IMT SCHOOL FOR ADVANCED STUDIES LUCCA

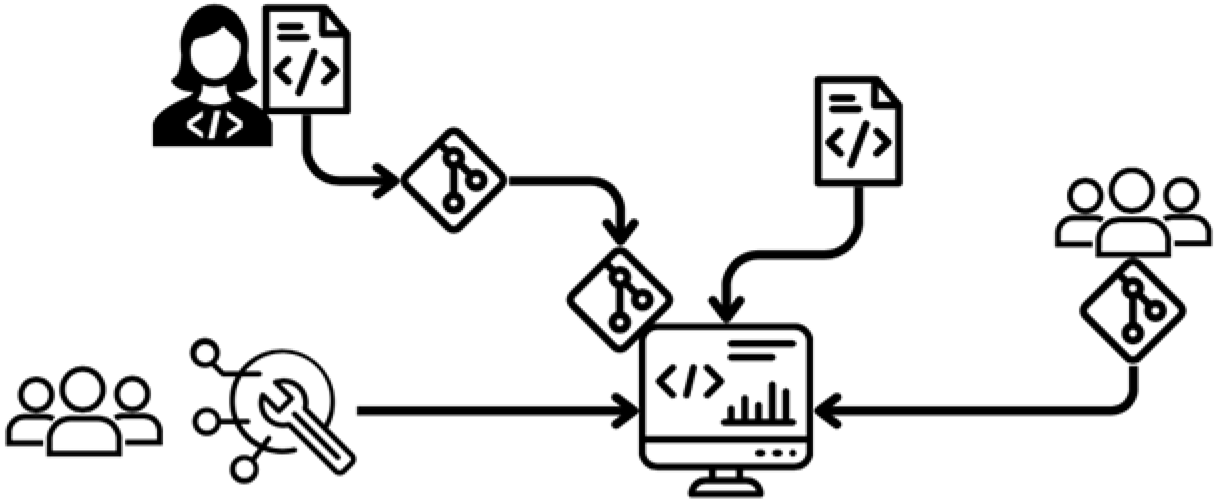
SBOM Generation Tools in the Python Ecosystem: an In-Detail Analysis

Serena Cofano, *PhD Student at IMT and University of Genoa*
Giacomo Benedetti, *Research Assistant at National Research Council of Italy*
Matteo Dell'Amico, *Assistant Professor at University of Genoa*

General Context

Context - Software Supply Chain

Software Supply Chain (SSC) refers to the **collection of devices, systems, and people** involved in creating and delivering **final software**.



Context - Software Supply Chain Security

The security of the Software Supply Chain is the security of its components.

How do we ensure Software Supply Chain Security?

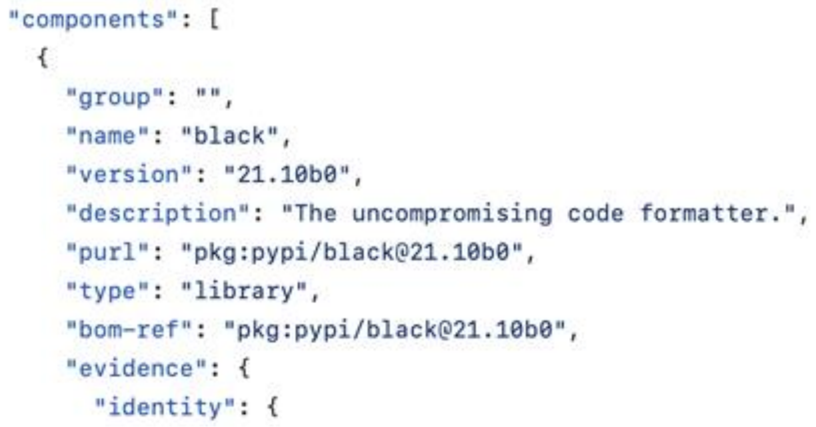
- **Transparency** → knowledge of the components that make the Supply Chain → security analysis on them.

How do we get the list of components present in the Supply Chain?

