UNIVERSITY OF GENOVA

PHD PROGRAM IN SECURITY, RISK, AND VULNERABILITY
XXXVII CYCLE

Curriculum: *Cybersecurity and Reliable Artificial Intelligence*

Improving Transparency, Trust, and Automation in the Software Supply Chain

by

Giacomo Benedetti

Prof. Luca Verderame	Supervisor
Prof. Serena Cattari	Head of the PhD program
Prof. Luca Oneto	Head of the PhD curriculum

Thesis Jury:

Prof. Gabriele Costa, *IMT School for Advanced Studies Lucca*	External examiner
Dott. Enrico Mariconti, *University College London*	External examiner

Dibris

Department of Informatics, Bioengineering, Robotics and Systems Engineering

Abstract

Modern software no longer functions as a single, monolithic entity but rather as an intricate composition of thousands of components. These components collectively contribute to the functionality and security of software, yet they also expose the ecosystem to an ever-growing array of vulnerabilities. High-profile incidents such as SolarWinds and the 2024 compromise of xz-util highlight the pervasive and severe nature of threats in the software supply chain. These attacks are often opaque, making detection and mitigation complex and slow, and they possess a significant amplification factor: a single compromised component can propagate downstream, impacting many dependent systems and users.

Addressing these challenges requires a focused investigation into the security of software supply chains. Governments, industry stakeholders, and community initiatives have emphasized the importance of secure, transparent, and trusted software ecosystems. However, the lack of appropriate tools and methodologies hinders the practical application of recommendations and guidelines. Developers need solutions capable of managing the complex networks that underpin modern software. This necessitates transparency within the supply chain, which can either be proactively ensured by developers or retroactively derived through artifact analysis. Unfortunately, even the most diligent developers remain vulnerable to last-mile compromises, as demonstrated by the xz-util attack, where malicious code was injected into the distributable artifact without detection.

This thesis addresses three critical aspects of the software supply chain to enhance security: transparency, trust, and automation. First, it investigates transparency as a mechanism to empower developers with accurate and complete insights into the software components integrated into their applications. To this end, the thesis introduces SUNSET and PIP-SBOM, leveraging modeling and SBOMs (Software Bill of Materials) as foundational tools for transparency and security. Second, it examines software trust, focusing on the effectiveness of reproducible builds in major ecosystems and proposing solutions to bolster their adoption. Finally, it emphasizes the role of automation in modern software management, particularly in ensuring user safety and application reliability. This includes developing a tool for auto-

mated security testing of GitHub Actions and analyzing the permission models of prominent platforms like GitHub, GitLab, and BitBucket.

By advancing these pillars of software supply chain security, this thesis provides the tools, methodologies, and insights necessary for practitioners to navigate and secure the increasingly complex and interconnected software landscape.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

The acceleration of digital innovation is driven by the widespread adoption of reusable software abstractions such as libraries, frameworks, cloud infrastructure, and artificial intelligence modules. This new paradigm, often referred to as the *software supply chain*, has revolutionized the software industry. Modern software projects are built upon and interdependent on other projects, creating a complex ecosystem. However, this interdependency has introduced significant risks, with the software supply chain becoming a primary target for cyberattacks.

Attacks within this domain are known as *software supply chain attacks*, where third-party components, such as libraries and dependencies, are exploited to compromise software systems. Unlike traditional software threats, which focus on vulnerabilities within standalone software, supply chain attacks exploit the interconnected nature of software ecosystems. Since 2022, the number of packages with malicious dependencies increased. Well-known incidents such as SolarWinds [121], Log4j [165], and the compromise of XZ Utils [58, 112] have affected thousands of organizations worldwide, highlighting the global impact of these attacks.

In response to this growing threat, governments and regulatory bodies have implemented policies to enhance software supply chain security. The US Executive Order (EO) 14028 [38] in 2021 and the European Union's Cybersecurity Resilience Act (CRA) in 2022 [79] both emphasize the need for greater transparency and integrity in open-source software to mitigate risks and enhance the security of the supply chain.

Complexity of Software Supply Chain Threats The field of software supply chain security is vast, touching upon various aspects of software engineering and often overlapping with long-standing cybersecurity challenges. One of the key contributions to understanding

this complexity comes from the work of Ladisa et al. [131], who developed a comprehensive taxonomy of threats and mitigation techniques. This taxonomy categorizes threats into three main types: 1) Developing or advertising malicious packages from scratch, 2) Creating name confusion with legitimate packages, 3) Subverting legitimate packages.

While the first category involves the creation and distribution of malware, which is a well-known problem, it takes on a new dimension in the context of the software supply chain. Even without directly installing a malicious package, users can inherit malicious code through transitive dependencies — dependencies of a dependency. The second category deals with vulnerabilities related to package metadata, and the third, perhaps the most critical, encompasses attacks that subvert legitimate packages at various points in the software lifecycle, such as securing the source code, build, or distribution phases.

Software Supply Chain Security The primary focus of this Thesis is on identifying the threats targeting the software supply chain and proposing effective remediation strategies.

Since the software is no longer developed in a monolithic fashion, traditional code analysis techniques are no longer sufficient for security. As software is developed through a supply chain, security must be provided along the process. Attack vectors can have different shapes, targeting specific weaknesses of the process.

The software development lifecycle can be abstracted into two phases. The *software composition*, when all the software dependencies are collected, and the *software build*, when the software code and dependencies are included in a distributable artifact. Looking at this model, two properties are necessary to enable security: *transparency* of the software composition, and *software trust* during the build process. A third property is necessary because of the fast pace required for development and production, secure *automation*.

Specifically, dependencies can be the carrier of malicious behavior injected by an attacker and they can be used to attack upstream projects. *Transparency* is a fundamental property to facilitate developers to understand the direct and transitive dependencies of the software. Malicious dependencies have become particularly common because of typo-squatting, dependency confusion, and project take-over attacks. In the last years, the Software Bill of Materials (SBOM) was introduced to distribute information about the software composition trustfully. Both the United States of America and the European Community targeted the SBOM as an enabling element of software supply chain transparency [38, 79]. Thus, SBOM generation is the starting phase of Software Composition Analysis (SCA), aiming to identify vulnerabilities inside of dependencies in a software supply chain. This is not the only usage

the SBOM is envisioned for, for example, license analysis is another important task where this artifact can be used.

Build infrastructure transforms software's code into actually deployable artifacts. Malicious code can be injected through it. This attack vector can be particularly insidious because it occurs at the end of the development life cycle. That is, malicious code injected at this point of development can be hardly spotted, since it becomes part of the distributable artifact. *Software trust* is a fundamental element necessary in the software supply chain to support developers, make the development process smooth, and enable security, preventing software compromise in the last mile between development and deployment. Finally, all the operations in the software development life cycle need to be supported through automation. Without automation, the entire supply chain of software would not be maintained. Automation technologies, such as GitHub Actions, help to manage the build, testing, and deployment phases of modern software development. Yet, this technology also adds potential security flaws. Enabling secure *automation* technologies is another important step toward software supply chain security.

1.1 Research Contribution

This Thesis is part of the ongoing effort to secure the software supply chain. By addressing both detection and mitigation of dependency manipulation at different stages of the software development lifecycle (SDLC), it aims to contribute to the body of knowledge on protecting the integrity of modern software ecosystems. The Thesis mainly contributes to three properties of software supply chain security: *Transparency*, *Software Trust*, and *Secure Automation*.

Transparency This is an enabling property of a secure software supply chain. Having *complete* and *correct* information about the supply chain components allows users to make decisions and prevent vulnerabilities. This Thesis proposes two contributions in this direction: (1) a methodology to automatically rebuild the software supply chain from source code; (2) an evaluation of the impact SBOM generation tools on dependency security assessment, and a proof of concept of a generation tool to obtain better SBOMs for security assessments.

The first contribution explores alternatives to the SBOM to represent the software supply chain. Thus, the SBOM requires active collaboration from all the parts developing components of a software supply chain. This is not always granted, and a developer may need methods to retrieve the supply chain composing its software. Moreover, SBOM models only

software components, while there are other types of elements, such as package managers, automation technologies, version control systems, and also the developers themselves. This Thesis proposes SUNSET, a methodology to rebuild the software supply chain starting from a project source code, and aggregate information security, considering the relationships among elements.

The second contribution investigates the current use of SBOM for dependency security assessment in the Python ecosystem. Since state-of-the-art tools for SBOM generation in Python resulted to be insufficient in proving an accurate enough SBOM for vulnerability assessment tools, this Thesis proposes PIP-SBOM. This tool shows how vulnerability assessment task performance greatly improves by using tools using the native dependency resolution process adopted by software ecosystems.

Software Trust Developers need to trust the software they rely on. The build is, usually, the last step before a package distribution. This means that injecting malicious content in the package in this phase makes useless all the security measures applied before in the software development life cycle.

Reproducible builds represent a solution to this problem, however achieving them is not as easy. This Thesis conducts an empirical analysis of open-source packaging ecosystems state w.r.t. reproducible builds. These ecosystems differ in the number of reproducible builds they can achieve. The Thesis challenges the idea that reproducible builds are a hardly-achievable property, showing that most ecosystems have already managed to achieve them, and, for those that are not there yet, it is possible to modify the package manager to enable reproducible builds without requiring actions from the developers. Understanding the causes of non-reproducibility caused by package managers is a fundamental step to enabling reproducible builds in software ecosystems.

Secure Automation Code repositories are central to the development of software, they contain not only code but functionalities to manage operations such as builds, testing, and tracking issues. The Thesis contribution focuses on (1) the analysis of security issues in GitHub Actions; and (2) the study of the permission life cycle in the major software management platforms.

The first contribution reports of many GitHub Actions vulnerable to severe vulnerabilities, and proposes the GitHub Actions Security Testing (GHAST) tool. This tool is validated against in-the-wild GitHub Actions workflows collected with SUNSET. The experimental analysis shows that GitHub Actions are not secure at all, containing some serious vulnerabili-

ties — e.g., command injection — and a huge number of misconfigurations, that may enable attacks — e.g., badly defined permissions.

The second contribution analyses the approach of the three main software management platforms — i.e., GitHub, GitLab, and BitBucket — to permission granting. The Thesis provides the model for a permission life cycle to provide such an analysis. With the vast number of software packages daily released, procedures must be automated whenever possible. This means authorizing third-party entities to deal with the entire software development life cycle. Authorization represents an efficient mitigation to prevent unexpected behaviors.

1.2 Thesis Structure

This Thesis is composed of eight chapters.

Chapter 2 provides the background necessary to understand the main contributions of this Thesis. It describes the structure of a software supply chain (Section 2.1) focusing on the vulnerabilities and attacks (Section 2.1.1). Section 2.3 describes the functionalities of CI/CD automation, focusing on GitHub Actions (Section 2.3.1) and security issues affecting them (Section 2.3.2). Section 2.4 provides background on software ecosystems (Section 2.4.1), packages distribution models (Section 2.4.2), and the challenges of making builds reproducible (Section 2.4.3).

The research contribution of this Thesis is divided into three parts, matching the main areas of contribution discussed in Section 1.1.

Part I has two chapters discussing the need of transparency in the software supply chain and the contribution of this Thesis. Chapter 3 discusses the possibility of modeling the software supply chain as a graph where nodes are entities and edges their relationships. Entities composing the SSC can be either functional components — e.g., software libraries — or structural components — e.g., package managers. A complete risk assessment of this structure means thoroughly navigating the whole graph and aggregating security information to produce a security score at different levels of granularity. It is possible to grasp the complexity and scaling of a software supply chain from this Chapter. This Chapter is based on the paper ”Alice in (Software Supply) Chains: Risk Identification and Evaluation“ published at the International Conference on the Quality of Information and Communications Technology (QUATIC) 2022, in collaboration with Alessio Merlo and Luca Verderame of the University of Genoa

Transparency is given by knowledge of the components of the supply chain. The Software Bill of Materials is a promising technology that may enable full transparency in the software

supply chain. Chapter 4 discusses the use of the SBOM for vulnerability assessment of Python applications. The results fo this research show that state-of-the-art SBOM generation tools are not ready to produce SBOMs accurately enough to produce usable security reports. Moreover, a novel methodology is proposed through the implementation of PIP-SBOM, an extended version of the Package Installer for Python (PIP). This tool is capable of generating an SBOM that can be actually used by a vulnerability scanner to produce an accurate vulnerability assessment. This Chapter is based on the paper ”The Impact of SBOM Generators on Vulnerability Assessment in Python: A Comparison and a Novel Approach“ accepted at the Conference on Applied Cryptography and Network Security (ACNS) 2025, in collaboration with Serena Cofano of the IMT School for Advanced Studies Lucca, Alessandro Brighente and Mauro Conti of the University of Padua.

Part II has a chapter discussing the need of secure build processes.

Chapter 5 investigates the state of reproducible builds in six widely used software ecosystems. This Chapter is based on the paper “An Empirical Study on Reproducible Packaging in Open-Source Ecosystems” accepted at the International Conference on Software Engineering (ICSE) 2025, in collaboration with Oreofe Solarin of the Case Western Reserve University, Greg Tyschal, William Enck, Alexandros Kapravelos of the North Carolina State University, Courtney Miller, Christian Kästner of Carnegie Mellon University, and Alessio Merlo, Luca Verderame of the University of Genoa.

Part III has two chapters discussing the security problems of automation in the software development life cycle and which countermeasures can be adopted. This Chapter is based on the papers ”Automatic Security Assessment of GitHub Actions Workflows“ published at the Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED) 2022 co-located with ACM CCS and ”A Preliminary Study of Privilege Life Cycle in Software Management Platform Automation Workflows“ published at the DevSecOps Research and Opportunities (DevSecOpsRO) 2023 co-located with IEEE EuroS&P. Both these works were in collaboration with Alessio Merlo and Luca Verderame of the University of Genoa.

Chapter 8 concludes this Thesis with some future directions for secure software supply chain research.

Chapter 2

Software Supply Chain Security: Background & Protection techniques

2.1 Software Supply Chain

The umbrella term *software supply chain* refers to the set of elements composing a software coming from sources external to the software itself. That is, each component of the software and its Software Development Life Cycle (SDLC).

In the software supply chain, we can identify all the elements, called *assets*, that contribute to the development of a software artifact, namely the final product. These assets include structural elements (e.g., code repositories and development servers), software components (e.g., libraries and executables), and organizational entities (e.g., developers and software maintainers). Inside the software supply chain, we can distinguish between *supplier assets* and *customer assets*. The former contains all assets not explicitly created or defined for the final product, e.g., an external library or package managers. The latter comprises assets the final software will interface with once deployed in the production environment.

In a typical DevOps scenario, depicted in Figure 2.1, the software supply chain provides assets for the pre-release phase, where the organization selects (plan), implements (code), packs (build), and tests (test) all the elements composing the final software.

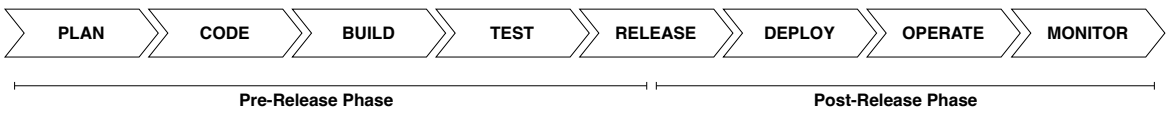


Figure 2.1 The DevOps Workflow.

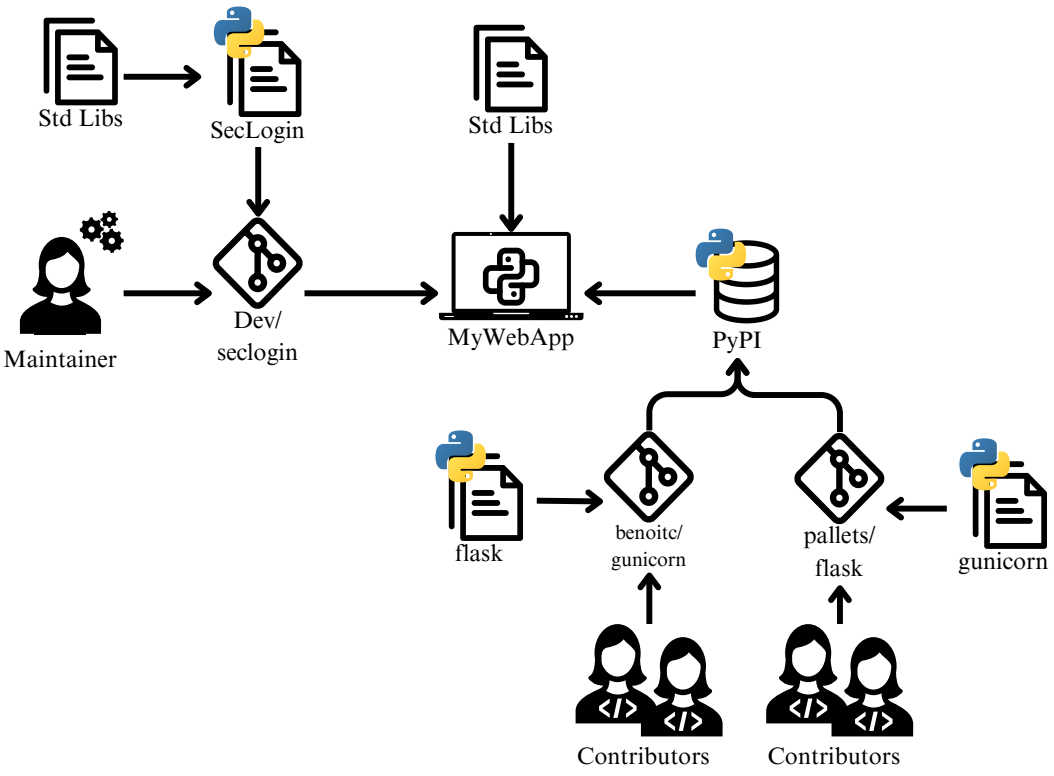


Figure 2.2 *MyWebApp* Software Supply Chain.

Figure 2.2 depicts an example of supplier assets composing the SSC of a Python web app called *MyWebApp*. *MyWebApp* uses a set of standard python libraries (e.g., `os` and `glob`) and two external modules (i.e., `flask` and `gunicorn`) imported using the PyPI package manager. Both modules are hosted on public GitHub repositories, where maintainers and contributors provide regular updates and new functionalities. Also, *MyWebApp* manually imports another library, called `SecLogin`. This library is developed and hosted on a GitLab repository by a single maintainer and relies on standard Python libraries as well.

2.1.1 Software Supply Chain Vulnerabilities and Attacks

A *software supply chain vulnerability* is defined as a security vulnerability affecting an asset that could evolve into an attack once exploited. Such vulnerability may occur at different stages of software development. In particular, Common Weakness Enumeration [190], i.e., CWE, highlights that 91% of security weaknesses are introduced during design (462 CWEs) and implementation (724 CWEs).

One of the most significant advantages of attack campaigns targeting software supply chains is that their impact is not limited to the final software. They also harm assets belonging to software supply chains other than the target ones. As a result, this form of attack is more likely to go unnoticed and deliver a higher payout to the attacker [210].

Supporting the importance of the security evaluation of software supply chains, the MITRE ATT&CK framework [196], identified the *supply chain compromise* as an initial access tactic.

