source code repository (e.g., GitHub, GitLab, Bitbucket, etc.). At this stage, developers write all the code of a project. Developers may run various tools to test the functionality of the project and the absence of security vulnerabilities. If a project is open-source, other people could access the code, check it, and suggest improvements. The source code repository is often an essential source of information for the developers to decide the quality of a software [35].

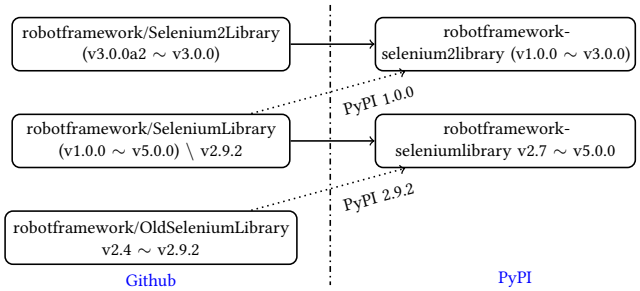
When developers decide to make the software version available for other people (i.e., make a release), they move the code to the *build* stage. At this stage, automated tools such as Travis CI, Jenkins, AppVeyor, or GitHub Actions use the information stored in the project configuration files to build it. These tools fetch the source code of the package and execute the build scripts that collect all the necessary dependencies, add package metadata, generate code (e.g., Swagger Codegen), and create artifacts that are ready to be distributed, like source archives, Linux, or Windows binaries, test coverage logs, and documentation.

At the *publication* stage, developers upload the artifacts to a package repository (e.g., PyPI, npm, Maven Central) either manually or automatically by using the build tool from the packaging stage. Most consumers will actually use the version of the software stored and published via package repositories. Uploaded artifacts need to go through the *security review* stage of a package repository. In PyPI, PyPI administrators run multiple checks (see Subsection 3.4) on uploaded artifacts. The checks will generate a verdict if an uploaded artifact contains suspicious behavior. The administrators are then reviewing the verdicts to decide to keep the artifact.

The match between the source code version of a project and the packages that correspond to that version is taken for granted [35]. However, *several tools (and humans) are involved at different stages of the pipeline, and some actions may result in a published artifact containing code that is not present in the source code repository.*

During the packaging stage, building tools add metadata files and augment existing code files (e.g., `setup.py`) with information [3], such as license, timestamp[3], release version, etc. Developers also use tools such as Swagger Codegen that automatically generate code files (e.g., server stubs and client SDKs for APIs). Developers may also change the code of a published artifact directly to backport a bug or a vulnerability fix [50].

Developers' actions might create difficulties to connect the distributed artifacts and their source code repositories (Figure 2).



When developers move stuff around repositories with different names the automatic traceability between package and source code repositories becomes hard as links in packages (solid lines) can point to the latest but possibly wrong source repository. A human must read the docs to find the correct Github repository.

Figure 2: GitHub tags and PyPI releases of `SeleniumLibrary`

Example 1. *The PyPI page of package `robotframework-selenium2library` points to the GitHub repository `robotframework/SeleniumLibrary` that contains the code for the releases before `v1.8.0` and after `v3.0.0a1`. The code for other releases is stored in a different GitHub repository `robotframework/Selenium2Library`.*

In this example, comparing a specific package version on PyPI with the corresponding GitHub tag for the releases in between `v1.8.0` and `v3.0.0a1` does not work as the corresponding code is not present in the referenced source code repository. One could only find the correct mapping between source and package repositories by manually inspecting repository descriptions.

Summary: Differences between the source and package repositories may be due to 'normal' activities.

3.2 Software Supply Chain Attacks

Software supply chain attacks occur when malicious or vulnerability is injected into diffent stages of the software development chain [20, 26]. Ohm et al. studied several attacks on different ecosystems [33] and found hijacking and typosquatting attacks to be the most common. Compromising the package owner's credentials would allow attackers to inject malicious payloads into the existing artifacts so that users will download and install them. Some examples of attacks are the injection of backdoors into PyPI `ssh-decorate` package [1], Ruby `rest-client` package, or npm `even-stream` package [19]. Attackers can commit malicious code into a source code repository [17, 37].

Package name *-squatting attacks are more prevalent than package hijacking [33]. In typo- and combosquatting attacks [33, 50] adversaries inject a malicious payload into the code of a popular package. Then they release this new package with a name nearly identical to the name of the original package to trick package users who mistype the package name and install the malicious one. This attack becomes especially attractive considering the limited automatic controls integrated into the package publishing process, and the certain unbalance concerning the number of package users and PyPI Administrators/Moderators (40K to 1) [53].[4] Several attempts

[3] https://github.com/pypa/wheel/issues/248

[4] Data collected on Feb 2020 from https://pypi.org/

Table 1: Discrepancies in files of legitimate and malicious typosquatting packages

Distrib shows the number of lines of code that are not present in the source code repository for the `requests` and `Flask` legitimate packages as well as the difference between the malicious packages `request` and `urlib3` from the original legitimate source code repositories of the benign packages (the targets) reported at [40]

Filename	#Lines of code	
	Source	+Distrib
`requests-1.2.2/requests/models.py`	687	+5
`Flask-0.5.2/flask/templating.py`	88	+2
`urlib3-1.21.1/setup.py`	174	+23
`request-1.0.117/hmatch.py`	27	+81

were made to identify the typo- and combosquatting packages present in the package managers [4, 42, 45, 50].

Example 2. *A typosquatting PyPI package `urlib3` [40] impersonated the popular package `urllib3`.[5] `urlib3` contains malicious code to exfiltrate user information to a remote server. `urlib3` has a single release `urlib3-1.21.1.tar` and the same Github URL as `urllib3`.[6]*

Table 1 shows the modified files in both legitmate packages: `requests` and `Flask` and malicious packages `request` and `urlib3`. Both kinds of packages differ from the source.

Summary: Discrepancies can also be due to malicious reasons. Attackers can inject malicious code or restore vulnerable code for later exploitation when a package is installed.

3.3 Reproducible Builds as an Ideal Solution

Reproducible builds [11] is a set of development practices that create an independently verifiable path from source to published artifacts. They could be the ideal solution to verify that no vulnerabilities or backdoors have been introduced during the build process.

However, to achieve the reproducibility of the build process, we must eliminate varying elements in release pipelines. For example, builds should not include any CPU, timestamp, or locale information in distributed artifacts [18]. Hence, reproducible builds require a significant overhaul in the language-based package managers such as PyPI or npm [3, 18] because current release pipelines augment packages with more information, such as metadata, debug data, or automatically generated code files (See Section 3.1).

Some free software distributions, such as Debian, have procedures to identify the original source code and a difference file that includes all changes made specifically for Debian, including all files related to packaging [23]. However, after trying for seven years, Debian states that "it is a stretch to say that Debian is reproducible".[7]

Summary: Reproducible builds are challenging to achieve given the diversity of packaging tools and current implementations of the release pipeline (e.g., embedding timestamp into artifacts).