- **Software Bill Of Material (SBOM)**

Context - SBOM Definition

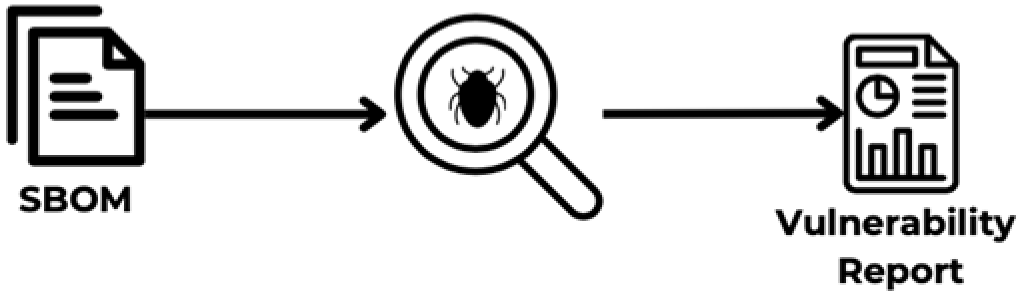
Software Bill of Materials (SBOM) is a **detailed inventory** of all **components**, **libraries**, and **dependencies** used in a software application, including their versions, sources, and licenses.



```
"components": [
  {
    "group": "",
    "name": "black",
    "version": "21.10b0",
    "description": "The uncompromising code formatter.",
    "purl": "pkg:pypi/black@21.10b0",
    "type": "library",
    "bom-ref": "pkg:pypi/black@21.10b0",
    "evidence": {
      "identity": {
```

Context - SBOM for Security

The **SBOM** is used as input for tools that search for vulnerabilities.



Context - SBOM Generation

How can be generated an SBOM?

- **Statically** → metadata parsing
- **Dynamically** → installation simulation, runtime monitoring, instrumentation …

SBOM generation properties

Completeness: the capability to include all the components present in the software.

Correctness: the capability to provide the exact information about them.

Research Problem

Research Problem

- Low quality of the generated SBOMs → lack of **completeness** and **correctness**
    - **Lack of completeness → false negatives** in security analysis
    - **Lack of correctness → false positives** and **false negatives** in security analysis
- Low reliability on the SBOM as a useful tool to enhance transparency
- Lack of deeper investigation of the causes that bring to lack of completeness and correctness of the SBOM

→ Insufficient support in tools that generate the Software Bill Of Material

Main Goal

Main Goal

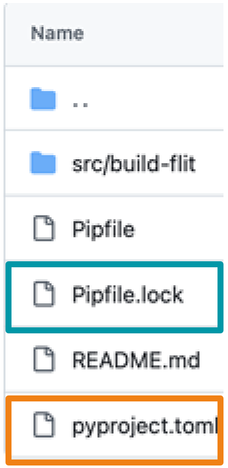
Investigation of the issues and related causes affecting SBOM **completeness** and **correctness**, focusing on the **Python ecosystem**.

In detail the contributions are :

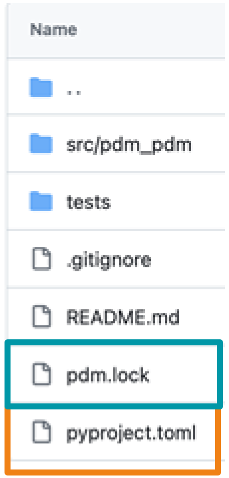
- A study on the impact of the **Python ecosystem** on the generation of SBOMs.
- A study on how the approach used by **SBOM generation tools** changes the final output of the SBOM.

Why Python?

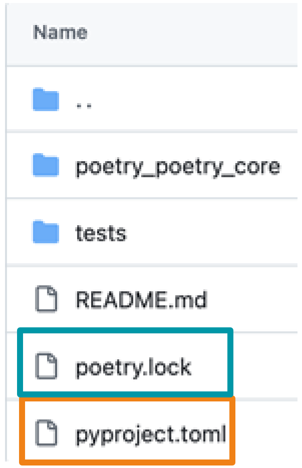
Flexibility → projects can be generated using **different package managers** → **different files** can be present in the project and **dependencies can be differently declared.**



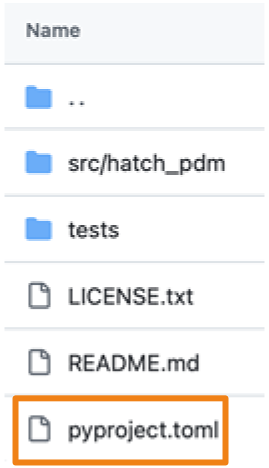
Name
..
src/build-flit
Pipfile
Pipfile.lock
README.md
pyproject.toml



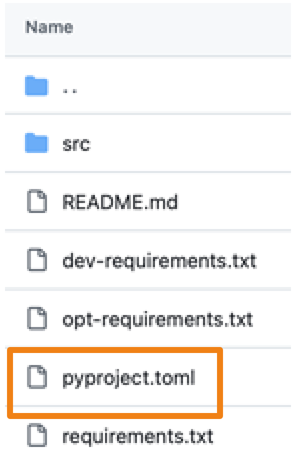
Name
..
src/pdm_pdm
tests
.gitignore
README.md
pdm.lock
pyproject.toml