Let us consider the supply chain of the *MyWebApp* software as in Figure 2.2. If the maintainer introduces a Static Application Security Testing (SAST) tool, during the build phase of the final software, to evaluate *MyWebApp*, the analysis may identify some security vulnerabilities and map them to known vulnerabilities — i.e., CVEs — and to the respective CWEs. In our example, the code analyzer detects CWE-20 (Improper Input Validation) [191], CWE-89 (Improper Neutralization of Special Elements used in an SQL Command) [194], and CWE-798 (Use of Hard-coded Credentials) [193].

Thanks to the SAST security report, the developers of *MyWebApp* can patch the source code. However, vulnerabilities are far more complex than the know-vulnerabilities a SAST can report, and developers have limited information on the asset and the precise attack vector that enabled the security flaw. Since vulnerability assessment techniques do not provide developers with enough information to take proper actions to mitigate the vulnerability, often causing either overacting or underacting to remote the vulnerability alert.

For instance, the vulnerability assessment tools cannot detect that the credentials of one of the contributors of the GitLab repository of the SecLogin module have been compromised using a social engineering attack. Indeed, thanks to this attack vector, attackers can inject malicious code into the module and, thus, the software supply chain, reaching the final software. This lack of information prevents developers readiness to operate for a mitigation of the vulnerability. Decoupling the SecLogin component is not an option, since it would break the functional operations of the final software. With a proper informative system, they would know that detaching the component from the production repository solve the security issue.

2.2 Software Bill of Materials (SBOM)

This section provides the necessary background on the SBOM generation process (Section 2.2.1), the dependency management and resolution for Python (Section 2.2.2), and the usage of SBOMs by vulnerability scanners (Section 2.2.3).

2.2.1 SBOM Generation Process

The SBOM partially provides a solution to the problem discussed above in Section 2.1.

An SBOM contains information on the components of a software. There are different standards regulating which information is required in the document and according to which format they must be included. The most used standards are CycloneDX and System Package Data Exchange (SPDX).

An SBOM is generated using tools commonly known as SBOM generation tools. Differently from SCA tools, a SBOM generation tool does not analyze licenses and the security posture of components in the software under scrutiny. However, it may be part of the SCA, providing inputs for further analysis of the identified assets. SBOM generation tools take as input the software's project folder and produce a list (i.e., the SBOM) of software components and their dependencies, along with their version and other information useful to trace the software composition.

SBOM generation tools analyze the structure of a package or project and produce an SBOM. How the SBOM is generated depends on the generation approach implemented by the SBOM generation tool. Most tools rely on static metadata-based generation methods. However, previous research [211, 163] reported that SBOMs have scarce accuracy when generated by tools currently implementing this approach. Other tools, such as cdxgen, aim to reproduce an installation environment where dependencies are collected according to metadata files. NTIA established the minimum required elements for an SBOM [149]. Tools such as sbom-scorecard [12] can quantify the level of compliance. However, concerning vulnerability assessment, the required elements include only dependency identifiers — i.e., name, version, and package URL (purl).

2.2.2 Dependencies Management in Python

Python offers a good example of how the dependency management process happens. The following process can be adapted to other ecosystems with few variations. Python has multiple package managers that a developer can choose to deal with dependencies and other project management operations. Each package manager chooses how to deal with the project's filesystem to coordinate the dependency management. Depending on the package manager, the project may result in very different filesystem structures.

A Python project should contain either a `setup.py` or a `pyproject.toml` file to be managed by a package manager. Both these files contain the project's metadata and list the dependencies required to properly build and operate the project. The project's build

produces an artifact, a wheel or source distribution, containing the source code files and all the additional files required in the metadata file.

A distributable artifact is installed through PIP. During installation, the dependencies listed in the metadata file are collected and installed on the user's system. Since transitive dependencies — i.e., dependencies of a dependency — are not shipped together with the distributable artifact, PIP uses the `resolvelib` package implementing a specific algorithm for dependency resolution [158].

Dependencies are listed in the metadata files by name and version. The dependency's version can be either pinned, non-pinned, or omitted — i.e., not specified at all, in this case, the package manager collects the dependency to the latest stable and available version. PyPI allows for multiple versioning schemas [9], such as semantic versioning [162], calendar versioning [119], and their combinations. The pinning vs. non-pinning choice is left to the developer by using the '==' operator vs. using range operators — i.e., <=, >=, <, >, !=.

2.2.3 Vulnerabilities Scanning with SBOM

A *vulnerability scanner* is a tool that analyzes a software artifact and provides a *security report* listing potential vulnerabilities affecting the scanned product.

Analyzing a software's dependency network is not an easy task. Modern software vastly relies on third-party software, resulting in the size of the dependency network rapidly increasing. Vulnerability databases, such as NVD and OSV, contain entries for known software vulnerabilities, making the dependency network analysis easy and fast. SBOMs enable vulnerability scanners to check the presence of known vulnerabilities without retrieving the dependency list. Currently largely used vulnerability scanners — e.g., ShiftLeftScan [13], Grype [19], KubeClarity [8], Bomber [2] — use the SBOM as source for their analysis of dependencies. They usually run a SBOM generation tool in the background and use the generated SBOM to resolve components names and retrieve their security information from public databases (e.g., NVD).

The use of SBOMs makes the behavior of these vulnerability scanners straightforward. They parse the SBOM collecting dependency identifiers, such as purls, and search for a match in vulnerability databases.

Grype is commonly used to analyze SBOMs and obtain security reports. A Grype's security report contains the following fields for each vulnerability:

- *Vulnerability*, information on the specific matched vulnerability (e.g. ID, severity, CVSS score, fix information, links for more information)

```
{
  "matches": [{
    "vulnerability": { "id": "GHSA-9wx4-h78v-vm56",
    "severity": "Medium",
    "fix": { "version": "2.32.0" }},
    "relatedVulnerabilities": [{
    "id": "CVE-2024-35195",
    "severity": "Medium" }],
    "matchDetails": { "type": "exact-direct-match",
    "package": { "name": "requests", "version": "2.31.0" },
    "found": {
      "versionConstraint": "<2.32.0 (python)",
      "vulnerabilityID": "GHSA-9wx4-h78v-vm56"
     }
    "artifact": {
    "name": "requests",
    "version": "2.31.0",
    "purl": "pkg:pypi/requests@2.31.0" }
  }]
}
```


Figure 2.3 Example of a condensed Grype scan report for a Python SBOM.

- *RelatedVulnerabilities*, information pertaining to vulnerabilities found to be related to the main reported vulnerability, e.g., if the tool matches a vulnerability on GitHub Security Advisory, also the upstream CVE is reported.
- *MatchDetails*, the elements matching the vulnerability, such as the version constraints for which the vulnerability is matched.
- *Artifact*, information about the location of the package within the directory, package type, licensing information, purl, CPEs, etc.

Figure 2.3 shows an example of a security report generated by Grype for a Python SBOM.

2.3 CI/CD Automation: GitHub Actions

This Section briefly describes GitHub Actions and their core elements. Then it discusses vulnerabilities affecting workflows, specifically GitHub Actions workflows, and how attackers can exploit them to compromise code repositories in the software supply chain.

2.3.1 GitHub Actions

GitHub Actions were introduced in 2019 to automate, customize, and execute software development workflows in code repositories [90]. They attracted developers to ease automation routines in their projects independently of their size.

GitHub Actions essentially is an event-driven API. GitHub Actions are defined as workflows in one or more files in YAML format that need to be stored in the `.github/workflows/` directory of the target GitHub repository.

A workflow is composed of one or more jobs. A job allows developers to define the environment and configuration where a sequence of tasks (namely, *steps*) will run. Each workflow can be associated with a list of events that trigger its execution. Examples of events include `pull request`, `push`, and `merge`. Runners are computing elements that host the execution of workflows for repositories. A Runner can be hosted both on GitHub dedicated servers and self-hosted machines.

At the start of each workflow run, GitHub automatically creates a unique `GITHUB_TOKEN` to authenticate the request, granting each runner privileges to interact with the repository on behalf of GHA. Administrators of the repository can set the permissions granted to the token to restrict access to specific resources or jobs. The default permissions can be either permissive or restricted. In the first case, the `GITHUB_TOKEN` has full access to the resources of the repository, while - in the second case - the capabilities are limited to read the content of the repository [99].

GitHub Actions offer developers artifacts to manage different aspects of workflow execution. Contexts are a way to access information about workflow runs, runner environments, jobs, and steps. Each context is an object that contains properties, which can be strings or other objects. Among the list of possible contexts, `github` [91] and `secrets` [98] are the most used ones.

The `github` context contains the event that triggered the workflow run plus some further information. It can be involved in the workflow execution to provide information like the actor who triggered the workflow or the body text of a newly created issue. The `secrets` context contains the names and values of secrets that are available to a workflow run. Secrets can be used to manage confidential information, like API keys and passwords. For example, the `GITHUB_TOKEN` is automatically included in the `secrets` context. Finally, GitHub Actions offer the possibility to make workflows reusable [97]. This mechanism enables anyone with access to the repository and the reusable workflow to call it from another workflow. Workflow reuse also promotes best practices by helping developers use well-designed workflows that have already been tested and proven effective. Also, reusable workflows enable the definition

of organization-wide libraries of workflows that can be used to speed up the creation of CI/CD pipelines.

2.3.2 Security Issues of GitHub Actions workflows

The introduction of workflows enabled code repositories to become an integral part of the CI/CD pipeline. In particular, GitHub Actions workflows can operate on the code repository by programmatically adding, removing, and modifying its content. Those capabilities, however, can directly affect the confidentiality, integrity, and availability of the software and the associated metadata information.

In detail, an attacker can leverage security weaknesses and misconfigurations in the definition and execution of a workflow. Some elements of the latter can be manipulated by the actor triggering the action, who, maliciously, can craft specific inputs to cause unexpected execution flows in the workflow.

For instance, context elements are susceptible to several security weaknesses and misuse, as stated in the official documentation [91]. On one hand, the `secrets` context can contain sensitive information — e.g., a token or a key — that is encrypted using the default key and can be safely manipulated inside a workflow environment. On the other hand, a poorly configured workflow may grant direct access to a secret, thereby allowing an attacker either to send it to unintended hosts or explicitly print it to the log output [95].

As for secrets, direct access to variables of the `github` context can lead to command injection attacks. In detail, the `github` context allows storing information that directly depends on the user's input (e.g., the body of an event). Those data can be manipulated and executed inside by a Runner using the `run` job. Unfortunately, a poorly-configured workflow can allow attackers to inject a crafted input to trigger its direct execution inside the Runner, thereby compromising the repository or the execution environment.

An example of vulnerable scenario is depicted in Figure 2.4. In this scenario, the `issue.title` element is directly executed in the run step. If an attacker can inject in the title of the issue the string: `New malicious issue title" && bash -i >& /dev/tcp/0.tcp.eu.ngrok.io/14872 0>&1 && echo "`, she can obtain a reverse shell on the remote machine hosting the Runner.

In addition to the security weaknesses of workflow elements, two other features of GitHub Actions directly impact the security of workflows — i.e., *permissions* and *reusable workflows*.

```yaml
on:
  issues:
    types: [opened]

jobs:
  vuln_job:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: vuln_step
        run: |
         echo "ISSUE TITLE: ${{github.event.issue.title}}"
```

Figure 2.4 Example of Github Workflow vulnerable to command injection attacks.

In the first case, permissions can affect the capability of a successful attack on a repository [94]. To this aim, developers should enforce the least privilege principle by assigning the least set of permissions tokens and workflows, and differentiate - when possible - the set of permissions granted to each job.

Finally, reusable workflows allow repository owners to import both workflows belonging to their organization and workflows developed by publicly available third parties [96]. In this latter case, the caller workflow does not control the imported one (the callee). The introduction of unmonitored workflows increases the attack surface since the callee may include potential weaknesses and misconfigurations, especially if the imported version is not updated. Also, attackers can manipulate reusable workflows by exploiting tag-reuse attacks to trick the caller workflow into importing a different version of the callee, which has been maliciously re-tagged. To prevent such issues, the best practice suggests selecting the workflow version using a commit hash (e.g., `workflow_name@cdd...08c`) [96].

2.4 Libraries Distribution

2.4.1 Packaging Ecosystems

Software applications are routinely developed by reusing existing (often open-source) components. The application with its dependencies on components, which again may depend on other components form *software supply chains*. Components are usually distributed as *packages* with a *package manager* and corresponding *package repository*, such as *npm* and *PyPI*.

Groups of packages, their developers, and their users often form an interdependent *ecosystem*. Software ecosystems are frequently underpinned by a common technological platform or market [202]. A software ecosystem can be defined as a *packaging ecosystem* when it revolves around package managers for a specific programming language [69].

A package within an ecosystem contains the files necessary for other software to use its functionality. These may include source code and binary files, as well as tests and documentation. Among those files, there is usually a *specification file* containing information on how the package manager must build the package and other metadata. There are two kinds of specification files: static and dynamic. A static specification file contains hard-coded metadata that the package manager should not modify during the build process. A dynamic file may contain arbitrary code or elements that are evaluated at runtime by the package manager during the build process.

In the npm, PyPI, and RubyGems package ecosystems, packages include primarily source code, but they may also include *native code extensions*. Native code extensions provide an API to compiled code, usually written in C, typically to optimize for resource-intensive activities.

A packaging ecosystem typically comes with tooling to create and publish components as packages, such as `pip` for PyPI and `mvn` for Maven. Most ecosystems have broadly used standard tools, but there may be competing tools such as `npm` and `yarn` in the npm ecosystem. In addition, most tools are highly configurable – for example `pip` uses a frontend–backend approach, where `pip` delegates the actual packaging to a configurable build backend; in addition `pip` works on two interfaces using two different specification files, an interface (legacy) based on the `setup.py` file and another interface based on the `pyproject.toml` file. Similarly, `mvn` is plugin based, injesting a `pom.xml` specification file to control the actions of various plugins.

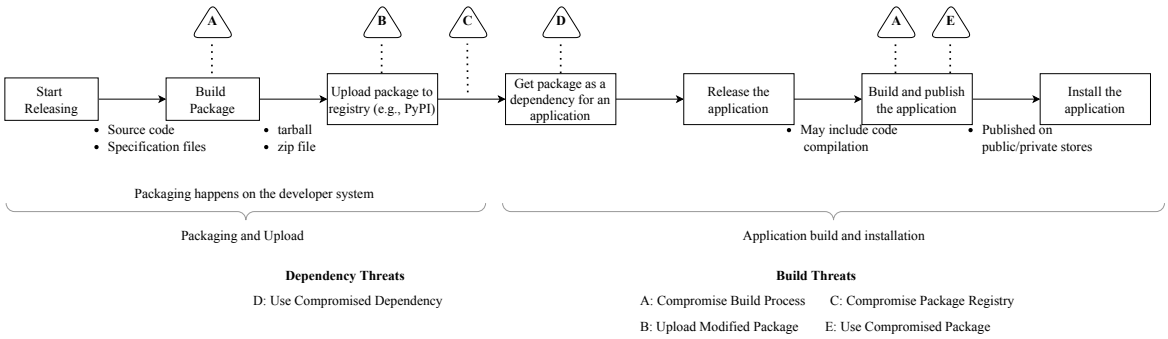


Figure 2.5 Workflow from source code to distribution.

2.4.2 Distribution Model

Figure 2.5 shows a typical distribution model for packaging ecosystems. It consists of two processes: (1) packaging and registry upload for components as packages and (2) build and installation for applications.

Components are usually developed and tested locally on a developer's machine or with some public build and continuous integration infrastructure. The source code of open-source components is publicly available and anybody can suggest modifications. The package manager's toolchain can then be used to release the package as a distributable artifact. At this stage, the package manager takes the source code and the specification files, collects dependencies, collects and possibly compiles code, and generates a package. This process is controlled by the various configuration files in the repository and potentially additional command-line arguments. During the packaging stage, information about the build environment can be captured as metadata (e.g., timestamps and release version). The resulting package file is then *uploaded* in the ecosystem's package repository.

Consumers of a component, such as other components and applications, usually use the packaged version in the package repository, downloading and installing it through the package manager, assuming that the distributed package corresponds to the component's source code in its public repository [154]. The dependencies of an application can be integrated during the build phase and included in the final artifact, or they might be gathered and installed on the end user's system separately. The resulting application can then be distributed to end users internally or publicly through various channels, including direct downloads, application stores, and as packages in package ecosystems (e.g., common for developer applications like static analysis tools or test runners).

Various threats can affect the distribution model (see Figure 2.5). According to Ladisa et al. [131], these threats can be implemented through multiple attacks. The build process can

be compromised (A) due to weak configurations, vulnerabilities, or malicious components. A modified package may be uploaded to the registry (B) by compromising the host or maintainer systems or by hijacking a legitimate account. Similarly, a package registry can be compromised (C). Once distributed, compromised packages affect other packages or applications that use them (D). Both package and application build processes may use compromised packages (E). Reproducible builds are one countermeasure to ensure build integrity, ensuring packages are built from current, *unmodified* sources and dependencies.

2.4.3 Challenges of Making Builds Reproducible

The idea of reproducible builds has been broadly promoted to ensure an independently verifiable path from source to published artifacts where verifiably no additional[1] vulnerabilities or malicious code has been introduced. Identical results for every build of a given source allow multiple parties to come to a consensus and highlight any deviations from the expected build result.

Reproducible builds have two requirements that can be difficult to ensure: (1) The build process must be deterministic. (2) The build environment must be either recorded or pre-defined. Build tools and programming languages were not originally designed for reproducibility and contain many causes of non-determinism that affect build reproducibility. Lamb and Zacchiroli [132] provide an overview of common sources of non-determinism during the build process:

- *Build timestamps* are the main source of unreproducible builds. Many tools embed timestamps inside build artifacts, even though they may have limited practical value. The Reproducible Builds project proposed the `SOURCE_DATE_EPOCH` environment variable as a way to communicate a fixed timestamp to build systems [42].
- *File ordering* for the `readdir(3)` system call is not specified in the POSIX Unix standard and differently ordered file lists may affect build artifacts. To avoid these issues, build systems should impose a deterministic order on any directory iteration encoded in its artifacts, e.g. via an explicit `sort()`.
- *Archive metadata* of `zip` archives and tarballs (i.e., `tar` archives) may contain timestamps and permissions for each file.
- *Randomness*, intentional or accidental, of any compilation or code generation step in the build may influence resulting binaries.

[1] Malicious code already hidden in the original source or tools used in the build may be built reproducibly.

Recent research has studied the perceptions of reproducible builds and the social challenges to their adoption. Fourné et al. [85] analyzed enablers and blockers for adopting reproducible builds in the open-source community. They report that most industry practitioners *considers reproducible builds to be out of reach* – reproducible builds are seen as valuable but not essential. Other studies confirm that many developers are not aware of reproducible builds when developing their build systems [179, 63]. Butler et al. [43] identified the need for reproducible builds from a security point of view and the reasons for their limited adoption, minimal business impact, and limited awareness and perceived challenges.

The reproducible-builds.org project provides educational materials, resources, and tools to support developers and software projects in making their build processes reproducible, including `reprotest` [133] to automate the process of building a package multiple times in diverse environments and `diffoscope` [171] to help find the differences between binary packages and directories.

However, adoption of reproducible builds is uneven, with dedicated efforts in Debian achieving substantial access [134] but minimal attention to reproducible builds in other areas.