3.4 Current PyPI Packages Scanners

Table 2 summarizes the existing tools that support identifying malicious code injections in Python packages. Several scanning tools [5, 13] parse files into abstract syntax trees (AST) and perform

[5] https://pypi.org/project/urllib3/
[6] https://github.com/urllib3/urllib3
[7] https://wiki.debian.org/ReproducibleBuilds

Table 2: Existing tools for analyzing PyPI packages

Regex (Regular Expression) bases on the raw lines of code while AST (Abstract Syntax Tree) requires transforming the code into a tree. The hybrid analysis consists of metadata, AST, and dynamic execution of an artifact

Tool name	Input	Technique
Malware Checks [52]	`setup.py` file	Static (Regex)
MalOSS [13]	Package	Hybrid analysis
Application Inspector [28]	Artifact	Regex
OSSGadget [29]	Package & Artifact	Static (Regex)
Ohm et al. [32]	Artifact	Static (AST)
Bertu [5]	`setup.py` file	Static (AST)

rule-based searches on their nodes. ApplicationInspector [28] and OSSGadget [29] use regular expressions to identify suspicious code lines. However, the tool authors mention that their tools generate many false positives if run on the entire package code [29]. This high number of false positives is to be expected.

```
# Establishing a socket connection to a server
s = socket.socket(socket.AFINET, socket.
    ↪ SOCKSTREAM)
rip = 'M' + 'TIxL' + 'jQyL' + 'jIx' + 'N' + 'y4'
    ↪ + '0NA' +  '=='
# Sending the encoded data via the socket
s.connect((base64.b64decode(rip), 017620))
```

Listing 1: Malicious code snippet opening of a socket to an encoded network address.

```
# Decoding a bundle of certs in PEM format
dercerts = [
        base64.b64decode(match.group(1)) for
    ↪ match in PEMCERTSRE.finditer(pembundle)]
```

Listing 2: Legitimate `b64decode` call in the `urllib3` package

Consider the code snippets from Listing 1 and Listing 2. Both code snippets use `b64decode` function from `base64` library. Listing 1 is a malicious fragment that collects the user information and sends it to a remote server via a network socket, while the code in Listing 2 simply decodes a (benign) certificate. A package checking tool that consider `b64decode` function as suspicious since it is often used in malicious packages will produce a true positive for Listing 1 and a false positive for Listing 2. Unfortunately `b64decode` function is widely used for benign purposes, and the tool will generate many false positive alerts as it has no way to distinguish benign from malicious usage without further analysis.

To avoid being overwhelmed by false positives, the current PyPI security review called Malware Checks [52] scans only the installation script `setup.py`. Unfortunately, several known attacks [27, 40] had malicious code injected into different files. Hence, the review of only `setup.py` files is not enough.

Example 3. *The typosquatting package `jellyfish` [27] mimicked the popular package `jellyfish` (the first L is an I) to steal SSH and GPG keys [8]. There are two injected files: `setup.py`, and `_jellyfish.py` in the typosquatting package. The malicious code is stored in the `_jellyfish.py` as shown Listing 4 and then being*

Table 3: Number of source code repositories found by locations

Metadata of a package contains multiple fields such as Homepage, Codepage. Package Homepage is the main page which contains additional information about a package (e.g., documentation)

Location	#GitHub Repos	Percentage (%)
Homepage (Metadata)	2618	77.9
Codepage (Metadata)	68	2
Package Homepage	1418	42.2
PyPI Homepage	1974	58.7
Total GitHub Repos	3662	100

implicitly called by the package installer via the `packages` *option in the* `setup.py` *file (Listing 3).*

```
# Process pure Python modules in 'jellyfish'
packages=['jellyfish']
```

Listing 3: The file jellyfish-0.7.1/setup.py

```
ZAUTHSS = PAYLOAD
# Decoding and executing the obfuscated payload
ZAUTHSS = base64.b64decode(ZAUTHSS)
ZAUTHSS = zlib.decompress(ZAUTHSS)
exec(ZAUTHSS)
```

Listing 4: The file jellyfish-0.7.1/jellyfish/_jellyfish.py

Summary: If scanning one file in a package is feasible but not enough and reviewing an entire package is unfeasible due to the high number of false positives, a different solution is needed.

4 RQ1: LASTPYMILE TO IDENTIFY CODE INJECTION

The upper part of Figure 3 shows the typical process of the security review process in package repositories (e.g., PyPI) for identifying suspicious artifacts that might occur during the release of a software project. First, the code in the published artifact is undergoing code review and scanning by the PyPI Administrators by running security checks [52]. Currently, they are using two checks `SetupPatternCheck` and `PackageTurnoverCheck` (see Section 7).

Depending on the automated tool used by the maintainers, this scanning could be done on the entire artifact for backdoor injection (e.g., Bandit) or on its files (e.g., Malware Checks [52]). Then the output of the scan is used to decide whether the artifact should be uploaded to the package repository.

The bottom part of Figure 3 shows how LastPyMile can augment the traditional security process. As a preliminary, LastPyMile looks for the GitHub URLs of a PyPI package in various places, including package metadata, PyPI, and package homepage. Table 3 shows the number of GitHub URLs we found. Most of the packages declare their GitHub repositories in the metadata available on PyPI.

In Step 1, LastPyMile iterate all commits to compute all file hashes and collect line contents from a source code repository. To ensure that all the files and lines are collected, LastPyMile processes commits from all branches and tags in the GitHub repository. LastPyMile supports processing the GitHub repository in parallel so that multiple commits can be processed simultaneously. Besides,

Step 1 Hashing files and lines from source code repository

Require: The Github URL of the package: $GithubURL$

Set of file hashes in the repository H_s: []
Set of lines of files in the repository L_s: []
$Cloned\_Dir = CloneRepositoryToDisk(GithubURL)$
$Commits = GetCommitsFromRepo(Cloned\_Dir)$
for each $c \in Commits$ **do**
    $F_s \leftarrow CheckoutFilesInCommit(c)$
    for each $f \in F_s$ **do**
        $H \leftarrow H \cup SHA256(f)$
        $L \leftarrow L \cup ReadFile(f)$
return Set of file hashes, lines: H_s, L_s

Table 4: Running time comparison between LastPyMile and `git-log` approaches

Both approaches had been run in the same environment. The differences obtained by both the approaches are the same (e.g., number of phantom files and lines)

Package	`git log` (seconds)	LastPyMile (seconds)
certifi	1244	48
idna	408	34
six	315	145
s3transfer	1095	44

to avoid processing the same commits in different branches, LastPyMile maintains a shared set of already processed commits for synchronizing the processing tasks.