Name
..
poetry_poetry_core
tests
README.md
poetry.lock
pyproject.toml

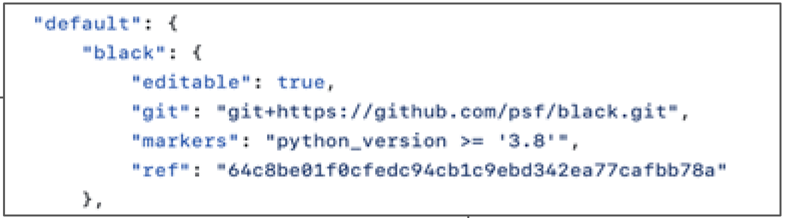


Name
..
src/hatch_pdm
tests
LICENSE.txt
README.md
pyproject.toml

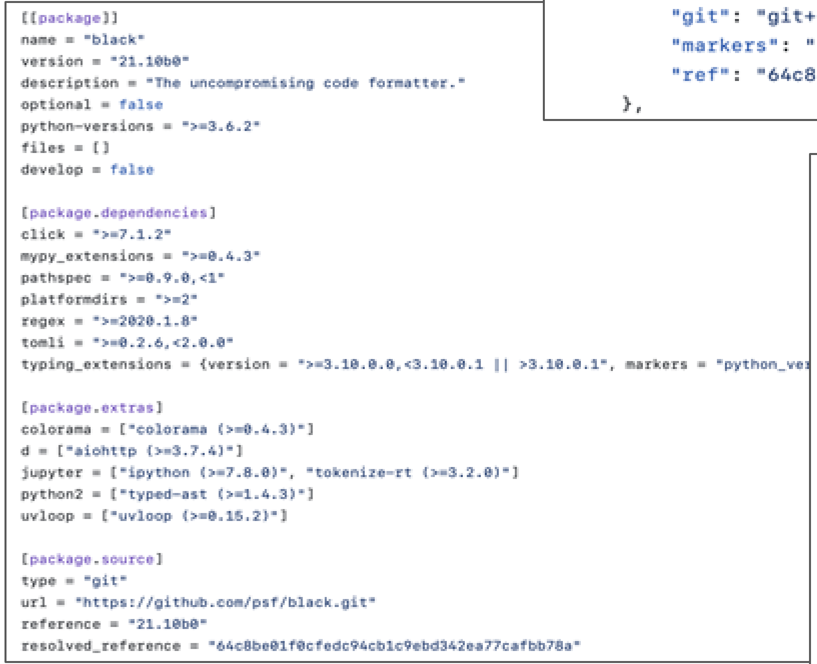


Name
..
src
README.md
dev-requirements.txt
opt-requirements.txt
pyproject.toml
requirements.txt

Why Python?



```
"default": {
    "black": {
        "editable": true,
        "git": "git+https://github.com/psf/black.git",
        "markers": "python_version >= '3.8'",
        "ref": "64c8be01f0cfedc94cb1c9ebd342ea77cafbb78a"
    },
```

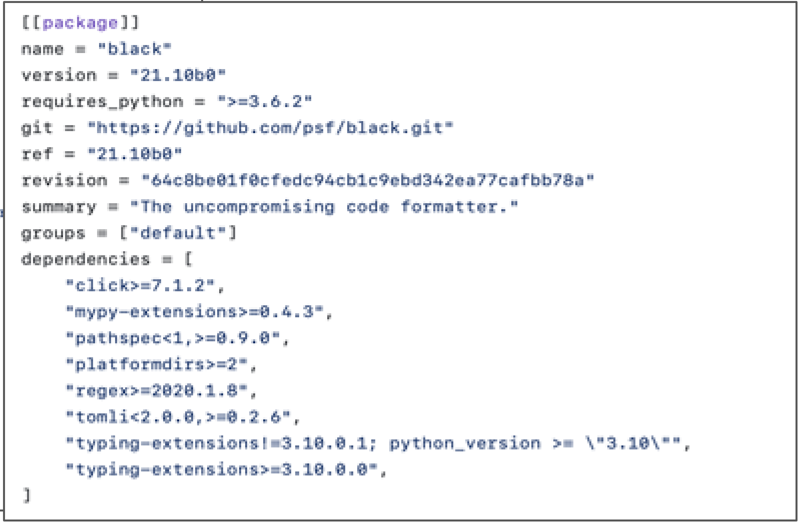


```
[[package]]
name = "black"
version = "21.10b0"
description = "The uncompromising code formatter."
optional = false
python-versions = ">=3.6.2"
files = []
develop = false

[package.dependencies]
click = ">=7.1.2"
mypy_extensions = ">=0.4.3"
pathspec = ">=0.9.0,<1"
platformdirs = ">=2"
regex = ">=2020.1.8"
tomli = ">=0.2.6,<2.0.0"
typing_extensions = {version = ">=3.10.0.0,<3.10.0.1 || >3.10.0.1", markers = "python_ver

[package.extras]
colorama = ["colorama (>=0.4.3)"]
d = ["aiohttp (>=3.7.4)"]
jupyter = ["ipython (>=7.8.0)", "tokenize-rt (>=3.2.0)"]
python2 = ["typed-ast (>=1.4.3)"]
uvloop = ["uvloop (>=0.15.2)"]

[package.source]
type = "git"
url = "https://github.com/psf/black.git"
reference = "21.10b0"
resolved_reference = "64c8be01f0cfedc94cb1c9ebd342ea77cafbb78a"
```



```
[[package]]
name = "black"
version = "21.10b0"
requires_python = ">=3.6.2"
git = "https://github.com/psf/black.git"
ref = "21.10b0"
revision = "64c8be01f0cfedc94cb1c9ebd342ea77cafbb78a"
summary = "The uncompromising code formatter."
groups = ["default"]
dependencies = [
    "click>=7.1.2",
    "mypy-extensions>=0.4.3",
    "pathspec<1,>=0.9.0",
    "platformdirs>=2",
    "regex>=2020.1.8",
    "tomli<2.0.0,>=0.2.6",
    "typing-extensions!=3.10.0.1; python_version >= \"3.10\"",
    "typing-extensions>=3.10.0.0",
]
```