Goswami et al. [114] examined the reproducibility of npm packages by comparing the build output of upstream repository code against artifacts stored on the npm registry. Vu et al. [205] did not directly focus on reproducible builds, but they argued for reproducible builds as a security solution to phantom artifacts, highlighting how difficult it is to achieve reproducible builds. In parallel with our study, Kenshani et al. [126] investigated the feasibility of automatically generating `.buildspec` files from metadata available in Maven packages. The build obtained by the automatically generated `.buildspec` file is then compared to the build hosted in Reproducible Central, which keeps trace of reproducibility for part of the packages in the Maven ecosystem. As additional result of their study, they conducted an explorative analysis trying to find common causes of reproducibility issues. They conclude that Maven packages reproducibility can be achieved by trivial adjustments to the POM.xml file. We confirm this speculation in Section 5.2.2.

Randrianaina et al. [166] studied the impact of configuration options on reproducible builds in highly-configurable system, i.e., Linux, Toybox, and Busybox. By fixing the build environment they were able to understand how such options affect the build reproducibility, a similar approach is used in this study to focus on reproducibility issues related to the package manager.

Automatic approaches to reproducible builds were proposed by Ren et al. with three tools: RepLoc [168], RepTrace [169], and RepFix [170]. However, the complexity of reproducible builds challenges increase because of different practises and policies adopted

by communities [40], and evolution of dependency networks in different ecosystems [65, 68, 66, 67, 69].

Part I

Transparency

Chapter 3

A Framework for Comprehensive Software Supply Chain Security Analysis

The DevOps paradigm has tightly integrated development, delivery, and operations, into the development process, facilitating and speeding up the continuous release of software components [77]. Such a paradigm drove the tight integration of heterogeneous components such as software artifacts (e.g., third-party libraries and binaries), assets (e.g., software repositories and package managers), and practitioners that contribute to a software product or that have the opportunity to modify its content (e.g., developers and maintainers). Those elements compose the software supply chain [18].

In the last years, software supply chains have become harder to manage due to the increasing number of elements composing a software. Practicioners should cope with the evolution of the underlying software including technological changes (e.g., changes in architectures, operating systems, or library upgrades) [213]. For instance, some parts of the software supply chain become unnecessary during software evolution and can be removed, or some others (e.g., testing environments) become obsolete and should be upgraded/replaced.

In addition, from a security standpoint, the software supply chain offers an appealing entry point for attackers aiming to target the final software and its consumers, as witnessed by recent security reports [173, 20]. In particular, a successful attack in the software supply chain might go unnoticed for a long period, impacting a large number of companies that use the affected supplier's software. CodeCov attack [137] exploited a configuration vulnerability in the Docker files of the CodeCov code coverage tool that allowed external attackers to access the source code stored in the repositories of 23,000 customers. The difficulty of maintaining and evaluating a software supply chain security posture requires the attention of both the industrial and research community. For this reason, different approaches emerged

in recent years. These approaches can be mainly divided into two groups. The first set of methodologies and tools focuses on detecting vulnerabilities in the software code directly in the DevOps pipelines. Notable examples include snyk [183] and slscan [180]. Other solutions, instead, focus on the integrity of software and its dependencies, such as Google SLSA [188], MITRE D3FEND [125], and ReproducibleBuilds [172]. However, we argue that the proposed solutions allow the mitigation of security vulnerabilities but fail to identify their root causes and, thus, prevent future attack campaigns. Supporting that, ENISA reports that 66% of attacks targeting the Software Supply Chain come from unknown sources [81]. To fill such a gap, this paper presents SUNSET (Software Supply Chain Risk Identification), a methodology that supports the maintenance and the risk evaluation of software supply chains. First, the methodology allows for the automatic reconnaissance of all the elements of a software supply chain and their dependencies. Then, it supports the identification of the security risks of supply chain components by generating a risk profile that details where threats originate, which path they follow to reach the final software and their severity.

3.1 SUNSET

This section introduces the basics of SUNSET, a methodology to automatically model the software supply chain and identify the risk assets pose to the final software.

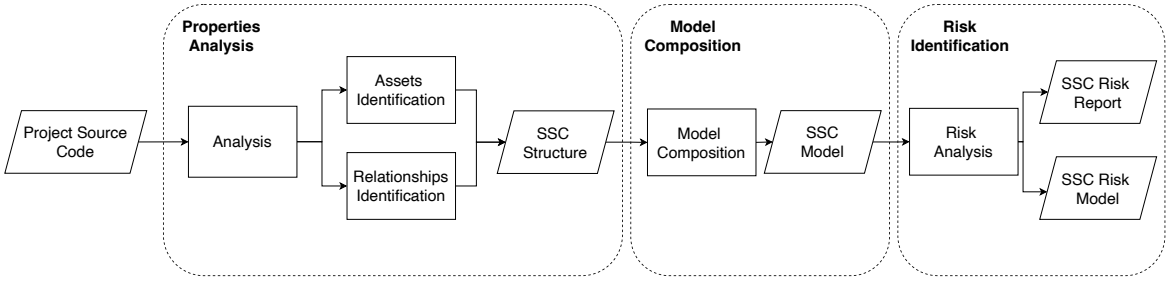


Figure 3.1 SUNSET architecture.

SUNSET takes the final software source code and automatically extracts a model of the software supply chain in terms of assets, and associated dependencies. Then, it extracts the cybersecurity risk of each asset and computes how it can impact the security of the final software product.

The workflow of the methodology, depicted in Figure 3.1, consists of three phases: (1) assets *identification* and functional properties *extraction*; (2) assets and relationships *modeling*; (3) assets *risk identification* and risk propagation computation.

3.1.1 Property Analysis

Asset Identification

SUNSET identifies four categories of assets, namely *software artifacts*, code *holders*, *distribution networks*, and *actors*.

Software Artifact represents any software included or developed in the software supply chain. SUNSET further discerns between *(i)* compiled software (e.g., binaries and compiled libraries) and *(ii)* source code artifacts.

Holder This type of asset is responsible for storing and maintaining software artifacts. A Holder can be further categorized in:

Local Storage A storage solution not connected to any management system — e.g., a local folder containing software artifacts.

Version Control System (VCS) A system that allows managing software artifacts using a management system that supports code control features like versioning and tracking. Depending on the location, a VCS is either *remote* or *local*.

Distribution Network A system providing services to categorize, search, and distribute software artifacts, also known as package manager. Common distribution networks are Maven and PyPI, which support the distribution of Java and Python libraries, respectively. Both open– and closed–source projects use distribution networks to import software dependencies [83].

Actor Humans involved in the software supply chain. An actor can be classified based on its privileges on the asset it is connected to.

Maintainer Full access to the asset who is connected to. Moreover, it can set privileges for other actors connected to the same asset.

Contributor Restricted access to the asset who is connected to.

As depicted in Figure 3.2, assets are identified in the first stage of the methodology. In particular, the methodology analyses the project files and their content to detect the assets composing the software supply chain.

The process starts by identifying software artifacts. The entry point of the project — e.g., the main function — is identified and the software artifacts are recursively retrieved and parsed. When a software artifact cannot be retrieved locally in the filesystem, SUNSET tries to

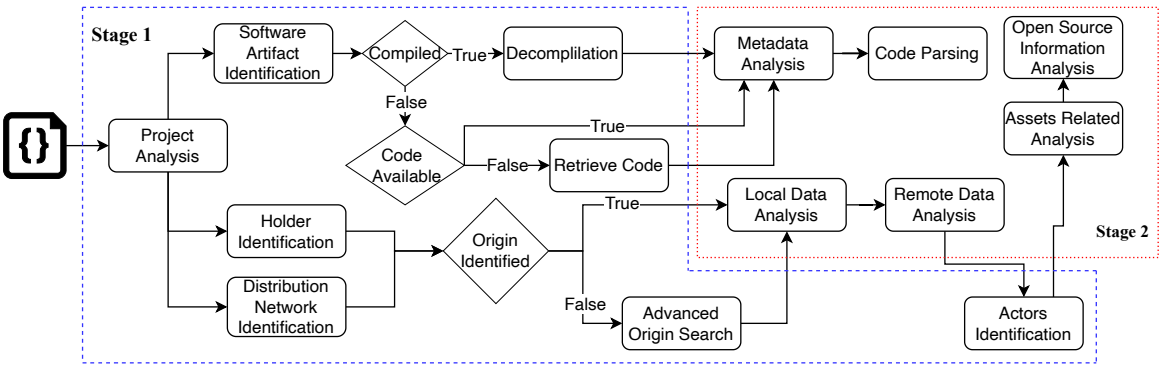


Figure 3.2 Property Extraction Workflow.

find a corresponding artifact online (e.g., by searching for publicly available implementation of the software artifact). If the methodology identifies multiple artifacts, it selects the most exhaustive implementation (considering lines of code and last update time, if available). Nevertheless, when this event happens, SUNSET collects the differences between artifact versions as evidence of possible attack vectors (e.g., typosquatting [204], i.e., trick users into downloading a malicious package by squatting the name of a popular package).

The methodology identifies holders through the analysis of project files. SUNSET detects the specific files VCSs use to deal with code versioning (e.g., indexing and configuration files). If there is no local fingerprint, the methodology uses software artifacts' source code to explore available public repositories to couple software artifacts to a VCS. When the methodology fails to associate software artifacts with a VCS, it creates a local storage holder to contain the software artifacts.

Distribution networks help to identify the provenance of software artifacts since programming languages are usually supported by a software ecosystem and the related package manager — e.g., PyPI supports Python, and Cargo supports Rust.

Actors are identified through the information contained by the holders and the distribution networks. The methodology merges multiple actor assets into a single one when there is evidence that they belong to the same person.

Properties Extraction

The methodology extracts a different set of properties for each category of assets. Each asset has two categories of properties:

Structural properties Characterize the asset providing information about the structure and quality of usage and involvement in the software supply chain.

Security properties Concern the security posture of the asset. They provide information regarding possible flaws and entry points.

SUNSET extracts properties from assets during their identification. Table 3.1 reports an example of categories that can be extracted. Depending on the asset category, the methodology extracts the proper groups of properties. The single properties of each group receive a quantitative evaluation, depending on their characteristics and the availability of plugins to support the extraction, e.g., a static code analyzer for security properties of software artifacts. Stage 2 of the workflow in Figure 3.2) depicts the corresponding extraction process.

	Structural	**Security**
Software Artifact	Conditional Statements	Buffers Validation
	Functions	Input Sanitization
	Required User Interactions	Insecure Patterns
	Read and Write Operations	
Holder	Commits	Security Policies
	Pull Requests	Community Standards
	Issues	Known Security Issues
	Workflows	
Distribution Network	Mirrors	Known Security Issues
	Packages Required	
Actor	Homepage	OSINT results
	Overall Contributions	Known Malicious Actions
	Public Repositories	
	Forks	

Table 3.1 Properties Categories and Groups divided per Asset Category.

For software artifact assets, properties extraction consists of analyzing the metadata, the source code (if available), and the result of SAST tools. In the case of a compiled software artifact, instead, SUNSET analyses the decompiled code relying on state-of-the-art decompilation tools [164, 122, 146].

Assets belonging to the Holder and Distribution Network categories are analyzed by considering *(i)* the metadata located in the project (e.g., indexing files, mirror files), *(ii)* the remote information (e.g., remote branches, pull requests, versioning information), and *(iii)* the existence of known security issues on publicly available vulnerability databases.

For the actor assets, the methodology takes advantage of the information provided by the assets from which the actor has been obtained (e.g., contribution to the repository). The information gathered through the asset where the actor contributes is integrated with the analysis of open-source information [59]. As said multiple actors can be matched to a single entity. Analyzing the links bounding them to a single entity allows a better understanding of their involvement in the software supply chain. Thanks to this understanding, it would be possible to capture information on potential threat actors. The identification of these links involves the use of state-of-the-art tools for the analysis of personas [182, 138].

3.1.2 Model Composition

The model composition phase (Fig. 3.1) allows building a structured representation of the software supply chain. The generation of the model organizes the assets and their inter-dependencies using a direct graph structure, where nodes represent assets and edges detail their relationships.

The final software is a sink node for the model — i.e., the end of each of its incoming edges. The edges entering the node connect the final software to the subgraphs containing the assets identified during the Asset Identification phase. Figure 3.3 depicts possible relationships between two assets.

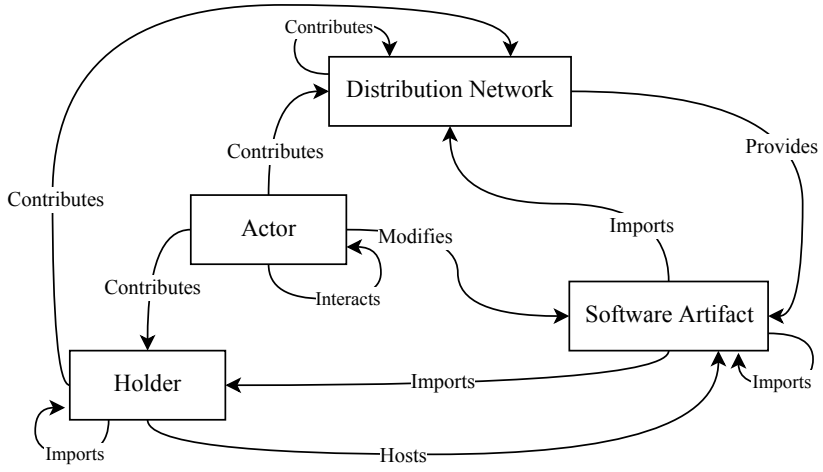


Figure 3.3 Possible relationships among SSC asset categories.

For example, Figure 3.4 depicts the model generated by the analysis and modeling phase of the software supply chain of *MyWebApp* (Figure 2.2). The software supply chain model also supports graphical plotting and manipulation using state-of-the-art tools, e.g., [116]. The high-level overview offered by the visual representation supports maintenance tasks and

preliminary security assessments. The model provides additional information — e.g., nodes centrality [32] can be used to understand which nodes have more influence w.r.t. the final software.

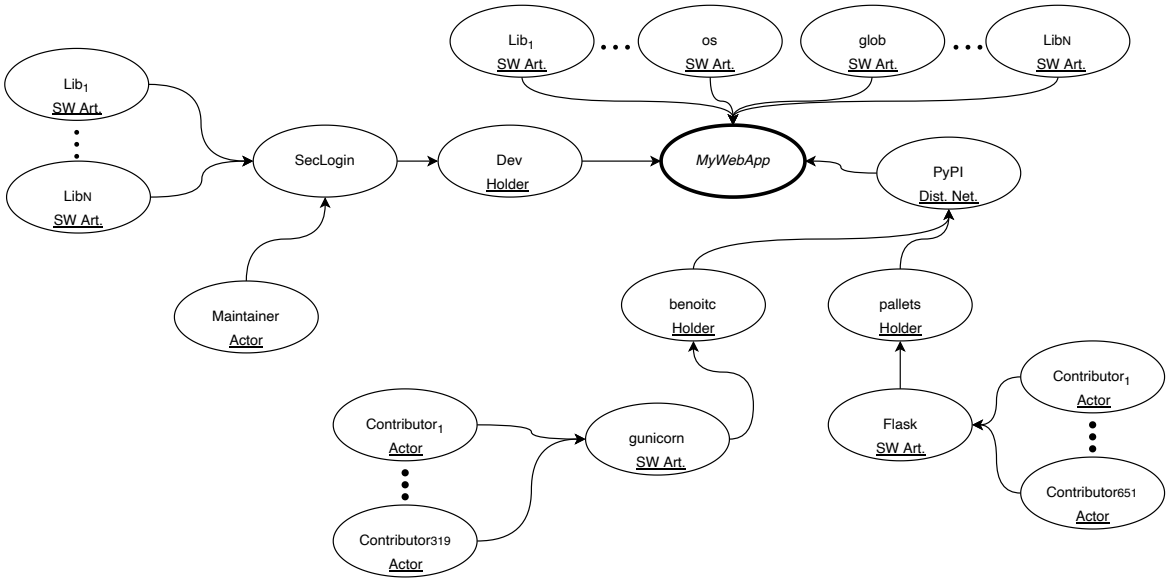


Figure 3.4 Example of model representation.

3.1.3 Risk Identification

The methodology uses the model obtained in the previous phase to carry on the risk identification. This phase concerns searching and analyzing risk sources in the software by considering the risk generated by the single assets and their propagation through the software supply chain. Figure 3.5 depicts the workflow for the risk identification phase.

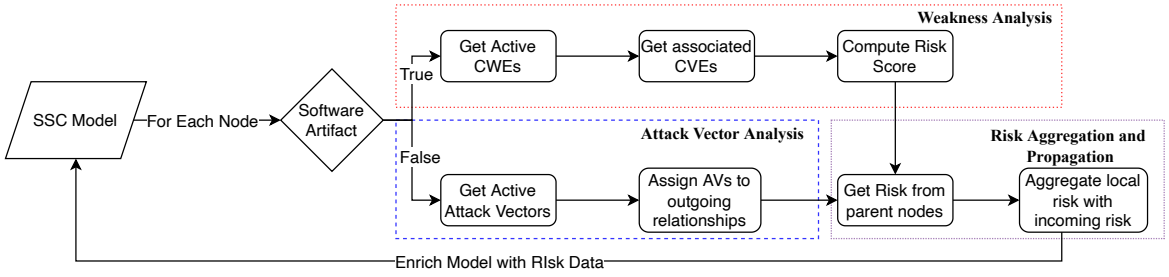


Figure 3.5 Risk Analysis Module Workflow.

SUNSET explores the model graph with a breadth-first search algorithm [56] starting from the outer bounds. two types of analysis are used, depending on the category of the

asset: (1) weakness analysis, for software artifacts; (2) attack vector analysis, for holders, distribution networks, and actors.

SUNSET builds a knowledge base to evaluate weaknesses and attack vectors. The knowledge base maps assets' structural and security properties with a CWE or an attack vector (AV). The reference set of CWEs and AVs are: CWE View 699 for software pre–release phase [195] and the set of attack vectors listed by ENISA [81].

In detail, the knowledge base contains a list of first-order logic statements evaluating structural properties (P), security properties (S), and the presence of specific attack vectors (AV). Each property can be compared with a threshold value (T), joint or disjoint with other properties or predicates, and evaluated in its presence or absence ($\neg$ operator).

For example, the first expression in Listing 3.1 states that attack vector AV_x is enabled when both properties P_1 and P_2 are greater than their respective critical values C_1 and C_2 and when the attack vector AV_y is active. The second statement in Listing 3.1 details how CWE_z is enabled either when properties P_1 and P_2 are greater than critical values C_1 and C_2 or when the attack vector AV_x is active. More detailed examples are presented in Listings 3.2 and 3.4.

$$\mathtt{AV}_x \implies (P_1 > C_1) \wedge (P_2 > C_2) \wedge \mathtt{AV}_y$$
$$\mathtt{CWE}_z \implies (P_1 > C_1) \wedge (P_2 > C_2) \vee \mathtt{AV}_x$$

Listing 3.1 Mapping rules of *Sunset* knowledge base.

Weakness Analysis

The weakness analysis uses the CWE database, the CVSS scoring system [82], and the properties extracted for the software artifacts.

The asset properties are used to evaluate expressions contained in the knowledge base, and then to identify CWE affecting the software artifact. Finally, SUNSET provides the list of active CWE of the asset by verifying attack vectors inherited from relationships with other assets.