Example 4. *18 distributed artifacts nameko-3.0.0.rcX contain the source code that is stored in the* `v3.0.0-rc` *branch.*

After collecting all the file hashes and lines from the Github repository, in Step 2 LastPyMile processes a package artifact to calculate file hashes and collect file lines. Finally, LastPyMile compares file hashes and lines of distributed artifacts and those in the source code repository to report the phantom files and lines (Step 3).

LastPyMiletakes only 0.04 seconds for scanning `jellyfish` artifact that consists of 530 unique files and 28104 lines on a laptop with four CPU cores and 8 GB RAM. Considering the top four most downloaded packages `six`, `idna`, `python-certifi` and `s3transfer` as shown in Table 4, LastPyMile is 16x faster than the default iterative approach that relies on calling *git log* command for every line of an artifact because LastPyMile preproceses all commits in a repository and require only a single pass over all code, while *git log* must iterate over all revisions each time it is invoked.

Table 5 compares the number of total files and lines present in the analyzed files with the phantom files and lines reported by LastPyMile. We observe that more than half of `setup.py` files are phantom, while the number of phantom lines of code in the `setup.py` files is six times smaller compared to the total number of lines in `setup.py` files. Globally the number of phantom lines of code is 16 times smaller. Table 6 shows that a median artifact contains two phantom lines that include at least one API call (e.g., execute some function) and two lines that import some library.

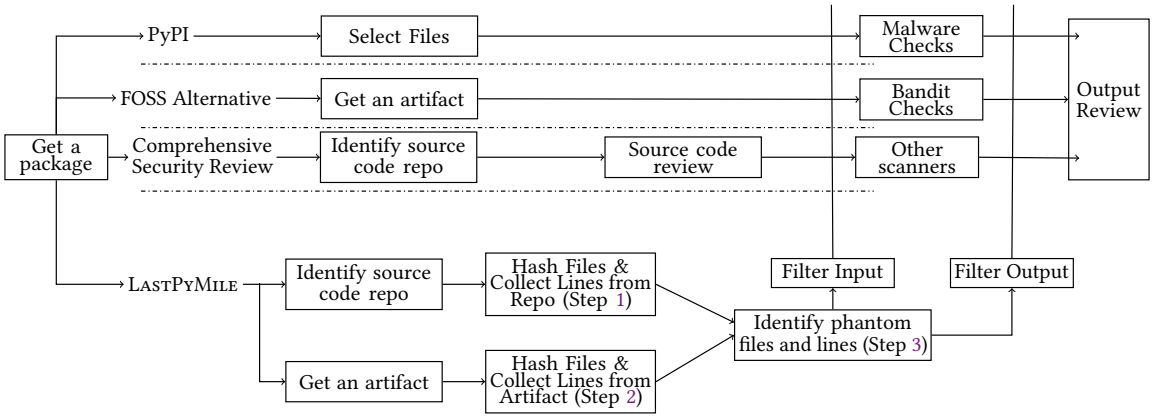


Figure 3: LastPyMile in the context of the overall security review pipeline

Step 2 Hashing files and lines from an artifact

Require: A, the PyPI package artifact to be evaluated
1: Set of file hashes in an artifact H_p: []
2: Set of lines of files in an artifact L_p: []
3: Artifact URLs: $A_s = ObtainArtifactURLs(p)$
4: $Local\_Artifact \leftarrow DownloadArtifactFromPyPI(A)$
5: $F_s \leftarrow UncompressArtifact(Local\_Artifact)$
6: **for each** $f \in F_s$ **do**
7: $H \leftarrow H \cup SHA256(f)$
8: $L_s \leftarrow ReadLinesFromFile(f)$
9: **for each** $l \in L_s$ **do**
10: $L_p \leftarrow L \cup l$
11: **return** Set of file hashes, lines: H_p, L_p

Step 3 Identifying phantom files and lines in distributed artifacts

Require: H_s, L_s, H_p, L_p
1: Set of phantom files: H_d: []
2: Set of phantom lines: L_d: []
3: **for each** $h \in H_p$ **do**
4: **if** $h \notin H_s$ **then**
5: $H_d \leftarrow H_d \cup h$
6: **for each** $l \in L_p$ **do**
7: **if** $l \notin L_s$ **then**
8: $L_d \leftarrow L_d \cup l$
9: **end if**
10: **end if**
11: **return** Set of phantom files hashes, lines: H_d, L_d

Summary: LastPyMile enables checking the entire codebase of a published artifact 16x faster than the `git log` approach as LastPyMilerequires only a single pass over all commits.

5 DATA COLLECTION

To select the sample of Python packages for our study, we start with the list of the top 4000 most downloaded packages [46], which is the established approach to study the Python ecosystem, adopted both in academia [6] and industry [30] and [12].

Table 5: Number of unique phantom files and lines versus total

The columns in the left are the files and lines that are processed by the PyPI Malware Checks and existing scanning tools while LastPyMile only processes the phantom files and lines on the right. Phantom files are counted by their unique hashes

	#Total		#Different	
	`setup.py`	All	`setup.py`	All
#Files	4056	90 143	2532	16 170
#Lines	38 750	14 027 895	7236	939 772

Table 6: Statistics about lines not in the repo

	Mean	Min	Q25%	Median	Q75%	Max
#APIs	4	1	1	2	3	946
#Imports	2	1	1	2	3	12

We identify 3662 packages (>91% of the selected Python packages) that use GitHub to maintain their source code. Among these packages, 3336 are unique repositories (83%). For simplicity, here we focus only on packages that claim their source code is on GitHub. Table 7 shows the characteristics of the collected repositories. Three repositories contain only two commits[8], while several repositories had tens of thousands of commits (e.g., `pip` has 10 730 commits[9]).

As we aimed to have a tool to be runnable "as you wait" [39], we set a timeout period of five minutes for analyzing all artifacts of a given package. As a result, the selected packages resulted in 109 062 artifacts. We had to exclude 15 810 artifacts (14%) belonging to 'surviving' packages with early versions being developed on versioning control systems other than Git and/or with the commit history not being included when moving to GitHub. We could not use them in our analysis as there was no source code to compare. The final dataset comprises 93 252 artifacts from 2438 packages, 65% of them are gzip, 29% are `wheel`, 4% are zip, and 2% are eggs.

After checking the differences between the number of different files and code lines between source and package repositories

[8]For example, https://github.com/datamade/probableparsing
[9]At the time of data collection

Table 7: Descriptive statistics of GitHub repositories for the selected packages

Tags includes Github tags and branches of a Github repository. Unique files and lines are determined by their hashes and contents, respectively

Number of	Mean	Min	Q25%	Median	Q75%	Max
Tags	29	1	9	19	36	678
Commits	477	2	91	232	548	10 730
Unique files	97	3	14	29	68	17 000
Unique lines	53	1	6	17	43	8732

Table 8: Number of processed packages and artifacts

The processing time threshold is set for a package. We exclude artifacts that predate the creation time of a Github repository

Step	Result
Top-most downloaded packages	4000
Processed #artifacts (proc. time $<$ 5 min)	109 062
Artifacts with corresponding tags in GitHub	15 810
Final #artifacts (linkable to source)	93 252

(Figure 4), we observed that *66* artifacts featured a huge number of changes (>1000 different files). We manually analyzed those artifacts and found that the explanation lies in 'developers moving stuff around' across repositories, making it close to impossible to identify source code repositories by automatic means. The example in Figure 2 requires one to actually read the documentation.