Why Python?

Flexibility → projects can be generated using **different package managers** → **different files** can be present in the project and **dependencies can be differently declared.**

Dynamic resolution of dependencies → resolution of dependencies' versions at installation time.

- E.g., `numpy>=x.y, numpy, numpy== X.*`

Methodology

How do we conduct our study?

Methodology

Experimental Setup

- Dataset creation
  - **Package managers** selection
  - **Dependencies** selection
- **SBOM generation tools** selection

SBOM generation

- SBOM generation tools **execution**

Analysis

- SBOM analysis → searching for **issues**
- Investigation of the **causes**
  - In SBOM generation tools
  - In Python ecosystem

Methodology - Experimental Setup

SBOM Generation Tools

- Trivy
- Syft
- Cdxgen
- Ort

Python Package Managers

- Poetry
- Hatch
- Pip
- Pipenv
- Pdm

Methodology - Experimental Setup

Name	Version	Type
numpy	Unversioned	Imported and used in the code
docopt	Pinned	Remote
black	Pinned	Remote from Git
seaborn	Pinned	Not used in the code
matplotlib	Constrained	Not used in the code
urllib3	Unversioned	Not used in the code
nltk	Unversioned	Optional dependency
pytest	Unversioned	Development dependency

Methodology - Analysis

- *SBOM Analysis*: **Manual analysis** on the generated SBOMs
  - **Completeness** → search for **missing** dependencies
  - **Correctness** → search for **wrong** dependencies (wrong or missing version)
- *Investigation of the causes*:
  - **Code inspection** of the SBOM generation tools.
  - **Tool documentation study.**
  - **Community reaction** assessment (e.g., Github issues analysis).
  - Identification of **different tools behaviours** related to **different package managers.**

Results
What do we get?

Results - Issues

Version Management Issues

Process of **assigning a version to a package**. If the version is explicit, the SBOM generation tools should detect it; otherwise, they should determine it.

Metadata Files Handling Issues

Metadata files management includes **identifying** them, **parsing** them, or **using them to install** dependencies in a simulated environment.

Results - Causes

Version Management Issues

Completeness

- Missing implementation of solving techniques for unpinned dependencies.

For instance:

- In Trivy and Syft, if a dependency is declared without version, it is ignored.

Correctness

- Inaccurate "guessing" techniques for versions.

For instance:

- Syft converts `>=` in `==`
- Cdxgen converts constrained dependencies with `>=`, in dependencies with latest version.

Results - Causes

Metadata Files Management Issues

Completeness

- Missing implementation for parsing a metadata file (i.e., Pyproject.toml, requirements.txt, lockfiles).
- Missing metadata file in the Python project.

For instance:

- Hatch does not provide a lock file.
- Pdm.lock is parsed only by few tools

Correctness

- Incorrect parsing of metadata files.
- Not standardized format for metadata files.

For instance:

- Pipfile.lock does not report in a specific field the version of the remote dependency

Discussion
What can we conclude?

Conclusion and Discussion

Defects in **SBOM generation tools**

- Inaccurate version-solving techniques
- Missing parsing of pyproject.toml
- No warnings about incompleteness and incorrectness

Lack of standard in **Python**:

- Unpredictable structure of the projects
- Unpredictable structure of the metadata files

Recommendations

- For the Python Ecosystem:

  → It should push for initiatives proposing standards.
- For the SBOM generation tool's developers:

  → SBOM generation tools are required to consider the new Python standard build interface by parsing the pyproject.toml file.

Questions?

Thank you for your attention