For example, the presence of conditional statements (P_{cond}) but the lack of variable sanitization (S_{san}) triggers the rule on CWE-478 [192] (lack of default condition in switch statements) detailed in Listing 3.2.

$$\mathtt{CWE\text{-}478} \implies P_{\mathsf{cond}} \wedge \neg S_{\mathsf{san}}$$

Listing 3.2 KB rule for CWE-478.

After the active CWEs evaluation, SUNSET defines the overall risk associated with the asset. For the computation, the methodology retrieves all the Common Vulnerabilities Exposures (CVE) [189] grouped w.r.t. a given CWE — i.e., the vulnerabilities linked to each CWE that are available on public databases. For each CVE, SUNSET extracts the corresponding CVSS score [82] and uses it to compute an aggregated CVSS risk score for the asset.

According to Listing 3.3, we define G_x as the group of active CWEs on an asset X **(1)**; for each CWE in G_x, the methodology gathers the corresponding list of CVSS vectors (S_i), one for each CVE. Each S_i contains L different metrics M **(2)**. The risk score of the CWE j (R_{CWE_j}) is a new vector where each metric K_i is the mean value of all the same metrics of each CVSS score contained in CWE j **(3)**. The overall score R_x of the asset is the max value among the set of R_{CWE} **(4)**.

1 **(1)** $G_x = \{\texttt{CWE}_1, ..., \texttt{CWE}_T\}$
2 **(2)** $\texttt{CWE}_j = \{\texttt{S}_1, ..., \texttt{S}_N\}$ `where` $S_i = \{M_{i1}, ..., M_{iL}\}$
3 **(3)** $R_{\texttt{CWE}_j} = \{K_1, ..., K_L\}$ `where` $K_i = \frac{M_{1i}+...+M_{Ni}}{N}$
4 **(4)** $R_x = \max\{R_{\texttt{CWE}_1}, ..., R_{\texttt{CWE}_T}\}$

Listing 3.3 Equations for computing the risk score of an asset.

Attack Vector Analysis

Attack vector analysis follows the same concept as weakness analysis. In detail, SUNSET identifies a set of active AVs on an asset X if and only if the functional and security properties allow their presence. The methodology exploits the rules defined in the knowledge base for the evaluation.

For example, the asset X is susceptible to manipulation attacks (AV_{man}) if the asset X inherits the attack vector social engineering (AV_{se}) from a linked asset and the security property weak password (S_{wp}) is below the threshold T_l, according to the rule in Listing 3.4.

1 $$\texttt{AV}_{\texttt{man}} \implies \texttt{AV}_{\texttt{se}} \wedge (S_{\texttt{wp}} < T_l)$$

Listing 3.4 KB rule for the manipulation attack vector.

Risk Aggregation and Propagation.

Attack vectors and risk evaluations propagate in the software supply chain model according to the relationships among the different assets. Weaknesses and attack vectors flow from the

boundary of the graph toward the final software. In this sense, a relationship between two assets has two goals: (1) carrying attack vectors for the weakness analysis; and (2) transporting the risk value obtained on connected assets.

Hence, the total risk on an asset consists of aggregating the risk value generated on the asset with the risk values inherited from inbound relationships.

In detail, SUNSET starts the exploration of the SSC model from the border of the final software SSC using a breadth-first search algorithm.

For each inbound connection, the methodology adds the risk score and the attack vectors inherited from the parent node. Then it iterates the process for each node until it finds the final software. On each step, SUNSET updates the weakness analysis and the AV analysis to match the new conditions.

Looking back at the example in Figure 2.2, the sources of risk of the *MyWebApp* are the relationships incoming from the `Dev/seclogin` holder, the `Std Libs` software artifacts, and the *PyPI* distribution network. Suppose the methodology reports a social engineering attack vector on the maintainer of *Dev/seclogin*. In that case, such AV will be propagated on the holder and, consequently, in the software artifact of the module and the final software.

3.2 Related Work

Both the industrial and scientific communities proposed several solutions to increase the security of software [75]. Most of the activities focused mainly on vulnerability analysis and software integrity.

Tools like the OWASP Dependency-Check [152], snyk [183], slscan [180], and shhgit [62] provide the developer with detailed vulnerability reports of vulnerability patterns and insecure dependencies. Also, the scientific community provided several solutions to detect and mitigate software vulnerabilities, such as [89, 135].

Another field of activities was devoted to ensuring the integrity assurance of open-source software. Such works aim to prevent unauthorized modifications/tampering of the software during the development pipeline.

Two of the most recent industry proposal are SLSA [188] and Reproducible builds [172]. Supply chain Levels for Software Artifacts (SLSA) is an end-to-end framework to guarantee the integrity of dependencies all along the development process. Through SLSA certifications, developers obtain information on the integrity assurance a given artifact can offer. However, the certification process requires an extended interaction with the developers and hardly copes with the level of automation needed for the DevOps paradigm.

Reproducible builds [172], instead, is a collection of software development processes that aims to standardize the build and compilation process in terms of configurations and requisites. This approach enables maintainers to detect if an attacker has compromised the building process by comparing the assets generated during the compilation process.

Hence, Reproducible Builds focuses on the integrity compromise happening in the build step. Weaknesses inserted in the software by mistake or intentionally are then considered a trusted part of the build. On the same approach, the authors of LastPyMile [205] proposed a methodology to detect the differences between build artifacts of software packages and the respective source code repository.

The aforementioned approaches might detect vulnerable dependencies and insecure code and contribute to software packages' integrity. Still, they hardly cope with the root cause of the problem, the attack vector, and the affected asset of the SSC, failing to mitigate the risk of new attack campaigns. Such lack of control is particularly disruptive in complex software supply chains containing thousands of assets, thereby limiting the benefits of adopting VA and integrity solutions. SUNSET is one of the first attempts to mitigate such pain for developers and SSC maintainers.

3.3 Conclusion and Future Work

SUNSET is a new methodology to model software supply chains and evaluate their security risk and the detail of the single asset. SUNSET is not intended to substitute traditional VA and PT procedures or risk management activities. The methodology, instead, aims to alleviate the burden of maintaining a secure and updated SSC by providing (1) risk evaluation of each asset and how it will influence the security posture of the final software; and (2) identification of sources of risk to prioritize mitigation activities. Also, the evaluation of SUNSET can be performed offline without impacting the performance of the development process.

Deeper research is necessary to make this methodology usable in real-world scenarios. Among them, the catalog of assets needs to be incremented with more precise and complex elements. While there is a proof of concept implementation of SUNSET, it does not reflect exactly the proposed methodology, requiring improvements for the modeling and risk evaluation phases.

Chapter 4

Using SBOM for Vulnerability Assessment

The security of the Software Supply Chain (SSC) is an increasing concern for users and developers as reported by both ENISA [80] and the UE Executive Order on Improving the Nation's Cybersecurity [38]. Indeed, incidents such as the infection of SolarWind's Orion platform demonstrated how far-reached and impactful the distribution of compromised software is [155]. The security of the SSC depends on multiple factors, including, but not limited to, the use of open-source software as dependencies included in the developed application [131]. In a 2023 study of 1,703 commercial codebases across 17 industry sectors, Synopsys found that 96% of them leverage open-source code, and 76% of the total application code was open-source [187]. Therefore, targeting software components, such as libraries, allows attackers to affect a wide range of software using a single entry point [131, 207, 117, 141].

To improve the security posture of the SSC, the Executive Order [38] pushed the SBOM as a tool to increase the transparency and verifiability of the distributed software. According to the Cybersecurity and Infrastructure Security Agency (CISA), an SBOM is "*a formal record containing the details and supply chain relationships of various components used in building software*" [61], hence providing developers and enterprises with transparency in the software composition. An SBOM provides benefits for both software suppliers and consumers, as it helps identify and avoid known vulnerabilities, quantify and manage licenses, identify security and license compliance, and manage mitigation of vulnerabilities. SBOMs are generated by automated tools and created according to different formats, with the most common being Software Identification (SWID) tagging, Software Package Data Exchange (SPDX), and CycloneDx [61]. To fully leverage the information provided by SBOMs, a

plethora of tools have been developed to receive SBOM as input and provide security information [13, 19, 2, 8]. To gather information on the security of components listed in the SBOM, these tools rely on open-source vulnerability databases, e.g., the NVD, to map components to vulnerabilities. Software components can be associated with different information security depending on the database [74]. Sometimes, SBOMs can also be complemented with a Vulnerability Exploitability eXchange (VEX), which provides additional information on possible specific vulnerabilities affecting software components of the SBOM. Overall, the use of SBOMs both increases the transparency of the distributed software and speeds up software adoption and testing. Indeed, retrieving known vulnerabilities of the listed software components via polling a public database is faster than running static or dynamic analysis application security testing.

Are SBOMs Improving Security? Although the premises are good, the SBOM is not what it is expected to be for security. Most SBOM generation tools use specification files (e.g., setup.py, gemspec, package.json) to gather the dependency index. Other approaches are based on source or binary code parsing. Due to the lack of a standardized SBOM format and the limitations in the accuracy of existing SBOM generation tools [37], it is possible to end up with different generated SBOMs for the same software. Indeed, the SBOM generation process depends on the tool's capability of correctly identifying components' names, versions, and dependencies. Wrongly identifying one or more of these elements impairs the representation capabilities of the resulting SBOM [37]. Moreover, the claims made by SBOM generation tools on their support capabilities for different metadata file formats and their performance on real code bases are not consistent. Indeed, different and well-known SBOM generation tools are prone to parsing errors or inability to correctly gather all dependencies, resulting in an SBOM that represents a subset of the entire code base [72, 31, 163, 52]. This represents a problem not only because the SBOM does not accurately represent the code, but also because missed key dependencies may lead to missed key security issues. Such dependency resolution problem is still an open issue for SBOM generation, and it is not clear how this impacts the security analysis capabilities provided via SBOMs. Approaches such as code-centric call graph analysis and behavioral analysis may solve this problem, however, they are highly resource intensive [120, 159–161].

The limitations of the currently existing SBOM generation tools and the need for secure solutions to improve the security posture of the SSC led us to the following research questions:

- **RQ1**: *How much does the SBOM generation process impact the detection of vulnerabilities in the dependency network of an SSC?*
  - **RQ1.1**: *How does a specific vulnerability scanner perform when fed with an SBOM generated by a specific state-of-the-art SBOM generation tools?*
  - **RQ1.2**: *How much does an SBOM generation approach affect the performance of a vulnerability scanner?*
- **RQ2**: *How can we improve the SBOM generation approach to achieve better performance on the security assessment of the dependency network in an SSC?*

Contributions. This Section presents how the Thesis evaluates to what extent the representational capabilities of SBOM generation tools impact the identification of known vulnerabilities in the SSC. Despite existing work evaluating SBOM generation tool according to their ability to correctly identify component's name, version, and dependencies for Java [31], JavaScript [163], and Python [52], no existing work evaluated their impact on the detection of security issues. As outlined in our research questions, these issues are expected to greatly impact the identification of known vulnerabilities in the SSC. To this aim, the study considers five of the most relevant SBOM generation tools — i.e., cdxgen, GH-sbom, ORT, Syft, and Trivy— for the evaluation. Since many SBOM generation tools operate on the set of the most used programming languages (e.g., Python, JavaScript), the tools are evaluated in the context of the Python programming language, the most popular programming language in 2024 according to IEEE [48]. To solve the issues of SBOM generation tools and improve the security posture of the SSC I propose PIP-SBOM, a novel `pip`-based solution. My solution overcomes the most relevant issues of SBOM generation tool, i.e., it correctly identifies component names and versions and can correctly report all software dependencies. I compare the performance of our solution with that of existing state-of-the-art tools and show that the SBOM generated with our tool drastically (64% more precise than the best performing SBOM generation tool) increases the capabilities of identifying known vulnerabilities in the SSC.

I summarize our contributions as follows.

- Evaluating the capabilities of SBOM generation tools in helping increasing the SSC security posture. By providing generated SBOMs as input to a vulnerability scanner tool, I evaluate each SBOM generation tool in terms of the number of identified known vulnerabilities.

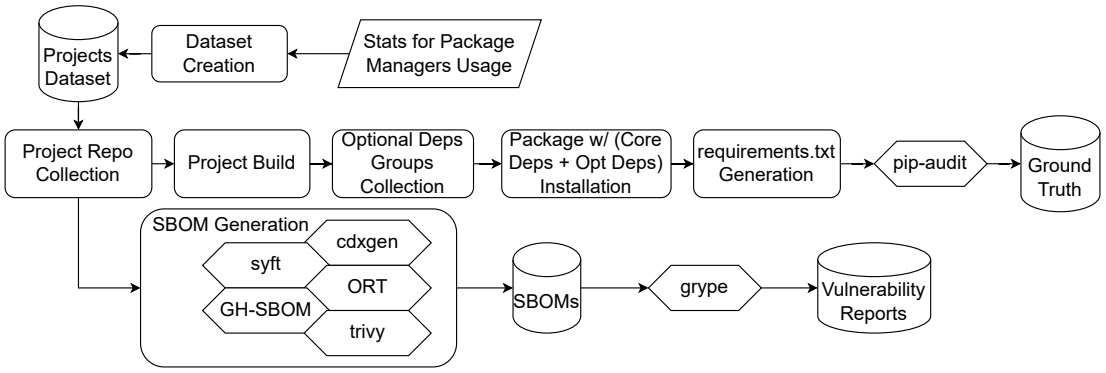


Figure 4.1 Experimental setup design. This approach provides us the necessary data to evaluate our research questions.

- Proposing PIP-SBOM, an extension for PIP to generate an SBOM directly from the package manager, improving both usability and accuracy of the SBOM in the Python ecosystem.

4.1 Experimental Setup

In this section, I describe the setup for our evaluation methodology. At first, Section 4.1.1 provides the process I used to gather Python projects. Section 4.1.2 describes our selection of SBOM generation tools based on their usage for security evaluation and their implemented generation method. Section 4.3 reports the workflow applied to the collected projects to generate SBOMs and obtain their security reports. Figure 4.1 depicts the overall experimental setup.

4.1.1 Projects Collection

Recalling from Section 2.2, Python allows the usage of multiple package managers, each implementing its way of dealing with dependencies. Since SBOM generation tools use specific parts of a Python project to generate the SBOM, I analyze tools' behavior in the context of different package managers. To collect a representative sample of projects, I extract the distribution of the package managers used in 1,351 packages randomly selected from the whole population list on ecosystem.ms. I discard packages that do not clearly state the package manager in their source code repository, obtaining the distribution of package managers shown in Table 4.1.

Table 4.1 Package Managers and Their Usage

Package Manager	Packages	Percentage (%)
poetry	38	6.44
pdm	9	1.53
hatch	85	14.41
pipenv	7	1.19
conda	0	0.00
setuptools	451	76.44

I collect a different sample of 1000 packages with the following process: (1) collect a random package from the entire package population hosted on PyPI; (2) check the package manager used by the package; (3) if I already reached the quota of packages using that specific package manager I discard the package, otherwise I add the package to the sample. This process allows me to have a sample with the same proportion of package managers identified in the previous steps, and generalize our results to the entire package ecosystem with a 3.04% margin of error at 95% confidence level by standard sample size calculations.

4.1.2 SBOM generation tools selection

For the selection of SBOM generation tool I applied the following process: (1) I manually scraped the list of tools on the CycloneDX tool center.[1] I obtained a list of 169 open-source tools. (2) I analyzed each tool in the list selecting those that generate SBOMs, operate on Python, and have a command line interface. I reduce the list to 24 elements. (3) I manually tested the 24 tools to prove their utility in this work. I excluded those that do not correctly execute, require external technologies (e.g., build-root), or were not maintained in the last year. Eventually, I obtained a list of 5 tools: cdxgen, GH-sbom, ORT, Syft, and Trivy.

Table 4.2 lists the selected SBOM generation tools, along with the implemented generation methodology and some example vulnerability scanners making use of them. The evaluation of the SBOMs generated by these tools for a security assessment of the dependency network gives us the answer to RQ1.

[1]https://cyclonedx.org/tool-center/

Table 4.2 List of the selected SBOM generation tools. Most of them are already officially used for dependency network security analysis. The selected tools can be also stratified based on the implemented generation method.

SBOM Gen. Tool	SBOM Gen. Met.	Example Sec. An. Tool
cdxgen	Environment Based	Shiftleft Scan, Macaron [16]
Syft	Metadata Based	Grype, KubeClarity
Trivy	Metadata Based	KubeClarity
ORT	Metadata Based	NA
GH-sbom	Dep. Graph Based	NA

4.1.3 Security Report Ground Truth

A fundamental step to understand the effectiveness of SBOM generation tools in generating an SSC description that leads to the correct identification of vulnerabilities, is knowing the vulnerabilities that affect a specific project. To obtain this ground truth, I follow a multistep approach. For each package in our collected sample I automatically: (1) retrieve the package's project from its code repository; (2) parse metadata files to obtain optional dependency groups; (3) install the package in a virtual environment along with both required and optional dependencies; (4) generate the requirements.txt file using the `pip freeze` command to filter out packages installed in the virtual environment by default; (5) pass the requirements.txt to pip-audit; (6) collect the security report. Thanks to this manual approach, I build the list of vulnerabilities associated with each project. A perfect SBOM generation tool will create a project representation the leads to the correct identification of this precise set of vulnerabilities.

4.1.4 SBOMs and Security Reports Generation

To generate relevant SBOMs, I feed the selected SBOM generation tools with the packages collected in the dataset. I parse SBOMs with `jq` [7] (a JSON parser) to verify they have the expected format. In particular, I verify they do not contain the metadata pointing out the correct analysis of the project by the SBOM generation tool.

I selected Grype [19] as the tool for the generation of security reports. Grype is a vastly used tool for the security analysis of projects [3, 4, 206, 14]. It covers multiple languages — e.g., Python, Go, Rust — and artifacts — e.g., Docker images, filesystems, SBOMs. As I do not need specific security analysis tool for this work, the tool just needs to parse the SBOM and query vulnerability databases for known vulnerabilities. Grype queries multiple

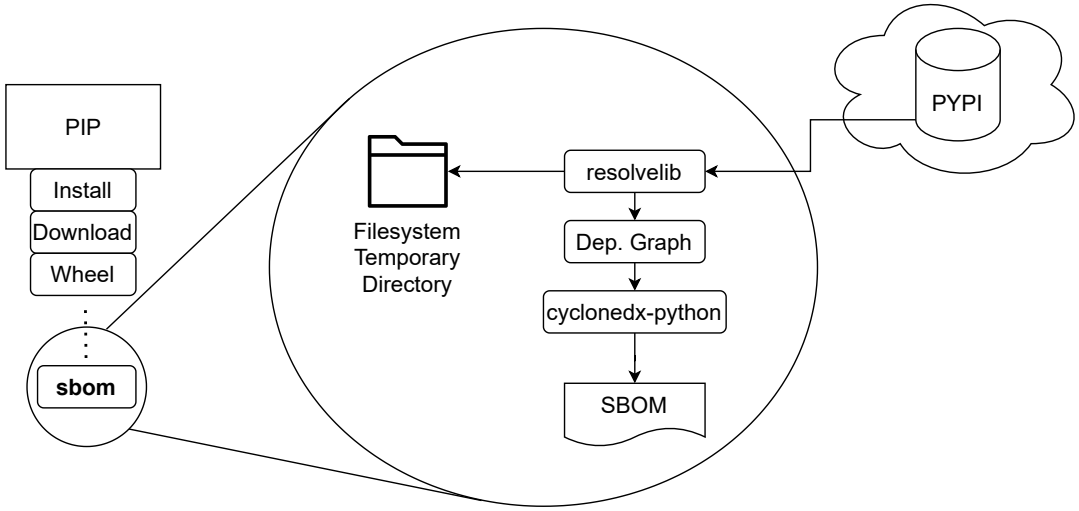


Figure 4.2 Design of PIP-SBOM. I extend the implementation of PIP to include SBOM generation in the build phase.

databases, cross-checking vulnerabilities, and hence represents a perfect choice for our purposes.