Besides the `robotframework-selenium2library` in Example 1, we found that `sas7bdat` package first hosted its source code on GitHub but then was moved to BitBucket. The other reason for not being able to locate the corresponding source code of a package automatically is the usage of submodules [54] by developers. We removed such artifacts from our analysis as their source code could not be found automatically. Hence, the final list of analyzed artifacts comprises 93 252 artifacts. Table 8 summarises the number of analyzed packages and corresponding artifacts.

Example 5. *The `gsutil` package refers to a GitHub repository `GoogleCloudPlatform/gsutil` with two submodules. For both `Pyrogram-0.8.0-py3-none-any.whl` and `Pyrogram-1.0.3-py3-none-any.whl` artifacts, we could not find the related GitHub tag or release. Our manual analysis of these packages did not reveal malicious injections.*

6 RQ2: DIFFERENCES BETWEEN SOURCE CODE AND PACKAGE REPOSITORIES

To answer RQ2, first, we compared the code distributed in PyPI artifacts with the corresponding source code repositories. Figure 5 shows that 65% of artifacts and 22% of files present in PyPI have changes with respect to the source code repository. I.e., they might have malicious code injected during the package release process. However, only 5.8% of artifacts and 2.6% of files have changes in Python files, while 59% of artifacts and 19% of files have changes in other files. These findings suggest that it might be promising to limit the process checking for malicious injections to those artifacts and files that have discrepancies, as the other artifacts cannot have malicious injections during the release process. In this paper, we focus on the changed Python files as they might be the target of attackers for injecting executable malicious commands.

Table 9: Differences between package artifacts and their source code repositories

Unique files are the files having different hashes while number of lines are the total number of lines in an artifact

#Files	Mean	Min	Q25%	Median	Q75%	Max
Number of Unique Files						
Python	9	1	1	1	6	994
Metadata	4	1	3	4	5	19
Number of Lines						
Python	19	2	2	4	12	1988
Metadata	8	2	6	8	10	38

Metadata files have a great impact on the number of differences between source code and package repositories: Table 9 shows that a median artifact has four metadata files[10] and nine Python files (twice more). This difference is also visible at code line level: a median artifact has 2-8 lines in phantom metadata files and 18 lines in phantom Python files.

We observe that nearly 15% of Python files that have differences with respect to the source code repository are `__init__.py` and `setup.py` files (Table 10). Most likely, this happens since the building tools introduce some additional information (e.g., timestamps, versions, etc.) into these files during the packaging process. Similarly, the `_version.py` and `version.py` files are used to identify the package version from a Git tag or release automatically.

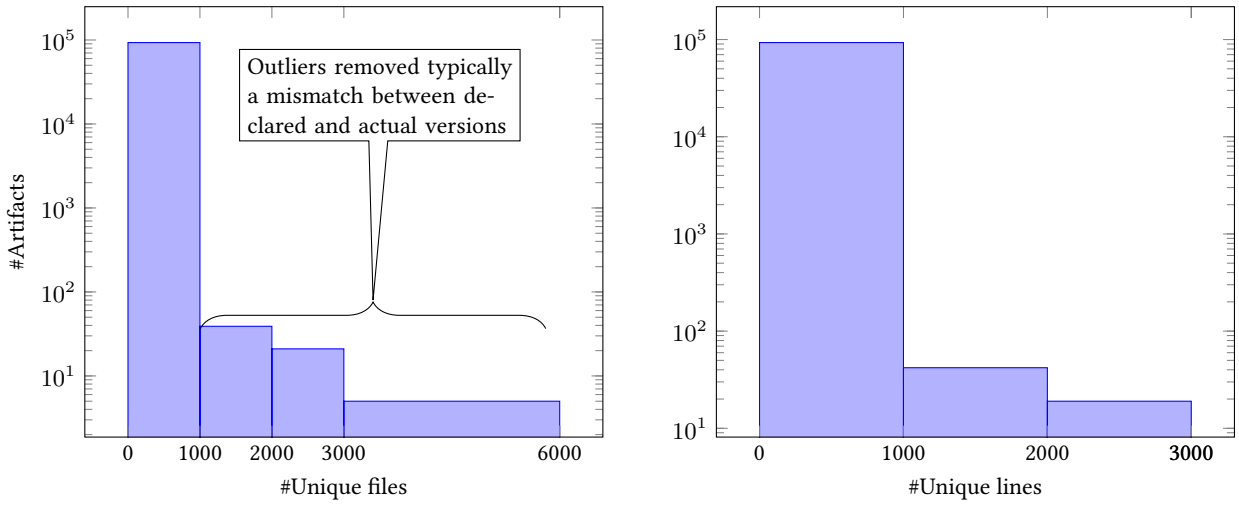
Table 11 shows the top ten regular and API calls related to networking and system in the Python files that differ from the source code repository. Many files have calls to such functions as `urlopen`, `socket.socket`, `request` to open URLs and make HTTP requests, `subprocess.Popen` and `exec` to open files. Usage of these functions could be harmful. At the same time, these functions are often used for legitimate operations, and one cannot simply mark all lines that include a call to 'possibly suspicious' APIs as 'actually suspicious' – there would be an unmanageable number of false alerts.

Summary: The code distributed via package repositories has many changes with respect to the code stored in the corresponding source code repository. On average there are 5.8% of artifacts and 2.6% of files have changes in Python files.

7 RQ3: LASTPYMILE COMBINED WITH OTHER PACKAGE SCANNERS

The combination of LastPyMile with existing security scanners is essential for two reasons: First, it allows to reuse mature detection techniques of FOSS and commercial security scan tools. Second, by doing so, developers and development organizations can use the very same tools in different stages of the security review process, which protects their investments into software licenses, the design and implementation of review workflows, and developer education.

[10] We identified metadata files as generated by packaging tools (e.g., WHEEL), dependency declaration files (e.g., `requirements.txt`), and documentation files (e.g., README.md)



(a) #Files differing between source and package

(b) #Lines differing between source and package

Figure 4: #Files and #lines differing between source and package repositories

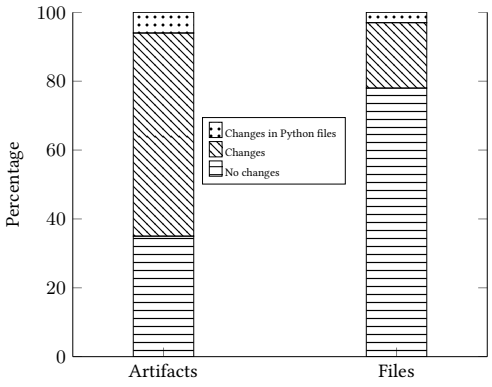


Figure 5: Percentage of different kinds of changes in artifacts and files

Table 10: Top different phantom Python files in our sample.

Phantom files are present in the package source code but have different content than the omonimous file in the source code repository. The same file name might occur multiple times in the same package with different paths. `__init.py__` and `setup.py` are the most common phantom files.

Filename	#Phantoms	Percentage (%)
__init__.py	36 480	14.5
setup.py	7414	3
_version.py	4152	1.7
version.py	3260	1.3
utils.py	2354	1
v1.py	1498	0.6
v2.py	1498	0.6
base.py	1404	0.6
client.py	1050	0.4
exceptions.py	1008	0.4

Table 11: Top ten API calls in modified Python files

API calls are grep from the line contents using a set of regular expressions. We exclude some internal calls of the packages.

Top	occurences	Network & System	occurences
__init__	72 413	urlopen	793
isinstance	55 115	socket.socket	711
datetime	37 393	subprocess.Popen	670
ttinfo	37 258	exec	580
len	36 325	request	541
read	31 582	http.request	511
getattr	21 575	s.setsockopt	413
super	16 760	requests.post	323
hasattr	16 358	request.get	317
join	13 869	os.chmod	303
append	12 548	platform.system	292

As shown in Figure 3, PyPI Administrators can achieve the reuse by filtering either the input or the output of such security scanners. They can feed tools operating on single files (Malware Checks), modules, or procedures (Bandit Checks) with input containing *phantom lines*, which is expected to reduce both the number of findings and the tool's runtime. Scanning tools performing the whole program or inter-procedural analyses continue to work on the package's entire code base. Still, their output can be filtered to only show findings in phantom lines.

In this paper, we focus on input filtering and show the results of combining LastPyMile with two well-known malware checking tools that are broadly used in the PyPI ecosystem:

- **Warehouse Malware Checks** [52] tool is used by PyPI to check the suspicious code lines in every package uploaded to PyPI. At the time of writing, the tool supports two

Table 12: LastPyMile on Malware Checks and Bandit alerts

Malware Checks Alerts (X rules on Lines), while Suspicious Bandit Alerts (Y rules on Files). The `setup.py` column of Malware Checks Alerts is what happens now in PyPI [52]

Artifact	Type	In setup.py	Problem Size				Malware Checks Alerts			Suspicious Bandit Alerts		
			#Files	#LoCs (all files)	#LoCs (setup.py)	Coverage (setup.py)	setup.py	whole pkg	LastPy Mile	setup.py	whole pkg	LastPy Mile
urllib3-1.26.3	Benign		80	25 348	97	0,4%	1	260	0	8	1398	0
requests-2.25.1	Benign		32	9325	112	1,2%	3	57	0	9	505	0
setuptools-53.0.0	Benign		244	70 794	162	0,2%	4	2932	0	5	762	0
urlib3-1.21.1	Malicious	Y	72	20 448	197	1%	4	177	3	20	1044	12
request-1.0.117	Malicious	N	3	166	52	31,3%	2	8	2	5	27	20
setup-tools-36.0.1	Malicious	Y	112	31 245	304	1%	8	1289	3	21	489	12

checks: `SetupPatternCheck`[11] for performing regular-expression based checks of the content of `setup.py` files for suspicious patterns on package upload and `PackageTurnoverCheck`[12] for performing daily scans for suspicious behavior about package ownership. Conceptually Malware Checks is close to other open-source tools for auditing FOSS packages [28, 29] that rely on regular expressions to the whole artifact.

- **Bandit** [38] is a tool supported by the Python Code Quality Authority. Bandit was designed to find common security issues in Python code by scanning all the files included in a software artifact. For each file in the artifact, the tool creates an abstract syntax tree (AST) representation and performs rule-based analysis (plugins) of the AST nodes. Most of the Bandit rules focus on the vulnerabilities in Python code (e.g., Start a process with a function vulnerable to shell injection)

For Malware Checks, we focus on `SetupPatternCheck`. Even though this tool currently only checks `setup.py` files, we have extended it to scan all files of a software artifact.

For the Bandit tool, we have used the default set of Bandit rules and then extended them with additional rules so that the tool is capable of findings all malicious lines of code injected into Python packages known to be used in typosquatting attacks [33, 50]. Our rule set checks for suspicious API calls (e.g., `exec`), imports (e.g., `socket`), and strings (e.g., an URL). Our rules can be found at [48].

To illustrate how the malware checking tools perform on the artifacts without malicious payloads, we compare their outcome on three example benign artifacts that correspond to the following malicious artifacts. We collected the malicious artifacts from the real attacks by contacting the researchers who reported the attacks.

- `urlib3-1.21.1` – malicious code was injected into the `setup.py` file. It triggered automatic extraction of data and sending it to a remote server using the `socket` library.[13]
- `request-1.0.117` – while the `setup.py` file contains the code to trigger the malicious execution from the `hmatch.py` file, the actual malicious functionality was implemented in the `hmatch.py` file: scanning the computer network and sending results to the remote server using `urllib3` library.[14]
- `setup-tools-36.0.1` – the malicious code injected into the `setup.py` file triggered automatic extraction of sensitive data and sending it to the remote server via `socket` library.

Table 12 presents the results of Malware Checks and Bandit tools' scans of the selected artifacts. Since the Malware Checks tool was primarily designed to scan only `setup.py` files, we report the number of findings the tools produced on the `setup.py`. Then we present the number of alerts when we run the tools on the whole package. Finally, we show the number of alerts the tools produced on the lines of phantom code as reported by LastPyMile. The replication package for Table 12 can be obtained at [51].

We observe that Malware Checks produced at most three alerts on each of the benign and malicious artifacts when only the `setup.py` file was considered. While this amount of alerts is manageable by humans, checking only the `setup.py` files allows one to have coverage of around 1% of the total code base of the analyzed artifacts, except the malicious `request` artifact where scanning `setup.py` has generated coverage of 31.3%.

When Malware Checks was executed on all files from the package, the number of alerts rockets to 2-3 orders of magnitude. Notably, the tool produced more alerts on the benign artifacts than on the malicious packages. This phenomenon corresponds to the more extensive code base of the legitimate artifacts.

We observe similar behavior of the Bandit tool. When applied on the `setup.py`, the tool generated alerts both on benign and malicious artifacts. However, Bandit produced significantly more alerts on the malicious artifacts. When looking at the alerts generated after running the tool on the entire package, we observe a large number of alerts. Notably, looking only at the number of alerts, one could not distinguish between benign and malicious artifacts: the number of alerts produced on the benign artifacts exceeds the number of alerts on the malicious artifacts.