A Grype's security report contains matches for found vulnerabilities, allowing us to compare the set of found vulnerabilities with the set of vulnerabilities reported in the ground truth.

4.2 PIP-SBOM: Our Proposed SBOM Generator

This Section describes PIP-SBOM, our PIP-based approach for native generation method. Figure 4.2 depicts its main steps.

PIP-SBOM is designed as part of PIP, the PyPI official package manager. I extend this specific package manager because: (1) it supports multiple front-ends and back-ends, i.e., it can build other package manager projects, such as poetry. (2) Both skilled and novel developers commonly use it.

PIP is internally based on modules representing the possible CLI commands for a user. I added a module — i.e., SBOM— containing the logic needed to generate the SBOM[2]. This allows developers to generate an SBOM for a Python project with the command: `pip sbom <project-path>`.

[2]PIP-SBOM is the extended version of PIP, while SBOM is the specific module extending PIP. Hereafter, I refer to both of them as PIP-SBOM, for simplicity of language.

PIP-SBOM includes an online process and an offline process. The online process interacts with the PyPI registry obtaining the dependency network. The offline process builds the dependency network graph and generates the SBOM document from a Python project.

Dependency Network Solving

PIP uses the `resolve-lib` package dealing with dependencies and version constraints. This package optimizes the solving algorithm with the optimal navigation path of the dependency tree [10]. I build upon the logic already implemented in PIP for this package to mimic the same solving algorithm used during dependencies retrieval.

This process has a similar behavior to the `download` command. The project's dependencies are collected from PyPI and stored inside a directory specified by the user. In my implementation, the dependencies are stored inside a temporary directory and removed at the end of the process. This process automatically solves version constraints similarly to the process happening during project build and installation, providing a reliable representation of the dependency network at installation time. When a constraint cannot be solved because of version incompatibility, it is discarded, as would happen during the project installation.

The generation of the dependency graph is coupled with this process. I decided to store collected dependencies as a graph to allow deeper investigation of dependency relationships when required.

Dependency Network Graph

The dependency network is defined as a direct unweighted graph $G = (V, E)$. Each element $n \in V$ is either a direct or transitive dependency of the input project $r \in V$. An edge $(u, v) \in E$ represents a dependency relationship between the nodes u and v. This kind of dependency is a transitive binary relation. That is, from $u \to v$ and $v \to w$ it follows $u \to w$. In this case, u is a transitive dependency of w.

PIP-SBOM outputs the generated graph as a `dot` file when required through the `-g <file-name>` option. The PIP-SBOM internally uses the dependency graph to generate the SBOM.

SBOM Generation

Once the graph is complete, PIP-SBOM module navigates the graph and creates an entry in the `components` field of the SBOM for each node. An entry contains: the bom-ref, dependency name, version and purl. I include only this information because they are necessary to the

vulnerability scanner. In general, the SBOM can be enriched to comply with the minimum required elements stated by NTIA [149].

Edges of the dependency network are used to fill the *dependencies* field of the SBOM, which contains the relationship between components. When the graph exploration ends, PIP-SBOM produces a CycloneDx-compliant SBOM.

4.3 Evaluation Methodology

In this section, I present our methodology to answer our research questions. The evaluation process is divided into two parts. The first looks at the accuracy the vulnerability scanner reaches with SBOMs generated by selected SBOM generation tools. The second looks at data specific to PIP-SBOM to evaluate how an SBOM generated with a different method affects the vulnerability scanner performance.

RQ1: SBOM Generation Impact on Vulnerability Scanning

This research question aims to understand to what extent the SBOM impacts the security analysis tool output. The SBOM should accurately represent the SSC. Thus, accurately representing the SSC is an enabling property for security analysis of the software dependency network.

RQ1.1: ***How does a specific vulnerability scanner perform when fed with an SBOM generated by a specific state-of-the-art SBOM generation tools?*** To answer this question, I compare the vulnerabilities identified starting from a SBOMs against the ground truth obtained through pip-audit. I use the Jaccard similarity index to compute the degree of overlap and commonality among the two vulnerability sets. For each tool and each SBOM $\mathscr{S}$, the process involves: (1) collecting the security reports generated from $\mathscr{S}$ and extract the matched vulnerabilities (`ToolVulns`); (2) getting the vulnerabilities stored in the ground truth for the project associated with $\mathscr{S}$ (`GrTrVulns`); (3) compute the Jaccard similarity index between `ToolVulns` and `GrTrVulns`, according to Equation (4.1).

$$J(ToolVulns, GrTrVulns) = \frac{|ToolVulns \cap GrTrVulns|}{|ToolVulns \cup GrTrVulns|} \tag{4.1}$$

RQ1.2: ***How much does an SBOM generation approach affect the performance of a vulnerability scanner?*** While Jaccard similarity represents a useful metric to assess the

extent a tool suits security purposes, it cuts off details on the reasons behind the tool's performance. To obtain these missing details and answer *RQ1.2*, I compute the false positives, false negatives, precision, and recall.

Specifically, *precision* assesses the fraction of correctly identified vulnerabilities, according to the ground truth, among all identified vulnerabilities (Equation (4.2)).

$$Precision = \frac{TP}{TP+FP} \tag{4.2}$$

Recall measures the fraction of correctly identified vulnerabilities to all actual vulnerabilities in the ground truth (Equation (4.3)).

$$Recall = \frac{TP}{TP+FN} \tag{4.3}$$

These metrics provide a holistic overview of the performance of each tool's generation method: precision addresses the trustworthiness of identified vulnerabilities, and recall looks at the generation methods' efficacy in allowing security assessment to pinpoint pertinent vulnerabilities.

RQ2: A Better SBOM Generation Approach

As I later show, the main issue with currently existing SBOM generation approaches and the resulting poor performance of vulnerability identification approaches is that SBOM generation tools are not able to correctly identify components and their dependencies. To answer *RQ2* and improve the security posture of the SSC, I extend PIP to investigate to what extent applying the same logic used in the retrieval of dependencies during a project installation for SBOM generation is beneficial. The extension leverages the already present `download` module in PIP. This module helps to download the package's archives without installing them. Modifying this module by adding functional elements for SBOM generation provides us the means for creating a novel and better approach. I provide the details of our implementation in Section 4.2.

To evaluate the improvement in the security analysis by using a SBOM generated with our approach, I apply the same metrics used to answer RQ1.

4.4 Evaluation Results

In this Section, I present the results of our analysis organized by research question. In particular, Section 4.4.1 presents the results of *RQ1*, while Section 4.4.2 presents the results of *RQ2*.

4.4.1 RQ1: SBOM Impact on Vulnerability Scanning

Our findings show that the SBOMs generation approach deeply impacts the performance of vulnerability scans. All the analyzed SBOM generation tools cannot lead to the correct identification of all vulnerabilities in more than 20% of cases, with the only exception of cdxgen achieving correct identification in almost 40% of cases.

RQ1.1: Impact of tools on vulnerability scanner performance

The security reports generated from state-of-the-art SBOM generation tools reveal some shortcomings, particularly regarding thoroughness and accuracy in identifying vulnerabilities. The distribution of Jaccard similarity indexes in Figure 4.3 provides a perspective of the true positive vulnerabilities found with SBOMs generated by different tools. From the results, it is clear that SBOM generation tools badly impact on the vulnerability scans operated by vulnerability scanners.

Among the analyzed SBOM generation tools, the one providing the best vulnerability scan results is cdxgen. This is motivated by the fact that cdxgen (1) installs dependencies in a virtual environment, and (2) supports most of the package managers. These two properties result in being critical for a SBOM generation tool for a proper representation of the SSC. Hence, all tools lacking at least one of these two capabilities result in worse vulnerability scans.

ORT uses an approach similar to dependencies installation in a simulated environment. That is, it uses an external Python module to query the PyPI registry and obtain information on dependencies. However, its support for package managers is limited to Poetry and Pipenv, or projects including `requirements.txt` files.

SBOM generators solely relying on static metadata — i.e., Syft and Trivy— display worse performances for vulnerability detection. This behavior is caused by the fact that they do not install dependencies, while they have good support for different package managers. GH-sbom results are highly influenced by the settings applied by repository owners. Thus, GH-sbom works *only* when the dependency graph feature is enabled on the repository. However, its

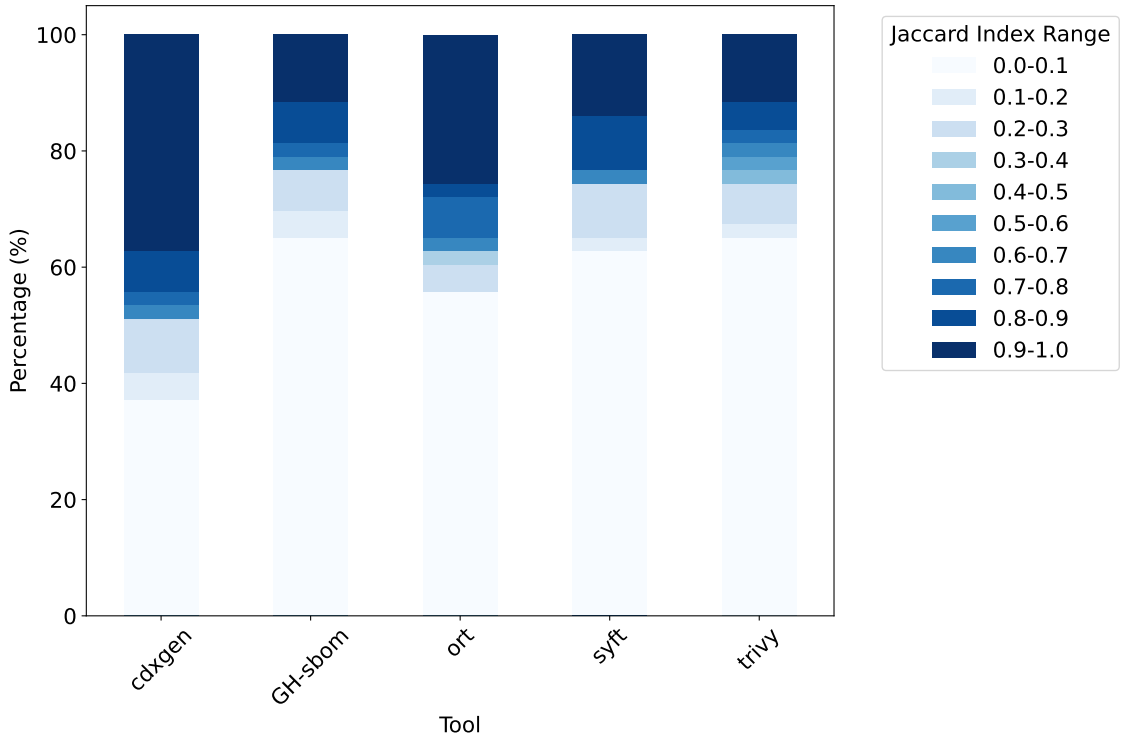


Figure 4.3 Jaccard Similarity Distributions. Each bar represents the percentage of SBOMs that lead to identification with a certain Jaccard index range. I include the vulnerability assessment obtained with SBOMs generated with PIP for comparison purposes.

main issue is that it targets the last commit on the main branch to generate the SBOM. Hence, GH-sbom cannot acquire an SBOM for a specific commit or tag [17], causing vulnerability scanners to analyze SBOMs belonging to code different from the targeted one.

Takeaway: Current state-of-the-art SBOM generation tools are not suitable to generate SBOMs for the proper vulnerability assessment of Python projects.

RQ1.2: Causes of tool impact on vulnerability scanner

While the Jaccard similarity represents how much the SBOM generation approach impacts the vulnerability scan results, precision and recall measurements give us information on the factors influencing the security evaluation. Referring back to Section 4.3, recall is the fraction of correctly identified vulnerabilities to all actual vulnerabilities in the ground truth. Precision is the fraction of correctly identified vulnerabilities, according to the ground truth, among all identified vulnerabilities. Figure 4.4 shows the precision and recall values for the

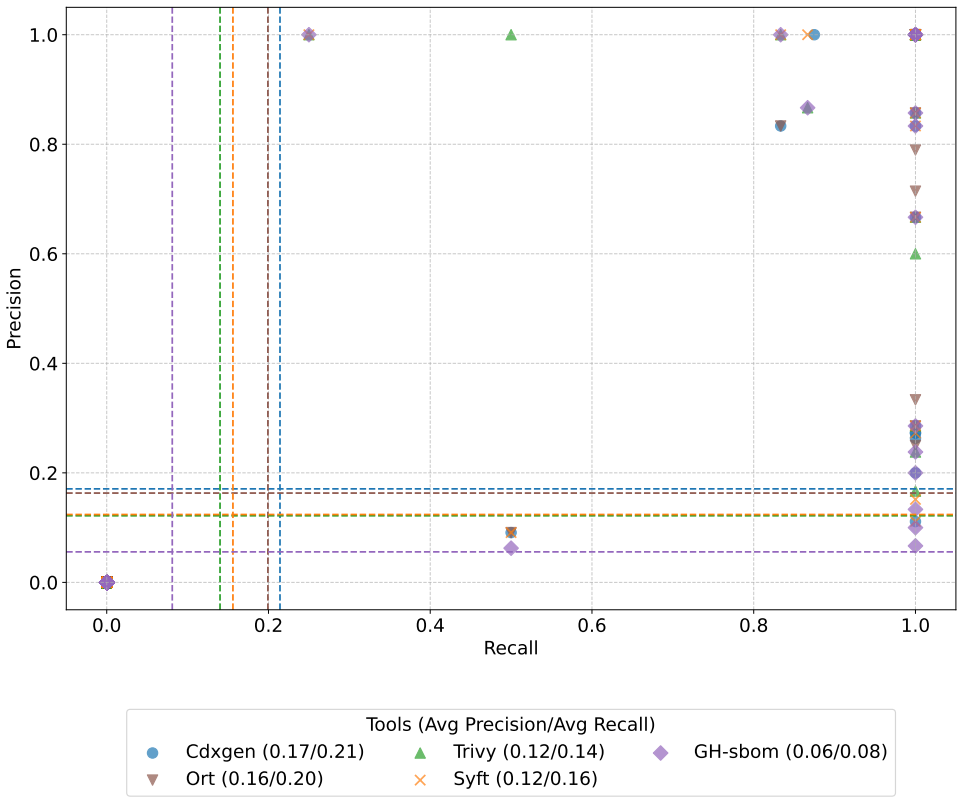


Figure 4.4 Precision and Recall for the vulnerability scans conducted through SBOMs generated by each of the selected SBOM generation tools.

analyzed SBOM generation tools. All the tools show low precision and recall average values, with cdxgen leading the group with 0.17 and 0.21 average precision and recall, respectively.

As shown in Figure 4.5, the analyzed tools have a very high number of false positives, while false negatives are present in a more manageable magnitude. For example, 99.5% (978 FP / 5 FN) of the misclassified vulnerabilities are false positives for cdxgen and 97.8% (926 FP / 21 FN) for Syft. While an overestimation is generally considered better than losing vulnerabilities [1, 15, 5], these numbers are out-of-scale, causing burdening during vulnerability investigation.

By randomly sampling projects with at least a false positive I identified the following reasons:

1. The SBOM list dependencies that are not collected during installation, and
2. the vulnerability is reported with a vulnerability identifier different from the one in the ground truth

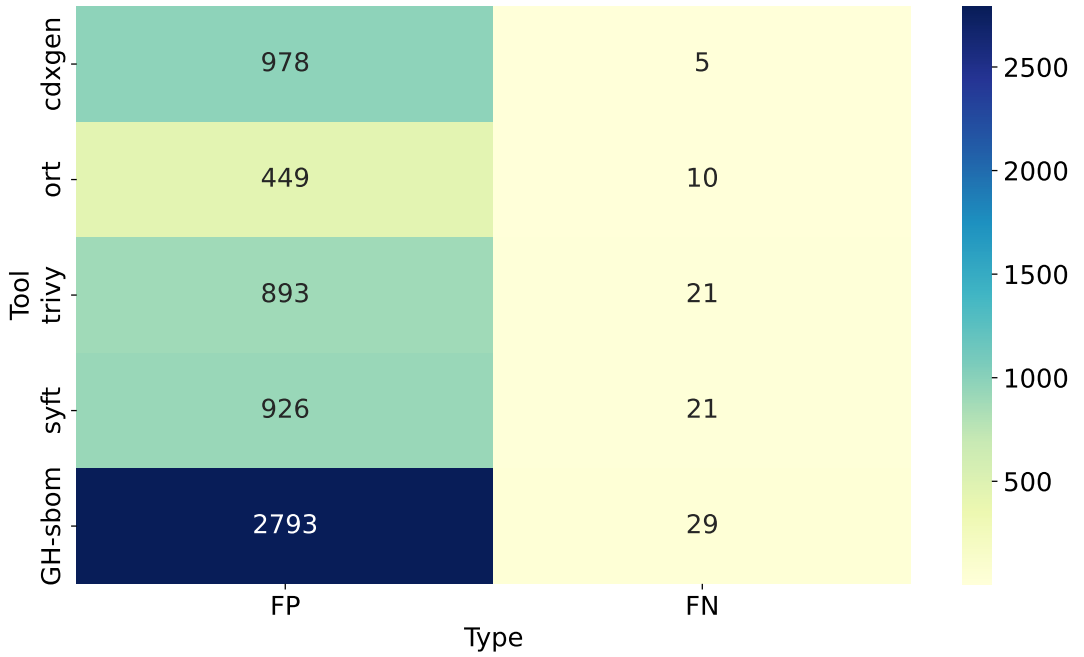


Figure 4.5 False Positives and False Negatives by Tool

The first reason is caused by the presence of metadata files inside of projects listing dependencies that are *not* actually collected during installation. By analysing the dependencies contained in the SBOMs I confirm that on average 75% of the dependencies listed in the SBOMs are not actually installed. When a package is built only the files specified inside of the metadata file are included in the package (see Section 2.2.2). Those are the dependencies that the package manager collects during the installation. This misalignment between files listing dependencies, and files used for installing dependencies, causes SBOM generation tools to generate a huge amount of false positives during vulnerability scanning.

The second reason is linked to a limitation of our methodology (see Section 4.6) and affects a limited number of vulnerabilities making it negligible. However, it highlights the problem of using vulnerability databases as the source for vulnerability scanning. These databases may be partially out-of-date, or experiencing problems in the collection of security advisories. Recently NVD had trouble collecting CVEs for a long time, causing many issues for tools relying on such database [153].

Takeaway: (1) SBOM generation tools do not properly support multiple package managers. (2) SBOM generation tools do not consider how dependencies are actually collected by Python's package managers.