After applying LastPyMile to the tool results after running them on the entire artifacts, we observe a significant reduction of the number of alerts for both tools. For example, Bandit tool produced only 12 alerts (out of 1044) after applying LastPyMile on the results of the `urlib3-1.21.1` scan. Similarly, the number of alerts produced by Malware Checks on the `setup-tools-36.0.1`

[11] https://github.com/pypa/warehouse/tree/master/warehouse/malware/checks/setup_patterns
[12] https://github.com/pypa/warehouse/tree/master/warehouse/malware/checks/package_turnover
[13] https://docs.python.org/3/library/socket.html
[14] https://urllib3.readthedocs.io/en/latest/

reduced to 12 instead of 489. Looking at the outcome of the benign packages, LastPyMile reduced the number of alerts to zero.

Being applied to `setup.py` files only, Malware Checks tool generates a number of alerts manageable by humans. However, scanning of only `setup.py` files does not guarantee the artifact to be free from malicious code as the 99% of code is not checked. The number of alerts that both tools produce after scanning the entire artifacts (3249 and 2665 false alerts for Malware Checks and Bandit respectively) demonstrates that such analysis does not scale for an 'on-upload' analysis by PyPI maintainers.

In contrast, LastPyMile shows an excellent potential to improve the scanning results. First, it makes the number of alerts after running a tool on the entire artifact comparable with the current number of alerts generated by the Malware Checks (currently used by PyPI). Second, we do not observe any alerts for benign artifacts, which allows us to easily distinguish benign and malicious artifacts in our manual validation of the alerts.

When run on all malicious code packages available from the literature, we were able to preserve all malicious alerts and did not introduce false positives over the current scanning process.

Those properties make LastPyMile a candidate for software vetting processes of government organizations or other OSS consumers with high-security requirements. The review effort is manageable, even though typical development projects have dozens of dependencies with more or less frequent release and patch cycles.

Summary: LastPyMile reduces the number of alerts produced by a malware checking tool to a number that a human can check. We checked our approach against known malicious packages, and we found that LastPyMile can detect all of them. Also, it removes all the alerts from benign packages, allowing a clear distinction between benign and malicious packages.

8 THREATS TO VALIDITY

The validity of results reported in this paper is impacted by several choices made during tool and experiment design.

We only consider repositories hosted on GitHub[15]. However, there are no significant obstacles to cover other version control systems and extend the current implementation to other Git service providers (e.g. GitLab or Bitbucket) as long as they support code commits (e.g., Apache Subversion).

The current implementation focuses on the Python packages in PyPI, and Python files in particular. The extension to other Python ecosystems (e.g., `anaconda`)[16], and interpreted languages and other file types seem straight-forward (e.g., Node.js/npm and Ruby/RubyGems). Yet, we only considered the top 4000 packages hosted on PyPI, out of more than 250 000 packages. A larger number of packages would need to be considered for an ecosystem analysis.

In terms of design, *LastPyMile checks only the code absent from source repositories* even though malicious code could also be included in the versioned code, either directly or in tests. This was the case of the *Pillow* Python framework[17] that was flagged by more than 15 Antivirus vendors. However, this situation lays out of the scope of the paper as the test files should have been spotted during the source code review.

We limit the line-by-line analysis to files with file extension `.py`. The main reason driving this design decision is to focus attention on files whose discrepancies, compared to what users can view in the respective source code repository, can alter the program flow (e.g., when downstream users install an artifact in their development environment or invoke its API as part of their development project).

Other phantom files might be also be used to inject malware. For example, the phantom files under the test directories are required by a popular testing framework like *pytest*. Another source of phantom files is the upload of modules specific to a developer development environment. They are usually not versioned with Git.

Example 6. *The phantom files in* `pydruid-0.5.4.tar.gz` *are the manually built Python packages stored in the site-packages directory. We can verify the origin of the local installed modules by comparing their code files with the correponding GitHub repository. By using LastPyMile, we can check that the code files in the local module called* `traitlets` *(e.g., https://github.com/ipython/traitlets) of the artifact* `pydruid-0.5.4.tar.gz` *belonging to the correponding Github repository.*[18]

Moreover, PyPI packages contain other executables, e.g., Windows portable executables, OSX disk image files, or C/C++ static libraries. For example, we found many Python bytecode files (ending with `.pyc`). These files should not be uploaded to PyPI as this can make the dependent package (e.g., a Debian package) fail to compile.[19] Investigating these cases would require a distinct paper.

We only check additions of code lines in the present version, even though a vulnerability could be introduced by deleting lines from a software artifact (e.g., by removing a sanitizing statement). Albeit LastPyMile does not report the deleted lines in such a case, it could detect that the files in the uploaded artifacts are different as their hashes would differ if compared to the hashes of the files stored in the corresponding source code repository. Limiting the false alerts, in this case, would require special care to avoid that the whole file is reported as different. We leave this case for future work.

Some packages contain code automatically generated by tools like Swagger Codegen or Python distutils. The current implementation of LastPyMile would generate conceptually false positives as such files do not conceptually differ from phantom files. These cases of automatically generated files could be checked by applying the same code generation tool on the code files in the Github repository and comparing with the files in published artifacts.

9 RELATED WORK

Zimmermann et al. [55] study security risks for users of npm, including potential vulnerable and malicious code in third-party dependencies. The authors showed that npm suffers from single points of failure in which individual packages could impact large parts of the entire ecosystem. Attackers could compromise a minimal number of maintainers' accounts to inject malicious code into most packages. The paper, however, does not investigate the potentially malicious code injection in the package artifacts. Our LastPyMile

[15] In our dataset, there are 56 packages hosted in bitbucket.org, 14 packages hosted in Gitlab, 13 packages hosted in the sourceforge.net, 19 packages are hosted in code.google.com, and four of them had been moved to GitHub.
[16] https://repo.anaconda.com/
[17] https://github.com/python-pillow/Pillow/issues/251
[18] https://github.com/ipython/traitlets
[19] https://github.com/googleapis/google-auth-library-python/issues/214

approach provides a way to audit the popular packages by identifying the delta in the code that developers consume and the original code developed by vendors in their source code repository.

Ohm et al. [33] presents a taxonomy of attack vectors and a dataset of malicious software packages used in real-world attacks on open-source software supply chains on three package repositories npm, PyPI, and RubyGems. This work highlights that typosquatting and infection of an existing package are the two most common attacks. This work, however, does not investigate the presence of code injections in legitimate and malicious packages. Our approach involves developing a method to identify the discrepancies between distributed artifacts and source code repositories that could be attributed to the malicious code injection.

Several works ([44, 50]) study the potential impact of typosquatting attacks on PyPI packages based on the Levenshtein distance and number of downloads of the targeted package. They show a large number of typosquatting candidates in PyPI that PyPI Administrators should investigate. However, relying only on package names may cause many false positives, and a more in-depth analysis of distributed artifacts is required. Our approach provides a methodology to verify the potential typosquatting package by highlighting the differences between the typosquatting packages and the GitHub repository.

Code-based approaches ([5], [34], [13], [36]) can provide more accuracy in detecting malicious packages. Several approaches scan the `setup.py` file of the package artifact to identify suspicious code. Bertu [5] statically parses the Python installation script `setup.py` as an AST tree and looks for suspicious patterns (e.g., network connections). Malware Checks uses a set of Yara rules on the lines of code on the `setup.py` file. Similarly, `ApplicationInspector` and `OSSGadget` [29] check the distributed artifacts using a set of regular expressions to identify potential backdoors within a package. Although these approaches are fast and straightforward, they can generate many false alerts when scanning `setup.py` performing legitimate behavior in the installation process (e.g., downloading a dependency from a remote server). Our approach fills this gap by allowing these tools to scan only the phantom lines which were potentially introduced.

Several techniques use dynamic analysis to expose the malicious behaviors of the package. `Buildwatch` [34] dynamically execute package code in the Cuckoo sandbox [15] and captures all system calls, such as kernel services requested by processes. Duan et al. propose `MalOSS` [13], a hybrid approach, which extracts various features of distributed artifacts using metadata, static and dynamic analyses. These methods, however, are resource-heavy which may be challenging to integrate into the development pipeline. LastPyMile uses a lightweight comparison to help these approaches by reducing the number of files and lines that need to be scanned to detect malicious code injections, making the existing techniques efficiently adapt to individual developers' development pipelines.

Garrett et al. [14, 16] use an anomaly detection based approach on features extracted from packages' metadata and source code to detect suspicious package updates in npm. The method could reduce the review effort by 89%. However, the approach cannot highlight the code injections with the existing features as it analyzes only the published artifacts. Our LastPyMile can highlight the code injections and can be adapted to provide explanations to developers.

Gonzalez et al. [17] uses commit logs and repository metadata to detect potential malicious commits automatically. The method identifies 53.3% of malicious commits while flagging less than 1% of commits in the studied dataset. Our LastPyMile instead looks for the code injections into the source code repositories by considering both the packages and source code repositories.

Summary: Although current approaches on auditing packages caught some malicious examples, their focus is on detecting malicious patterns in published artifacts, which may cause many false alerts. Also, scanning the whole code of an artifact would not be effective in case of code injection attacks where only a small subset of code is malicious. Instead, our approach focuses on detecting code injections in distributed artifacts, thus complementing the current techniques by reducing the number of code lines to be analyzed to detect software supply chain attacks.

10 CONCLUSION

We investigated the discrepancies between published artifacts and source code repositories to understand the risk of malicious injections during the software release process. Our empirical analysis of 2438 most downloaded PyPI packages shows that there exist differences between packages in PyPI and the corresponding source code repositories at different levels of granularity (artifacts, files, and lines). The differences are attributed to developers and automated tools (e.g., packaging tools), and could impact the consumers, e.g., causing compilation issues or representing a potential for containing malicious code injections.

The flexible combination of LastPyMile (as input/output filter) with other security tools offers the possibility to reduce the number of findings and the time required by vetting processes. We instructed Malware Checks and Bandit to only consider phantom code as input, and the resulting decrease in false alerts makes it possible to use LastPyMile as an additional check in the PyPI vetting processes with minimal impact on review efforts.

A replication package is available at [51] and we plan to submit LastPyMile as a new check to PyPI.[20]

Several issues still remain: malicious code can be hidden in many other forms, such as webpages (HTML with embedded or external JavaScript) or project configuration files (`requirements.txt` with a malicious dependency). We notice a high number of `requirements.txt` configuration files, which contain the list of dependencies to be installed automatically with `pip install`. This could be a potential vector for adversaries to add malicious injections worth further investigations.

ACKNOWLEDGMENTS

We are grateful to Michael Salsone, Lukas Martini for sharing with us the malicious examples, and the anonymous reviewers for their insightful and actionable suggestions.

This research has been partly funded by the EU H2020 Programs H2020-EU.2.1.1-CyberSec4Europe (Grant No. 830929), AssureMoss (Grant No. 952647) and SPARTA project (Grant No. 830892).

[20] https://warehouse.pypa.io/development/malware-checks.html#adding-new-checks

REFERENCES

[1] 2018. Backdoor in ssh-decorator package. https://www.reddit.com/r/Python/comments/8hvzja/backdoor_in_sshdecorator_package/. Accessed 29 July 2020.

[2] 2020. in-depth comparison of files, archives, and directories. https://diffoscope.org. Accessed 15 Feb 2020.

[3] 2020. Tracking which wheels can be reproducibly built. https://www.redshiftzero.com/reproducible-wheels/. Accessed 03 January 2021.

[4] William Bengtson. 2020. Python Typosquatting for Fun not Profit. https://medium.com/@williambengtson/python-typosquatting-for-fun-not-profit-99869579c35d. Accessed: 2020-08-17.

[5] Bertus. 2018. Detecting Cyber Attacks in the Python Package Index (PyPI). https://medium.com/@bertusk/detecting-cyber-attacks-in-the-python-package-index-pypi-61ab2b585c67. Accessed 18 January 2020.

[6] Ethan Bommarito and Michael Bommarito. 2019. An Empirical Analysis of the Python Package Index (PyPI). *arXiv preprint arXiv:1907.11073* (2019).

[7] Catalin Cimpanu. 2018. Backdoored Python Library Caught Stealing SSH Credentials. https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/. Accessed: 2020-08-17.

[8] Catalin Cimpanu. 2019. Two malicious Python libraries caught stealing SSH and GPG keys. https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/. Accessed: 2020-08-17.

[9] Russ Cox. 2019. Surviving Software Dependencies: Software Reuse is Finally Here but Comes with Risks. *Queue* 17, 2 (April 2019), 24–47. https://doi.org/10.1145/3329781.3344149

[10] Stanislav Dashevskyi, Achim D. Brucker, and Fabio Massacci. 2016. On the Security Cost of Using a Free and Open Source Component in a Proprietary Product. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639* (London, UK) *(ESSoS 2016)*. Springer-Verlag, Berlin, Heidelberg, 190–206. https://doi.org/10.1007/978-3-319-30806-7_12

[11] Debian. 2019. Reproducible Builds. https://reproducible-builds.org/. Accessed: 2020-08-17.

[12] Charlie Denton. 2021. Python Wheels. https://pythonwheels.com/.

[13] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *Proc. of NDSS'21*.

[14] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *Proc. of ICSE'21)*. IEEE, 1334–1346. https://doi.org/10.1109/ICSE43902.2021.00121

[15] Stichting Cuckoo Foundation. 2021. cuckoo: Automated Malware Analysis. https://cuckoosandbox.org/.