Table 4.3 Comparison of average values for Jaccard similarity, Precision, and Recall for PIP-SBOM against state-of-the-art tools.

	cdxgen	ORT	Syft	Trivy	GH-sbom	**PIP-SBOM**
Jaccard Similarity	49.77%	36.50%	26.33%	23.63%	23.98%	**78.39%**
Avg Precision	17.08%	16.31%	12.39%	12.17%	5.57%	**80.95%**
Avg Recall	21.42%	19.93%	15.61%	14.01%	8.10%	**80.26%**
F. Poss. / F. Negs.	978/5	449/10	926/21	893/21	2793/29	**47/3**

4.4.2 RQ2: Trying a Different SBOM Generation Method

Implementing PIP-SBOM, a PIP extension using the dependency resolution algorithm native to the package manager, I drastically improve the vulnerability assessment of the dependency network. As Table 4.3 reports, almost 80% of the vulnerability reports match the ground truth. *No one of the analyzed pre-existing tools is able to provide the same performance.*

PIP-SBOM achieves a 64% increase in precision and a 59% improvement in recall for vulnerability scans, outperforming the best-performing existing tool.

Having a limited delta between average precision (80.95%) and recall (80.26%), our proposed tool allows a vulnerability scanner to effectively identify vulnerabilities in the dependency network requiring a limited manual effort to discard false positive vulnerabilities. Thus, it has only 47 false positive vulnerabilities. While the false negative vulnerabilities have the same magnitude as the other SBOM generation tools, false positives are way lower than other tools. By providing developers with a manageable number of vulnerabilities to check, I want to push towards the adoption of SBOMs as a useful means for security.

Concerning projects' security assessment differing from the ground truth: I manually reviewed these differences, and all of them are due to issues with vulnerability identifiers (See Section 4.6 for details).

Takeaway: PIP can be extended by re-using most of its code to generate an SBOM that drastically improves the vulnerability assessment results.

4.5 Discussion

SBOM generation is a hard problem. Since software is usually constructed on third-party components, having a complete and correct SBOM represents a great improvement for security, and many other aspects of software usage — e.g., licensing. Using SBOM as input for vulnerability scanners speeds up the analysis of the software's dependency network, also providing clear information on the vulnerable dependencies and their transitive dependencies. However, state-of-the-art tools do not provide SBOMs that can be efficiently used for vulnerability assessment in the Python ecosystem.

I identified two main causes for this problem:

1. SBOM generation tools do not provide support for the high number of package managers used for Python projects.

2. They do not correctly build the dependency network.

However, it is possible to greatly improve the current situation, without much of an effort. Since dependencies are collected by package managers, using them for SBOM generation represents an efficient approach. I tested this *native* generation method by implementing it in PIP.

4.5.1 Implications

Our results can be interpreted in two ways:

- Developers currently relying on tools like shiftleft scan or Grype are missing out most of the actual vulnerabilities in the dependencies of the analyzed software.

- The problems affecting the SBOM generation tools can be easily solved by adding just some changes to package managers. Once the SBOM is correctly generated, it greatly benefits the vulnerability assessment.

These implications can be easily transferred to other languages and ecosystems. Most sofware ecosystems do not support provenance and SBOM generation. According to OpenSSF only npm ships the generation of SBOM for the hosted packages, while only npm and Homebrew provide provenance information [185].

On the other hand, having a proper SBOM enables great result in the vulnerability assessment. With just a few changes to the package manager, the SBOM can be shipped with a project.

4.5.2 Recommendations

Our recommendations are addressed to communities of software ecosystems. There is a need to include SBOMs as part of projects by default. Thanks to our proposal, I showed that this is a possible and easily achievable goal. The following recommendations may help those communities:

- Using centralized SBOM may provide a common knowledge base, easily accessible, and distributed for all developers. GH-sbom provides such a feature, however, it is not always supported and it is limited to providing SBOM for the latest commit on the main branch. Fostering discussion on having such a powerful tool may relieve some of the efforts on the single communities.
- Push for a standard set of files to list the dependencies installed in a project. Most ecosystems already use this approach, for example, npm (`package.json`), and Cargo (`Cargo.toml`). Multiple data sources may be confusing and can introduce unexpected dependencies and associated vulnerabilities.
- Work for a SBOM generation tool implemented inside of the ecosystem package manager(s). As I showed, SBOM generation can largely be improved by using a native generation method. SBOMs can benefit from ecosystem-specific information that may improve SSC transparency for that specific ecosystem.

4.6 Threats to Validity

Projects Collection. Our sample can be generalized to the whole package population on PyPI, with the margin of error stated in Section 4.1.1. While it is relevant because of generalization, it may miss specific corner cases providing insightful knowledge. I internally cross-validated the sample with other random samples taken from PyPI without filtering on package managers. The cross-validation confirmed the accuracy of our results with an error of $\pm 2\%$.

SBOM generation tools Selection. The selection is manually conducted by filtering SBOM generation tools based on the criteria discussed in Section 4.1.2. The list of tools has been reviewed by more than one author, and I agreed on the five selected tools. However, a degree of subjectivity would be present in the selection, leading to the exclusion of other potentially effective tools. I tested the tools that were eligible for our study and excluded the

ones that were not functional for our research goals. The manual testing may have caused the exclusion of potentially effective tools.

Ground Truth. The reliance on pip-audit for establishing the ground truth introduces potential limitations, as this tool may not detect all relevant vulnerabilities, which could impact the baseline used for comparing other tools. pip-audit is largely used to detect vulnerabilities in the dependency installed inside of an environment. However, it is subject to vulnerability databases as well as vulnerability scanners. Since our measurements with Grype were conducted at the same time as the ground truth generation, I argue that any potential gap was avoided.

Evaluation Methodology. I experienced false positives and negatives while evaluating vulnerability scan results. Some are due to a missing vulnerability identifier inside the vulnerability scan result. These errors can be easily fixed by establishing a unique vulnerability database mapping vulnerabilities to their identifiers on the various databases. I manually fixed the issue in our dataset since such an event is rare.

Evaluation Scope. The methodology and results are tailored to Python, while I consider these results extensible to other languages, I cannot state their transferability. Our study wants to raise concerns about the reliability of current SBOM generation tools for security analysis of dependency networks, and to push on the development of native SBOM generation method by package managers.

4.7 Related Works

Research produced a large amount of literature on SBOM in the last period. Related to this work, it can be divided into two categories: research on (1) technical challenges and (2) adoption.

Technical Challenges Yu et al. [211] conduct a differential analysis examining the correctness of SBOMs generated by four SBOM generation tools. The analysis is conducted over seven program languages. They highlight how the SBOM generation tools have difficulties to correctly finding dependencies.

Torres-Arias et al. [200] conduct a study on the fulfilment of minimum required elements issued by NTIA [149] for SBOMs using the SPDX standard. Similarly, Halbritter and Merli [118] do the same for CycloneDX SBOMs.

Balliu et al. [31] focus on the Java ecosystem with the analysis of the SBOM generated for a known Java application. They provide an overview of the challenges that SBOM generation tools face to generate an SBOM on Java projects. Similarly, Rabbi et al. [163] focus on the npm ecosystem. Cofano et al. [52] conduct a study on the relationship between the Python ecosystem and four SBOM generation tools. They identify challenges posed by the ecosystem and an excess of approximation by the SBOM generation tools. All these works do not look at the impact of the generated SBOM. Differently, I focused on the usage of the SBOM to understand to what extent this technology can be effectively adopted.

SBOM adoption While the SBOM represents a resource for SSC transparency, enabling both functional and security testing, its adoption is delayed. The efficacy of SBOM for security has been recently reported by Sharma et al. [178]. They propose a technology using SBOM to mitigate vulnerabilities affecting Java applications.

Enck et al. [78] report that the use of SBOM for security is debated among practitioners. More than a year later the landscape is not brighter. Zahan et al. [212] report that attended of the S3C2 [11] Industry Summit were sceptical about the adoption of SBOM. The problems come from including SBOM generation in the CI/CD pipeline, however, it has been suggested *to embed SBOM generation within the build template as part of a standardized build pipeline and making SBOM generation a mandatory task when setting up CI/CD templates*. I show that this approach can be easily adopted without breaking the build process, at least concerning the Python ecosystem.

4.8 Conclusion and Future Work

This study shows how SBOM generation heavily affects vulnerability scans of software. The current state-of-the-art SBOM generation tools cannot provide SBOMs accurate enough for vulnerability scanner to identify more than 20% of the vulnerabilities actually present in the software. This problem is due to the lack of support for Python and the techniques adopted to solve versions of Python project dependencies. I proposed PIP-SBOM, a proof of concept implementation extending PIP to generate an SBOM directly from the Python package manager. The performances provided by our PoC for vulnerability assessment

suggest that using native resources of the ecosystem — e.g., the package manager — to generate the SBOM may largely improve the overall security posture of software.

Some future works in this direction can support the generation of SBOM in different software ecosystems. Moreover, research in the use of SBOM for security should provide further elements to push software communities and developers to adopt this technology.

Part II

Software Trust

Chapter 5

An Empirical Study on Reproducible Packaging in Open-Source Ecosystems

Reproducible builds to secure build integrity. While the world runs on open-source software, software is rarely consumed as source. The software build process that transforms source code to its consumed artifact is a long-standing security risk. In his 1984 Turing Award Lecture, Thompson [197] described a process to create an undetectable backdoor in software by modifying the compiler that compiles a compiler, leaving no trace of the backdoor in source code. Nearly 40 years later, the 2020 attack on *SolarWinds* subverted the build system to produce binary artifacts signed with SolarWinds's official code signing keys [121], and the 2018 *event-stream* and 2024 *xz* supply chain attacks try to inject malicious code into deployed components that are not visible in the source code [131].

The best known defense against build system attacks is creating *reproducible builds* [132]: A build is reproducible if executing the build on two or more different machines (e.g., by different organizations) produces a bitwise-identical artifact when given the same source code, build environment, and build instructions. A reproducible build provides strong evidence that the build process was not tampered with and that the resulting artifact corresponds to the source code, which is particularly important for high-profile projects (e.g., Tor [156] and Bitcoin [73]) and essential digital infrastructure. Approaches such as CHAINIAC [148] build on top of reproducible builds to create collectively verified builds. For example, it is unlikely that Google, Microsoft, and Amazon's build processes will be simultaneously compromised, especially when the compromise of any subset of the parties is easy to identify.

Achieving reproducible builds is widely viewed as very hard [205], and currently, only a few developers invest effort into reproducible builds. There are many intricate reasons why a build may not be reproducible in practice [132] – for example, time, environment

variables, and build location may be embedded in binary executables and archive files that package artifacts. In addition, there are many more potential sources of system differences and non-determinism in builds, such as different system locales, pseudorandom number generators used during compilation or code generation, and process scheduling. While the Debian project has spent a decade removing sources of unreproducibility and has achieved over 95% reproducible builds on AMD64 [134], reproducible builds remain a hard challenge – a recent interview study of practitioners invested in reproducible builds highlighted many technical and social challenges of resolving build unreproducibility [85].

Reproducible component builds to secure package ecosystems. In this paper, we consider reproducible builds in a novel context: the reusable *components* distributed as *packages* with package managers in popular software ecosystems, including npm (JavaScript), Maven (Java), PyPI (Python), Go, RubyGems (Ruby), and Cargo (Rust).[1] Today, most applications are built with reusable open-source components, and build processes usually rely on the archives distributed with package managers rather retrieving the original source code from repositories.

Attackers are actively exploiting the gap between the source code in a repository and the release of a package. In 2018 the account controlling the popular PyPI package `ssh-decorate` was hijacked to upload a version that collected users' SSH credentials to send them to a remote server [51] – an independent build of the package would have identified that the hash of the version in PyPI differed from the one built from the source. The *xz* attack [112] also exploited this gap: While the malicious code was obfuscated in the source code repository as a malformed compressed archive used as a negative test case, it was only enabled by an Autoconf file that was *not* in the repository. The released dist tarball of xz included build instructions generated by the malicious Autoconf file. Interestingly, the Debian Reproducible-Builds project *did not* detect the *xz* attack, because they used the released dist tarball as their source. Even though we do not study reproducibility of C/C++ dist tarballs, this is exactly the kind of scenario of manipulated distributed packages (source or binary) we address in this paper.

Although the primary focus of the discourse on reproducible builds is on applications rather than components, the limited prior research on component reproducibility paints a gloomy picture, reporting severe reproducibility issues for most components studied in npm and PyPI. Specifically, Vu et al. [205] attempted to match the code in over 2,000 popular

[1]We use the terminology of *components* and *applications* common in the discourse of software supply chains. A component here can be any reusable software artifact, including libraries, frameworks, infrastructure tools, and non-code artifacts. Components are typically, but not necessarily distributed as *packages* with a *package manager*.

PyPI packages with their original source code (without actually building the packages) and found a wide range of differences, most benign. Furthermore, Goswami et al. [114] studied the build reproducibility of over 3000 versions popular npm packages and found significant challenges resulting from version drift of build tools.

In our work, we explicitly adopt a much broader scope, comparing practices in multiple ecosystems and analyzing the role of package managers' tooling: Open-source infrastructure has formed distinct communities, interdependent *ecosystems*, often around languages, frameworks, and platforms, often with distinct practices and tools [40, 139, 69]. Among others, practices and tools for packaging and indexing artifacts differ widely between (a) compiled languages that share binaries and (b) interpreted languages that share archives of source code, sometimes transpiled, sometimes including some binary code extensions, sometimes including code in multiple languages – for instance, the package managers *npm* and *PyPI* have adopted entirely distinct toolchains. Different practices and tools may lead to widely different outcomes for reproducibility. Hence, *we specifically study the difference of component reproducibility across ecosystems and the influence of tooling choices in package managers.*

Research overview. With a quantitative study, we attempt to reproduce a representative large random sample of 4000 packages in each of six packaging ecosystems (npm, Maven, PyPI, Go, RubyGems, and Cargo) and investigate the reasons for reproducibility issues . For clarity, we want to highlight we are studying the role of package managers in reproducibility issues occurrence, and not trying to reproduce artifacts hosted on package registries. Specifically, we answer the following research questions:

RQ1 (Reproducible Package Builds): *How many packages are reproducible* as is *in each ecosystem?* We find that almost all packages in Cargo and npm are reproducible[2], but only very few to none are in RubyGems, PyPI, and Maven. Go packages simply include a reference to a source code repository and do not include any artifacts – therefore component builds are not relevant for Go. This highlights the substantial differences between ecosystems and also seems to initially confirm the gloomy reports of vast reproducibility problems from previous research.

RQ2 (Causes of Unreproducibility): *To what extent are unreproducible package builds caused by the toolchain and fixable through toolchain changes?* Analyzing the causes of unreproducibility, we find that nearly all unreproducible package builds were caused by nondeterminism in the build process (e.g., paths, time, file permissions, locale) that can

[2]For simplicity we use the term *reproducible* when the package manager tooling does not insert any reproducibility issue

be controlled by the package manager's tooling, either through configuration or through small changes to the tools. With configuration or tooling changes in RubyGems, PyPI, and Maven, almost all packages become reproducible in these ecosystems. This result paints a much more optimistic picture than prior reports and our initial findings: Our results highlight that small changes to ecosystem-wide tooling have a substantial lever to improve package reproducibility to the point that almost all packages are reproducible in each studied ecosystem, without having to convince every single package maintainer to take actions to ensure reproducibility.

RQ3 (Native Code Extensions): *To what extent does the presence of native code inside of packages influence build reproducibility?* Ruby, Python, and even JavaScript packages can include native code extensions. Analyzing specifically packages that contain native code extensions, even though the package managers in these ecosystems incorporate native code differently (e.g., compiling it into the package vs. including source code), our study shows that native code extensions rarely negatively influence reproducibility, with reproducibility rates in each ecosystem almost identical to that of packages without native code extensions. This is another positive finding, highlighting that native code extensions do not pose additional barriers in practice.

RQ4 (Reproducible Builds and Compilation): *To what extent does compilation of package dependencies into distributable artifacts of application affect the build reproducibility?* As components in ecosystems are usually used to build applications, we explore the downstream effects of component reproducibility (or a lack thereof) for applications relying on components in the ecosystem. We find that application builds using the studied packages are reproducible for 100% of the Go packages and greater than 90% of the Maven packages. The results for Cargo were more nuanced: due to timestamps used as part of cryptographic signatures, potentially all packages can cause downstream build reproducibility problems; however, in practice, these cryptographic signatures will not change, and greater than 80% of the studied packages had reproducible builds. This is a final supporting piece that also points toward the message that package reproducibility may be a smaller obstacle than originally expected.

Recommendations. Overall, our results largely paint a positive picture for the studied software ecosystems. Our results indicate that package managers can take advantage of reproducible builds to protect against build system or account compromise by doing the following: First, the identified toolchain issues must be addressed by providing more options to control nondeterminism during the build process. Second, package manager tooling should adopt default configurations that create reproducible builds without having each maintainer

configure their build. Finally, package managers *must* make `buildinfo` files available. Our study did not consider build tool version drift, and prior work [114] identified this as a significant challenge for npm. Without `buildinfo` files, it is nearly impossible to compare our independent build with the packages distributed by package managers.

Contributions. In summary, we contribute a large-scale quantitative analysis of the state of reproducible builds across six software ecosystems, contrasting the prevalence of reproducible package builds and infrastructure changes that can support increased reproducibility for each of the ecosystems respectively. Our results have implications for understanding how to improve the state of reproducibility on a macro-ecosystem level as well as concrete changes to support individual package users and developers.

The source code, data, and additional material for this Chapter are available in our replication package [34] .

5.1 Research Design

In this paper, we conduct an in-depth empirical study of reproducible builds in six packaging ecosystems to answer the four research questions we outline in the introduction: • **RQ1:** How many packages are reproducible *as is* in each ecosystem? • **RQ2:** To what extent are unreproducible package builds caused by the toolchain and fixable through toolchain changes? • **RQ3:** To what extent does the presence of native code inside of packages influence build reproducibility? • **RQ4:** To what extent does compilation of package dependencies into distributable artifacts of application affect the build reproducibility?

We use the following high-level research design: For each packaging ecosystem, we randomly generate a large, representative sample of packages and attempt to build them under varying environmental conditions from source code in their open-source repositories. This allows us to quantitatively study reproducibility rates across ecosystems. Based on our findings, we then explore sources of unreproducibility and corresponding interventions in package manager toolchains to quantify how tooling changes affect reproducibility rates.

5.1.1 Analyzed Packaging Ecosystems

We focus on ecosystems revolving around package managers for specific programming languages.