[16] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting suspicious package updates. In *Proc. of ICSE'19: New Ideas and Emerging Results*. IEEE Press, 13–16. https://doi.org/10.1109/ICSE-NIER.2019.00012

[17] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. 2021. Anomalicious: Automated Detection of Anomalous and Potentially Malicious Commits on GitHub. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 258–267. https://doi.org/10.1109/ICSE-SEIP52600.2021.00035

[18] Pronnoy Goswami, Saksham Gupta, Zhiyuan Li, Na Meng, and Daphne Yao. 2020. Investigating The Reproducibility of NPM Packages. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 677–681. https://doi.org/10.1109/ICSME46990.2020.00071

[19] Danny Grander and Liran Tal. 2018. A Post-Mortem of the Malicious event-stream backdoor. https://snyk.io/blog/a-post-mortem-of-the-malicious-event-stream-backdoor/. Accessed: 2020-06-01.

[20] Trey Herr, June Lee, William Loomis, and Stewart Scott. 2020. Breaking Trust: Shades of Crisis Across an Insecure Software Supply Chain. https://www.atlanticcouncil.org/in-depth-research-reports/report/breaking-trust-shades-of-crisis-across-an-insecure-software-supply-chain/. Accessed: 2020-07-30.

[21] Daniel Holth. 2012. PEP 427 – The Wheel Binary Package Format 1.0. https://www.python.org/dev/peps/pep-0427/. Accessed: 2020-10-10.

[22] Maya Kaczorowski. 2020. Secure at every step: What is software supply chain security and why does it matter? https://github.blog/2020-09-02-secure-your-software-supply-chain-and-protect-against-supply-chain-threats-github-blog/.

[23] Andrey Kuyan, Sergey Gusev, Andrey Kozlov, Zhanibek Kaimuldenov, and Evgeny Kravtsunov. 2013. Experience of Building and Deployment Debian on Elbrus Architecture. In *Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering*.

[24] Chris Lamb and Stefano Zacchiroli. 2021. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* (2021). https://doi.org/10.1109/MS.2021.3073045

[25] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting third-party libraries: The practitioners' perspective. In *Proc. of the 28th ACM European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*. 245–256.

[26] Kim Lewandowski and Mark Lodato. 2021. Introducing SLSA, an End-to-End Framework for Supply Chain Integrity. https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html.

[27] Lutoma. 2019. PSA: There is a fake version of this package on PyPI with malicious code. https://github.com/dateutil/dateutil/issues/984. Accessed 6 February 2020.

[28] Microsoft. 2019. Microsoft ApplicationInspector: A source code analyzer. https://github.com/microsoft/ApplicationInspector. Accessed 21 February 2020.

[29] Microsoft. 2020. OSS Gadget: Collection of tools for analyzing open source packages. https://github.com/microsoft/OSSGadget.

[30] Maximilian Nothe. 2021. Who has already dropped Python 2 support? https://maxnoe.github.io/who-dropped-python2/.

[31] National Institute of Standards and Technology (NIST). 2020. Security and Privacy Controls for Federal Information Systems and Organizations, SP 800-53, Revision 5, September 2020. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf. Accessed 21 Feb 2021.

[32] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. 2020. If You've Seen One, You've Seen Them All: Leveraging AST Clustering Using MCL to Mimic Expertise to Detect Software Supply Chain Attacks. *arXiv preprint arXiv:2011.02235* (2020).

[33] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Proc. of DIMVA*. Springer, 23–43. https://doi.org/10.1007/978-3-030-52683-2_2

[34] Marc Ohm, Arnold Sykosch, and Michael Meier. 2020. Towards detection of software supply chain attacks by forensic artifacts. In *Proceedings of ARES'20*. 1–6. https://doi.org/10.1145/3407023.3409183

[35] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1513–1531. https://doi.org/10.1145/3372297.3417232

[36] Brian Pfretzschner and Lotfi ben Othmane. 2017. Identification of Dependency-based Attacks on Node. js. In *Proc. of ARES'17*. 1–6. https://doi.org/10.1145/3098954.3120928

[37] Nikita Popov. 2021. Changes to Git commit workflow. https://news-web.php.net/php.internals/113838.

[38] PyCQA. [n.d.]. Security oriented static analyser for python code. https://pypi.org/project/bandit/.

[39] David Saff and Michael D Ernst. 2003. Reducing wasted development time via continuous testing. In *Proc. of ISSRE 2003*. IEEE, 281–292. https://doi.org/10.1109/ISSRE.2003.1251050

[40] Slovak. 2019. skcsirt-sa-20170909-pypi. https://www.nbu.gov.sk/skcsirt-sa-20170909-pypi/. Accessed 6 February 2020.

[41] Sonatype. 2019. 2019 State of the Software Supply Chain Report Reveals Best Practices From 36,000 Open Source Software Development Teams. https://www.sonatype.com/press-release-blog/2019-state-of-the-software-supply-chain-report-reveals-best-practices-from-36000-open-source-software-development-teams.

[42] Steve Stagg. 2017. Building a botnet on PyPi. https://hackernoon.com/building-a-botnet-on-pypi-be1ad280b8d6. Accessed: 2020-2-11.

[43] Synopsys. 2020. Synopsys 2020 Open Source Security and Risk Analysis Report. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.pdf.

[44] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending Against Package Typosquatting. In *International Conference on Network and System Security*. Springer, 112–131. https://doi.org/10.1007/978-3-030-65745-1_7

[45] Nikolai Philipp Tschacher. 2016. *Typosquatting in programming language package managers*. Ph.D. Dissertation. Universität Hamburg, Fachbereich Informatik.

[46] Hugo van Kemenade. 2021. Top PyPI Packages. https://doi.org/10.5281/zenodo.4486832

[47] Laurie Voss. 2018. npm and the future of JavaScript. https://slides.com/seldo/npm-future-of-javascript.

[48] Duc-Ly Vu. 2020. A fork of Bandit tool with patterns to identifying malicious python code. https://github.com/lyvd/bandit4mal.

[49] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Towards Using Source Code Repositories to Identify Software Supply Chain Attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2093–2095. https://doi.org/10.1145/3372297.3420015

[50] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and Combosquatting Attacks on the Python Ecosystem. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. https://doi.org/10.1109/EuroSPW51379.2020.00074

[51] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2021. LastPyMile Replication Package. https://doi.org/10.5281/zenodo.4899935

[52] Warehouse. 2020. Malware Checks. https://warehouse.readthedocs.io/development/malware-checks/#malware-checks.

[53] Warehouse. 2020. Warehouse codebase. https://warehouse.readthedocs.io/application.html.

[54] Joshua Wehner. 2016. Working with submodules. https://github.blog/2016-02-01-working-with-submodules/.
[55] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 995–1010.