To select the package ecosystems used in our analysis, we searched for ecosystems representing a large community with over 100,000 packages each; we then used an information-

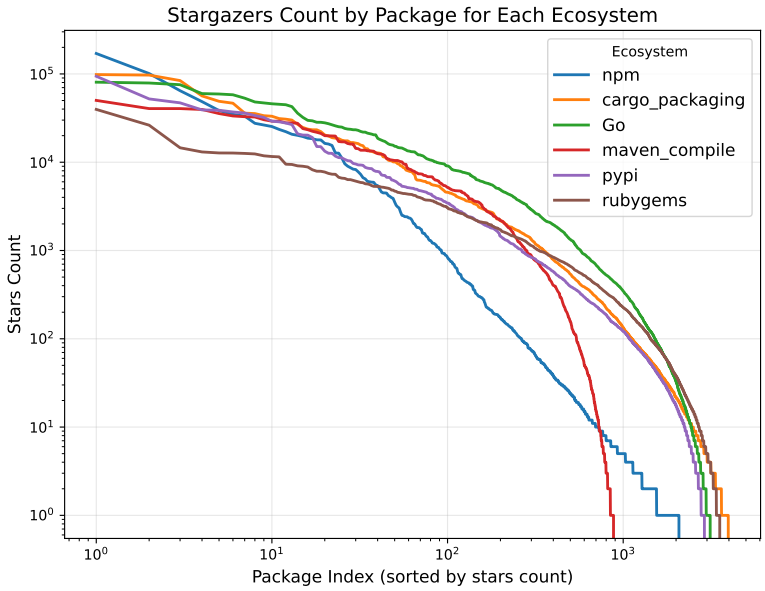


Figure 5.1 The samples used in our analysis allow us to obtain the practices of the average developer dealing with reproducible builds. Thus, most of packages in the samples have a very low popularity, however, our samples also contain some very popular packages.

oriented selection strategy (maximum-variation cases, following case study research logic [84]), to identify a set of ecosystems with different characteristics – e.g., native code extensions, dynamic specification files – and with different strategies in their distribution model – e.g., compilation output: binary/bytecode, plugin-based build, package indexing. This process yielded the following six ecosystems: Cargo, Go, Maven, npm, PyPI, and RubyGems. We argue that this set of ecosystems represents the current software landscape. Their position regarding reproducible builds is critical for global software supply chain security posturing.

5.1.2 Sample of Packages

Given the size of the ecosystems and the prohibitively high cost of repeatedly building packages, it is infeasible to analyze all 100,000 to 5 million packages in each ecosystem. Instead, we analyze a large, representative, random sample from each and make statistical generalizations.

Packages (RQ1, RQ2) To generate our sample of packages for each ecosystem, we begin with a list of all packages in the packaging ecosystem indexed by ecosyste.ms. In order to minimize the chance of build issues during the analysis, we filter packages by checking for the presence of ecosystems' specification files — e.g., `package.json` for npm and `setup.py` /

`pyproject.toml` for PyPI—, discarding packages without them. We then generate a random sample of 4000 packages from this pool. We intentionally did not select only the most popular projects so that we can generalize our findings to the entire population of packages in the ecosystem, rather than merely reporting numbers about the most popular packages. Figure 5.1 shows the popularity of packages considering the number of stars. It shows our sample to contain a few very popular packages and a lot of low-popular ones. Having such a distribution of popularity allow us to obtain practices of average developer. With the relatively large sample size, results derived from this sample afford high generalizability with less than 1.54% margin of error at 95% confidence levels by standard sample size calculations. The precise margin of error varies slightly between ecosystems due to their different population sizes and population proportion of reproducibility, but the differences are negligible.

Packages with native code extensions (RQ3) To concentrate on the effect of native code on reproducible builds, we generated a sample of packages with native code extensions and a second sample of packages without native code extensions. To generate these samples, we first started by dividing the packages from our RQ1 sample according to whether native code extensions were present and designated them to the appropriate sample. This resulted in a sample of 398 (PyPI), 394 (RubyGems), and 129 (npm) packages with native code extensions and a sample of complementary cardinality without native code extensions. However, because we wanted these samples to contain 4000 packages so they were comparable to the RQ1 sample, we continued to draw fresh random samples from the whole package population until we found 4000 packages for each of the samples. The detection of native code extensions in packages is ecosystem-dependent: It is based on the presence of specific files (e.g., the extconf.rb file for RubyGems) or configurations (e.g., the ext modules in the setup.py file for PyPI). The margin of error is again less than 1.54% with 95% confidence. The complete list of criteria can be found in the replication package [34].

Compilation Process Impact (RQ4) Similarly to RQ3, to explore the impact of the complication process on reproducible builds we generate a sample of packages with packages that can be compiled. For Cargo, Go, and Maven, we collected large random samples of 4000 packages each by applying the following filters.[3] For Cargo, the Cargo.toml file must contain a binary target to allow the package manager to compile the project. For Go, an entry

[3]The other ecosystems, i.e., npm, PyPI, and RubyGems, do not have a proper process to generate a compiled artifact.

point function, i.e., the main function, must be present among the project files. For Maven, the maven-compiler-plugin must be listed in the pom.xml specification file.

5.1.3 Reproducibility analysis for packages (RQ1–3)

For each package in our samples, we identify the corresponding open-source repository and clone the most recent version of the code. We then use the ecosystem's default command to build the package (e.g., `pip wheel .`). The list of package managers and commands that we used is available in our replication package [34]. When the open-source repository is missing or the build fails the package is reported as missing and discarded from the analysis since it does not offer information on its build reproducibility. In those cases, a new package matching the same criteria is randomly selected from the whole package population to replace the discarded one. We discarded about 800 packages for each ecosystems before reaching the required number of 4000. The build failures were caused by missing system libraries and mistakes in specification files. The package build reproducibility of all packages is tested using `reprotest` on the same machine. Fixing the build infrastructure specifications — e.g., toolchain versions, dependencies, operating system — makes it possible to focus on the impact of reproducibility issues related to the package manager. The tool is set up with a build command and a build output for each ecosystem. For example, `reprotest -variations=+time 'pip wheel -w dist <package_path>' 'dist/*.whl'`: the time -variation is applied to the build of packages for PyPI and the resulting wheel files are tested for differences. `reprotest` runs one time for each variation. Our replication package contains the full catalog of the used variations. We consider a build as reproducible when `reprotest` does not report reproducibility issues for any of the tested variations. Failing variations are recorded to subsequently investigate the causes.

We explicitly do not compare the compiled artifact with the artifact uploaded to the package manager for two reasons. First, identifying which specific revision of the source code repository was used to produce the released package is nontrivial and the subject of extensive research [205, 181, 46] but entirely orthogonal to our exploration of whether a package can be reproduced from the same source code. Second, environmental dependencies such as compiler versions, system libraries, and operating system configurations can vary significantly across different build environments, leading to potential discrepancies in the compiled artifacts, which is again orthogonal to our research on whether the package manager build process influences a package build reproducibility. Hence, we only compare build

outcomes under different environments from the latest revision of the source code in the package's repository.

The results of the tests show which reproducibility issues have the biggest impact on build reproducibility. We look for the reasons for unreproducibility behind the package manager implementations'. The majority of the analysis is done manually, by looking at particular portions of code that appeared to be the cause of an unreproducibility problem. We created several automated scripts to search for possible unreproducibility causes when reproducibility issues were not triggered by the package manager directly. For example, one such script looked for dynamic dates in specification files.

5.1.4 Reproducibility analysis for compilation (RQ4)

We examined Cargo, Go, and Maven since they offer a proper compilation method. The first two ecosystems yield binary files, while the last one generates an archive with compiled Java bytecode. We employed the same methodology as for the other research questions to examine the influence of compilation. The configuration of the `reprotest` tool uses the same variations used for RQ1–3, but it receives different build commands and builds outputs (such as `cargo build`). We use the assumption that the compiler requirements are known and that reproducibility problems are not the result of the compiler since we are interested in examining how package manager operations affect reproducible builds.

5.1.5 Limitations and threats to validity

Evaluating build reproducibility includes technical details varying by the level of abstraction taken into consideration. For the purpose of this study (package manager impact on reproducible builds), we designed our methods to identify reproducibility issues at the build infrastructure level. By excluding reproducibility issues originated by the system level, e.g., architecture specifications, we may miss unreproducible builds caused by the package manager because of platform-specific causes.

While build reproducibility has a binary result (reproducible or not), identifying causes of reproducibility issues is not as straightforward. We used a specific tool, `reprotest`, in an attempt to identify the causes of reproducibility issues; however, `reprotest` may be subject to unexpected bugs and design issues that have the potential to impact our results. To mitigate this risk, we carefully reviewed the `reprotest` implementation and currently open issues on the code repository, identifying minor potential sources of false positive unreproducible

builds. After patching these (potential) implementation issues, we assume that the tool's results can be considered scientifically accurate.

We can statistically generalize our results to report trends for reproducible builds for the entire ecosystems with tight error margins. This method analyzes the general, representative behavior of package builds, but we may miss the behaviors of individual developers; for example, the behaviors of a few developers of high-impact packages may differ from those of the general population.

5.2 Results

I present results of our experiments by research question.

5.2.1 RQ1: Reproducible Package Builds *As Is*

Attempts to reproduce packages *as is*, without any changes to package manager toolchains, yields wildly different results across ecosystems as shown in Figure 5.2: A first group of ecosystems achieves reproducible builds for nearly all the sampled packages. Specifically, Cargo and npm have 100% reproducible package builds. A second group of ecosystems achieves very low percentages of reproducible package builds. Specifically, Maven, PyPI, and RubyGems have 2.1%, 12.2%, and 0% of reproducible package builds, respectively.

Takeaway: Rates of reproducible builds vary widely between ecosystems, with some ecosystems having all reproducible packages whereas others have reproducibility issues in nearly every package.

5.2.2 RQ2: Tooling Sources of Reproducibility Issues and Fixes

Fortunately, the situation is not as bleak as it might seem when analyzing the sources of reproducibility issues. In Figure 5.2, I report that even in package ecosystems with low package reproducibility as *as is*, most packages are reproducible if considering package manager toolchain configurations or small patches to package manager tooling. For example, just by changing the configuration options of the package manager toolchain, Maven and RubyGems increase to 92.6% and 97.1% reproducible package builds, respectively. As I discuss in Section 5.2.2, our analysis led us to discover small changes that can be made to

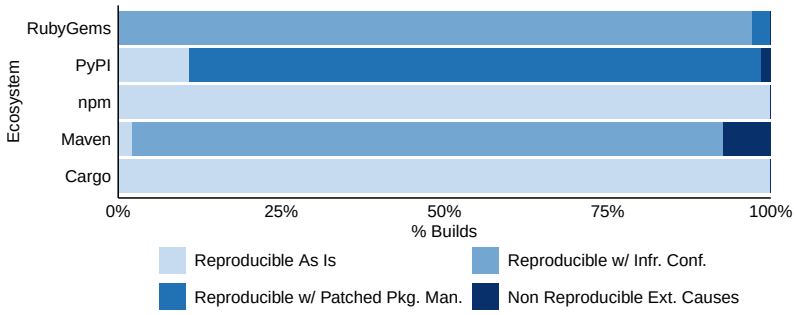


Figure 5.2 This figure shows the reproducibility of package builds across different ecosystems, categorized as: i) reproducible as is, ii) reproducible with infrastructure configuration as requested by the package manager, iii) reproducible with a patched package manager, and iv) unreproducible due to external issues not related to the package manager.

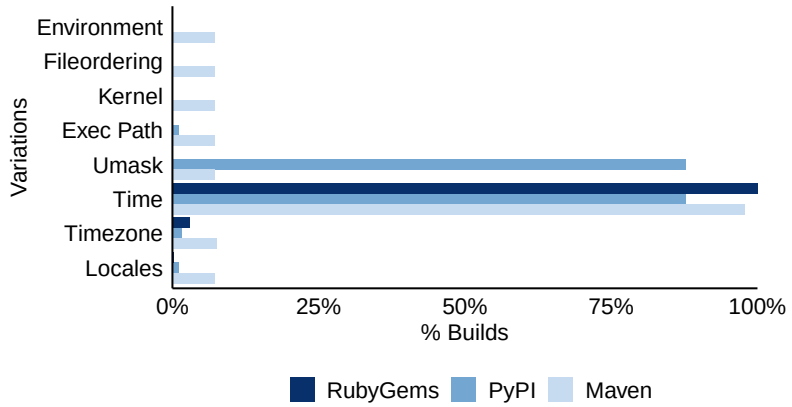


Figure 5.3 Percentage of builds per ecosystem that were unreproducible because of different variations.

PyPI's package manager that increase it to 98% reproducible package builds. Thus, what might be perceived as very low package build reproducibility is, in fact, quite promising. In Section 5.2.2, Section 5.2.2, and Section 5.2.2 I detail how changing the build configuration and tooling drastically increases the reproducibility of package builds, and how some causes of unreproducibility cannot be easily solved, contributing to answer RQ2.

Reproducible with Infrastructure Configuration

I applied solutions proposed by the selected ecosystems to study how infrastructure configuration affects the build reproducibility. By reviewing the `reprotest` logs, I found that timestamps and permission masks are the largest contributing factors, as Figure 5.3 shows. In particular, timestamp metadata impacts most of the ecosystems: 92.4% in Maven, 87.77% in PyPI, and 97.1% in RubyGems.

All five studied ecosystems use archives to distribute package artifacts. However, Cargo and npm drastically reduces (actually removes in our sample) the presence of reproducibility issues caused by hard-coding such fixed values in their package managers' code. The other three ecosystems use a different approach: they allow developers to communicate a specific build time by configuring the package manager. This approach makes builds unreproducible using the package manager as is.

The packaging infrastructure can be configured to help remove sensitivity to timestamp metadata. For example, the PyPI and RubyGems packaging infrastructure can set the build time using the `SOURCE_DATE_EPOCH` environment variable. However, the variable value must be communicated along with the package, e.g., via a `.buildinfo` file [64]. In contrast, Maven allows the developer to use the `outputTimestamp` POM specification file property to set a fixed timestamp in the package metadata. Unfortunately, this solution suffers from two major limitations: (1) the developer has to set it in the pom.xml file — it is not by default[4]; and (2) Maven plugins have to implement this feature to make it effective.

In general, the studied package managers do not require or suggest that configuration of the build infrastructure is necessary. Among the three ecosystems, only Maven's documentation clearly explains how to achieve build reproducibility. This lack of information for other ecosystems, combined with the challenges of communicating build parameters (e.g., `SOURCE_DATE_EPOCH`) is a large factor for unreproducible package builds.

Finally, PyPI package build reproducibility is largely impacted by both timestamp *and* umask values. Umask values cannot be addressed via packaging infrastructure configuration and are discussed in Section 5.2.2. That said, different package manager tools operate differently. Recall Section **??**, `pip` is a frontend to deal with multiple building backends. It refers to the specification file to invoke the right backends. I found that the `flit` and `hatch` building backends fix the archive metadata similar to Cargo and npm. Since 12% of PyPI builds use either `flit` or hatch, they are "reproducible as is." However, most of the remainder of the PyPI ecosystem suffers from archive metadata reproducibility issues.

Takeaway: Configuring packaging infrastructure to control timestamp metadata makes over 90% of Maven and 97% of RubyGems package builds reproducible.

[4]During the camera-ready preparation, an automation was proposed and implemented in Maven 4.0.0-beta (issues.apache.org/jira/browse/MNG-8258)

Reproducible with Patched Tooling

In this subsection, I investigate how patching the packaging tools can address umask and other contributing factors. While controlling timestamp metadata via infrastructure configuration has a significant impact on package build reproducibility, it does not address all issues. As shown in Figure 5.3, umask is also a large contributing factor for PyPI. As for timestamp reproducibility issues, most of the issues caused by umask affect the archive metadata. The source of much of the remaining unreproducibility are dynamic metadata.

Recall from Section 2.4.1 that dynamic metadata allows RubyGems and PyPI developers to define a metadata value in the specification file programmatically. Developers define dynamic metadata using the ecosystem's language or specific properties offered by package managers. The value is then interpolated within the build environment. I identify five root causes in the PyPI and RubyGems package managers: file ordering, locales, umask, time, and timezone. Appedix A.1 shows example of these causes behavior.

The code creating dynamic metadata is defined in packages. Contacting package maintainers to modify their code to produce reproducible package builds would be very time-consuming and may not result in changes to the build specifications. For example, a recent interview study of practitioners working on reproducible builds found that project maintainers are not always receptive to making changes simply to make a build reproducible [85]. I propose patching packaging tools to enable reproducible package builds. Our key insight is that *packaging tools can set default environments and post-process dynamic metadata to ensure build determinism*.

As shown in Figure 5.2, these patches have a drastic impact on the package build reproducibility of PyPI, increasing the percentage of reproducible builds from 12% to 98%. As suggested by the `reprotest` variation data shown in Figure 5.3, most of this increase was the result of addressing umask determinism. RubyGems also received a meaningful impact, increasing the percentage of reproducible builds from 97% to 99.9%. While this increase is small, the ability to achieve almost 100% reproducible package builds is extremely valuable for the ecosystem.

Takeaway: Package managers can set default environments and post-process dynamic metadata to provide reproducible package builds without changing code in individual packages.

Unreproducible Packages

As shown in Figure 5.2, not all package builds could be made reproducible by configuring package infrastructure or patching the package managers. I randomly sampled some of these packages to investigate causes of reproducibility issues.

Recall that PyPI has multiple backends to produce a package, and these backends use either a `setup.py` or a `pyproject.toml` specification file to interface with the build process. Developers using `setup.py` can run arbitrary code during the build. reproducibility issues caused by this code are not easily addreasable. For example, statements using the `os.path.expanduser` function make the build unreproducible because of the `home` variable. This kind of actions can be hardly managed by the build infrastructure because: i) the infrastructure cannot alter external libraries, such as the `os` library in our case, and ii) the package's developer set up an ad-hoc approach, interfering with it can easily break the build process. In contrast, using `pyproject.toml` should address many of the problems caused by dynamic metadata. However, some build backends, e.g., `poetry`, allow developers to call pre-build scripts, declaring them inside of the specification file. These scripts may cause unreproducible builds and cannot be easily patched at the tooling level. I found that 1.6% of the builds for PyPI in our results are affected by these kinds of issues.

RubyGems is subject to similar issues. The cause of unreproducible builds is the run of arbitrary commands launched through the specification file. For example, package version tag can be programmatically set in the specification file by using Ruby Time functions — e.g., `VERSION = "0.0.#{Time.now.to_i}"`. The Ruby Time library does not use the `SOURCE_DATE_EPOCH` variable, making time a reproducibility issue. This event shows how this kind of behavior can be dangerous for reproducibility and how it can hardly be addressed through the build infrastructure.

Maven's challenges are different. Section 5.2.2 discussed that plugins need to support the `outputTimestamp` property declared in the POM.xml specification file. Older plugin versions and custom plugins may not use it. This requires a developer to carefully inspect the plugin used in the package build pipeline since, depending on the pipeline design, a single plugin can impact the build reproducibility [140]. I found that 7% of the builds for Maven are affected by these kinds of issues.

5.2.3 RQ3: Native Code Extensions

As discussed in Chapter 2, PyPI, npm, and RubyGems allow packages to include native code. This subsection studies how the inclusion of native code impacts the reproducibility of

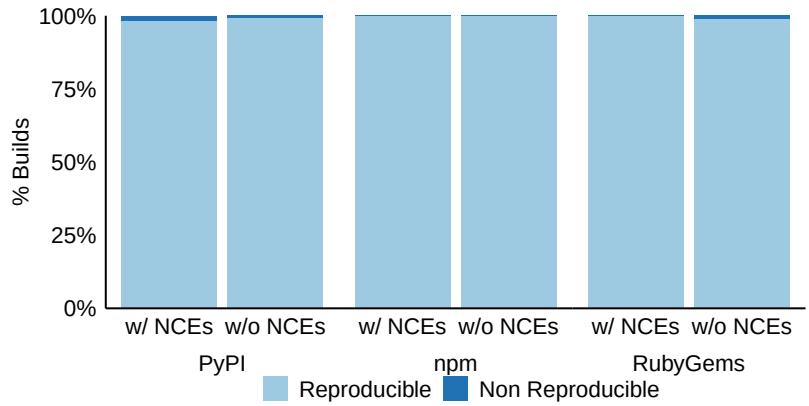


Figure 5.4 Reproducible package builds that take into account extensions for native code extensions (NCEs). In order to concentrate on reproducibility issues caused by native code extensions, these findings are obtained using patched package managers.

package builds (RQ2). Considering the original dataset where packages are not distinguished based on native code extensions, the incidence of native code extensions is 9.96% for PyPI, 9.85% for RubyGems, and – just – 3.24% for npm. Due to the low relative infrequency of native code in these package ecosystems, I created an additional dataset that focuses specifically on packages that include native code (see Section 5.1.2). Our comparison uses infrastructure configuration (Section 5.2.2) and our packaging tool patches (Section 5.2.2) to isolate the impact of native code extensions.

Note that although native code extensions are a - more or less - common feature in interpreted languages, reproducible builds appear to be unaffected by them.

Quantity of reproducible builds

Figure 5.4 compares the reproducibility of package builds of the dataset without native code extensions (w/o NCE) to the dataset with native code extensions (w/ NCE). As shown in the figure, there is a negligible difference between the two datasets. PyPI has 98.43% with native code and 99.28% without. npm has 100% with native code and 100% without. RubyGems has 99.94% with native code and 98.87% without.

Unreproducible builds are caused by the same reproducibility issues discussed in Section 5.2.2. In the end, npm, PyPI, and RubyGems show the same reproducibility issues independently by the presence of native code extensions. Using patched package managers solves those issues as I saw in the previous section. Hence, native code extensions do not impact the build's reproducibility.

Causes of reproducibility

Since C code has multiple reproducibility issues that can affect the build process, I originally expected native code extensions to affect the reproducibility of the package build. Since I do not have any specific pointer to the causes of reproducibility in the analyzed ecosystems, I systematically searched for common reproducibility issues in the C code in 1000 packages randomly picked in the sample, e.g., time-dependent macros. I did not find any occurrence of such issues from this search. Note that our analysis used the same build configuration and tools. If no assumptions are made regarding system configurations, such as compiler specs, there will be fewer reproducible builds. Build specification files (e.g., `.buildinfo` files) are not incorporated into the build processes of any of these three ecosystems.

In PyPI, the Meson backend is largely adopted to compile native code extensions. Meson states to achieve reproducible builds[5] by addressing multiple issues, e.g. rpaths. Similarly in npm, where the gyp module is often used to deal with native code. This additional layer in compilation can be the reason behind the high number of reproducible builds. However, it does not justify the almost complete reproducibility of builds. In RubyGems the system default compiler is directly invoked to compile extensions. Even without additional layer, there is a high number of reproducible builds.

It is difficult to completely understand the reasons behind the non-affection of native code extensions in builds reproducibility. However, it is possible to connect it to two main factors. First, native code extensions are usually used in specific resource-intensive tasks. They deal with computation and are not part of the program logic [144]. Second, native code extensions require more expertise to be set up and properly work than regular code. Developers with higher skills in programming are easily aware of reproducible builds [43, 85], and may deal with them.

Takeaway: Reproducible builds are not affected by the presence of native code extensions in the package. This means that reproducible builds could be easily achieved by solving issues related to the system configurations, such as the compiler specs.

5.2.4 RQ4: Reproducible Builds and Compilation

Some packaging ecosystems allow the shipment of dependencies together with the developed code. The dependencies are inserted inside of the build artifact through the compilation process. Figure 5.5 shows the number of Cargo, Go, and Maven builds that are still reproducible

[5] https://github.com/mesonbuild/meson

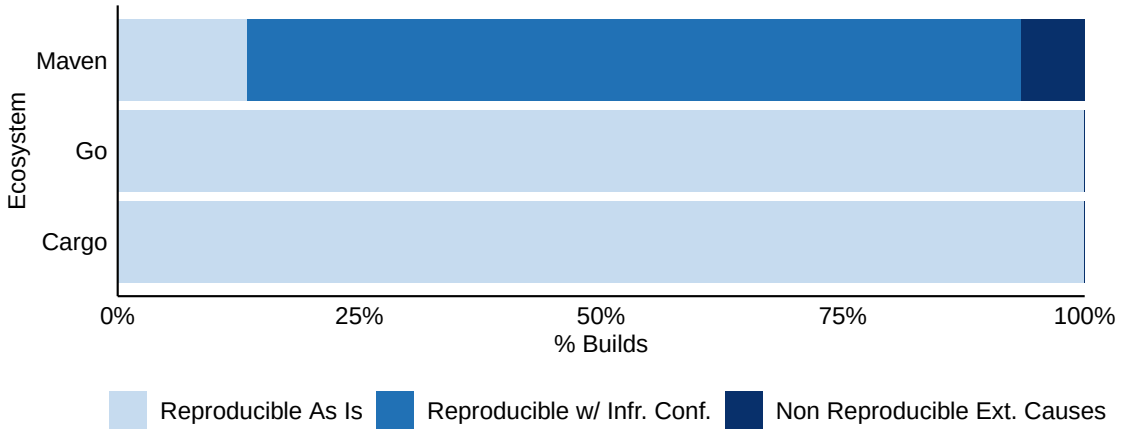


Figure 5.5 Reproducible Builds obtained by compilation. I used unpatched package managers since the compilation process requires different configurations w.r.t. packaging.

after the dependencies have been inserted into a reproducible application. I limited this analysis to Cargo, Go, and Maven, as they are the only ecosystems implementing compilation in their official package managers.

Quantity of reproducible builds

The results point out that the involvement of compilation in the build process does not insert reproducibility issues. Thus, Cargo, Go, and Maven builds result as almost completely reproducible, with 100%, 100%, and 92.5%, respectively. For Maven, the results were obtained using the package manager with the `outputTimestamp` property configured. Thus, in Maven, the produced artifact is still an archive where Java files are compiled to bytecode.

Causes of reproducibility

For Cargo and Go I randomly sampled 1000 packages to understand which are the causes of reproducible builds, and searched for common reproducibility issues in those packages' code. For Maven, I checked whether packages with unreproducible builds were also unreproducible against packaging to understand if there is any specific reproducibility issue inserted by the compilation process. All three ecosystems have initiatives working towards reproducible builds that can justify our results.

Cargo has a proper issue label in its GitHub repository[6] with several closed issues, such as stripping absolute paths — a well-known issue for compilation. This hard pushing towards

[6]https://github.com/rust-lang/rust/labels/A-reproducibility

reproducibility made `rustc` (the Rust compiler used by Cargo) almost deterministic, apart from system specifications such as the linker and compiler versions.

Go takes care to remove potential causes of reproducibility issues during compilation, for example, by trimming absolute paths and by removing build identifiers. The many issues and community discussions on having reproducible builds within Go highlight the interest of this specific community[7]. Moreover, in order to facilitate reproducible builds across different systems, Go strives to establish reproducible builds for the toolchain [57].

Javac (the Java compiler) used in Maven is known to be deterministic [54]. The results confirm this behavior. The reproducibility issues in Maven packages are caused by archive metadata since the compiled Java classes are inserted into a Jar archive, as happens for packaging. Maven builds are affected by the same reproducibility issues discussed in Section 5.2.2, affecting archive metadata and not the compiled artifacts. As for packaging, configuring the build infrastructure addresses most of the unreproducible builds. The remaining unreproducible builds are caused by the same reproducibility issues discussed in Section 5.2.2.

Takeaway: Build reproducibility is unaffected by compilation. Build infrastructures do not introduce reproducibility issues thanks to the community efforts working on them.

5.3 Related Work

Two lines of work are particularly relevant to this study: Studies on the barriers to adopting reproducible builds and studies to identify and repair common reproducibility issues.

Reproducible builds adoption. Recent research has studied the perceptions of reproducible builds and the social challenges to their adoption. Fourné et al. [85] analyzed enablers and blockers for adopting reproducible builds in the open-source community. They report that most industry practitioners *considers reproducible builds to be out of reach* – reproducible builds are seen as valuable but not essential. Other studies confirm that many developers are not aware of reproducible builds when developing their build systems [179, 63]. Butler et al. [43] identified the need for reproducible builds from a security point of view and the reasons for their limited adoption, minimal business impact, and limited awareness and perceived challenges.

[7] https://github.com/golang/go/issues?q=is:issue+is:open+"reproducible+builds"

As discussed previously, two studies focused on technical enablers and blockers for reproducible builds in specific packaging ecosystems: Goswami et al. [114] examined the reproducibility of npm packages by comparing the build output of upstream repository code against artifacts stored on the npm registry. Vu et al. [205] did not directly focus on reproducible builds, but they argued for reproducible builds as a security solution to phantom artifacts, highlighting how difficult it is to achieve reproducible builds.

Technical challenges. Lamb and Zacchiroli [132] reviewed the landscape of reproducible builds within the Debian community and cataloged common causes of reproducibility issues. The *reproducible-builds.org* project [41] similarly provides a detailed catalog of common issues and typical solutions, along with other educational materials, resources, and tools. Also, many tools exist to facilitate reproducible builds especially in the Debian community.

Bajaj et al. [30] evaluated build reproducibility over time in Debian and Arch Linux, computing the likelihood of non-reproducible packages becoming reproducible and identifying the root causes behind non-reproducible builds.

Ren et al. proposed three tools for the localization, root cause tracing, and *patching* of non-reproducible builds: RepLoc [168], RepTrace [169], and RepFix [170] respectively. These kind of tools are intended to help developers to perform targeted changes to make their builds reproducible.

The complexity of these challenges increase in the context of packaging ecosystems. Different community of developers have different values and habits. Bogart et al. [39] analyzed how ecosystems differ in practices, policies, and tools, highlighting the role of community values in shaping the ecosystem and the insularity of such communities. Moreover, the dependency network evolution plays a critical role in the maintainability of reproducible builds over time. Decan et al. studied dependency network and their evolution focusing on single ecosystems [65, 67] and comparing multiple ecosystems [66, 68, 69].

Zooming out, a lack of reproducibility is often also a concern for technical artifacts in academic papers, with many studies identifying that code released with papers is often not compilable (not even asking for bit-for-bit identical binaries) due to missing environment specifications, missing dependency declarations, and various sources of nondeterminism [e.g., 53, 157, 21, 45, 209]. Although reproducibility of academic artifacts has additional challenges beyond reproducing binaries, in the long run, principles from reproducible builds can also support reproducibility of science.

5.4 Discussion and Recommendations

Reproducible builds are a notoriously hard problem. Few developers pay active attention to reproducible builds and those that do usually face a laborious, tedious, and incremental process [85]. With software being usually constructed using many reusable components, ensuring reproducible builds for components is an important building block for the community to move toward reproducible builds for applications and infrastructure.

Our results can be interpreted in two ways:

- On the negative side, given that components in many ecosystems are distributed as archives of source code, often with little build-time processing, it is shocking/frustrating how few components are packaged in a reproducible way. Almost no RubyGems, PyPI and Maven packages are reproducible.
- On the positive side, most reproducibility issues are rooted in the packaging tools of the ecosystem and can be addressed with small changes to the default settings of those tools. In contrast to usual discussions of reproducible builds for applications, only very few components in our study had reproducibility issues that cannot easily be addressed, even for components with native code extensions. This signals that reproducible builds for components are within easy reach.

Recommendations to practitioners After patching packaging tools, the remaining unreproducible builds in our study were caused by ad-hoc solutions applied by developers. I recommend that developers avoid extending specification files with custom scripts, even if the ecosystem permits such behavior. Relying on custom configurations can easily compromise reproducibility, as I have seen (Section 5.2.2).

Recommendations to maintainers of packaging tools I make three recommendations: (1) secure defaults, (2) build information, and (3) warning when unreproducible. First, in line with past work on usable security [87, 184, 113], I emphasize the importance of secure defaults – in this case, defaults that eliminate common reproducibility issues. Reaching every single developer is inefficient and hopeless, but with simple changes to the packaging infrastructures most components will be packaged reproducibly without any manual effort from developers. Second, packaging tools should automatically record information that enables build reproducibility. This includes the used compiler, the exact dependencies, and build options. Third, packaging tools should issue warnings if packaging steps rely on undeclared infrastructure (e.g., JavaScript transpilers installed on the local computer but

not declared as a dependency) and nondeterminism in builds (e.g., by explicitly running reprotest-like experiments during the build).

Recommendations to researchers Most of the findings of this study enable further research. Our aim was to provide a comprehensive study of reproducible builds in different ecosystems. While easily achievable, reproducible builds can be compromised by additional issues that require more examination. Further research may also consider developing methods that recreate build information for published packages as a stop-gap solution until package maintainers include this information by default. Recently, AROMA [126] proposed such an approach for Java packages in Maven.

5.5 Conclusion

Overall, *our study results make us optimistic* about the future of reproducible builds. For software components, reproducible builds are much closer than I expected, and a few small changes to infrastructure tools have a large lever to get the community to a future where most packages are fully reproducible. Barriers to ensuring that nearly all published components are exactly reproducible from the public source code are minimal. I can focus our attention more on other last-mile issues, such as linking published components on package managers to specific commits of their publicly available source code, ensuring the provenance of code changes to the public source code [174], and ensuring that packages are consumed in a reproducible manner when building applications (e.g., locking floating dependencies) [88].

Part III

Automation

Chapter 6

Automatic Security Assessment of GitHub Actions Workflows

Code repositories (also known as CRs) are a crucial part of every software development process. They are used to store, share, distribute, and version software and its dependencies. Indeed, code repositories are widely used in SSCs as they enable the storage of the core software and its distribution. In detail, the software's coding and building phase integrates several pieces of code (e.g., modules and third-party libraries) hosted in separate CRs. To this aim, the corresponding SSC will include a set of CRs contributing to the final software.

Also, in the last years, the push for automation procedures of Continuous Integration/Continuous Delivery (CI/CD) led the distributors of the most common platforms (e.g., GitHub, GitLab, Bitbucket) to introduce engines for the execution of *workflows*. A workflow is a sequence of actions aiming to automatize software's building, testing, and verification. In practice, they allow automatizing CI/CD processes directly into the repository without relying on external services. Workflows can be configured to run when manually triggered, at a scheduled time, or when a particular event on the repository occurs.

GitHub publicly released GitHub Actions (GHA) in 2019, and thanks to the provider's popularity, this service started to be widely used by developers. In a nutshell, GHA consists of an API and a dedicated engine that allow users to define and execute workflows on their repositories. GHA engines support the execution of workflows on GitHub dedicated machines or self-hosted ones.

The ability of workflows to manage and modify the content of CRs makes them an appealing target for attackers. For instance, several technical reports [136, 176] provided examples of attacks targeting GHA workflows to obtain control of GHA engines or inject malicious code into a repository. Through the SSC, the compromised CR is able to affect

the other nodes that rely on it, e.g., the final software importing the compromised code in its codebase.

In this Chapter, I provide the following contributions. First, I analyzed the GitHub security guidelines for hardening workflows, and I extracted a set of security constraints regarding confidential information, third-party workflows, permissions, and context variables. Then, I proposed a methodology to evaluate the security posture of GHA workflows, and I presented a prototype evaluation, called GitHub Action Security Tester (GHAST), based on the *Sunset* SSC security framework [35]. Finally, I conducted an in-the-wild experimental campaign on 50 GitHub - publicly available - repositories. The results allowed us to provide an overview of the security status of repositories. From this evaluation, I analyzed the workflows of 646 unique repositories involved in the SSCs of all 50 projects and identified 20 previously unknown security vulnerabilities and 24,885 security misconfiguration.

In the rest of the Chapter, I present our security assessment methodology to evaluate the security posture of workflows (Section 6.1). Moreover, I provide an implementation of the methodology (GHAST), and the techniques applied to rebuild the SSC and extract the required data in Section 6.2. Section 6.3 is dedicated to the analysis of results obtained applying GHAST in the wild against 50 open source projects. Section 7.4 discuss some related work. Finally, Section 7.5 draws some conclusions and discusses some future extensions of this work.

6.1 Security Assessment Methodology

I propose a novel methodology composed of two phases, namely *Workflow Collection* and *Workflow Security Evaluation* in order to evaluate the security of GHA in a software supply chain. The Workflow Collection phase is devoted to analyze the SSC and extract a model that includes all the involved CRs and - for each repository - the set of GHA workflows. The Workflow Security Evaluation phase performs a security analysis of each GHA workflow to assess five security categories extracted from the analysis of the official documentation [90] and the security hardening guidelines [93]. The result of the analysis is a report containing a list of security issues affecting the workflows used in the SSC, classified according to their exploitability.

Security Categories	**Security Constraints**	**Security Checks**
Confidential Data Disclosure	- Registering generated values as secrets. - Registering modified values as secrets.	(SC-1) Check the use of the `secrets` context outside environment. (SC-2) Check the use of `secrets` context for generating new data.
Command Injection	- Do not directly use github context in scripts.	(SC-3) Check the use of `github` subcontexts inside run steps.
Third-Party Workflows	- Audit the use of branch names, emails, and external inputs. - Audit information passed to TP actions. - Keep third-party workflows up-to-date.	(SC-4) Verify the version of reusable workflows. (SC-5) Verify the use of the pinning Commit Tag.
Workflow Permissions	- Specify permissions to avoid uncontrolled access.	(SC-6) Check if the workflows enforce the least privilege principle.
Triggering Events	- Audit the kind of events that trigger the workflow. - Audit the filters applied to triggering events.	(SC-7) Verify which is the exploitability score of the event.

Table 6.1 List of security categories, requirements, and checks of the Workflow Security Evaluation phase.

6.1.1 Workflow Collection

The first part of the methodology takes as input a folder containing the software under test (hereafter, SUT). The procedure parses the code and configuration files of the project to recursively identify software dependencies, code repositories, and distribution networks composing the Software Supply Chain of the SUT. From such a piece of knowledge, the methodology builds a direct graph structure where nodes include any code repository involved in the SSC, while edges represent the relationships among them.

Then, the procedure focuses on each CR to extract its GHA workflows (if any). More specifically, the methodology relies on GitHub API to search and retrieve the YAML files of the available workflows. The set of workflows is then linked to the corresponding CR node in the graph.

6.1.2 Workflow Security Evaluation

The Workflow Security Evaluation phase scans the set of GHA workflows to detect security vulnerabilities and misconfigurations that can affect code repositories in the SSC.

The detection logic of this phase is based on the evaluation of a set of *security categories* that are mapped in constraints - i.e., security requirements that workflows must comply with - and *security checks* - i.e., technical controls to assess the enforcement of constraints. The list of categories, constraints, and checks applicable to GHA workflows derive from a manual review of the GitHub guidelines for workflow hardening [93] and from the official GHA documentation [90]. Table 6.1 reports the results of our analysis.

In particular, I identified four categories from the analysis of GH guidelines concerning the security of GHA workflows, namely *Confidential Data Disclosure*, *Command Injection*, *Third-party Workflows*, and *Workflow Permissions*. Also, I extended those categories with *Triggering Events* as they represent the entry point for workflows.

```
( ISSUE_TITLE_INJECTION,
    "issues[opened]",
    "vuln_job",
    "",
    2,
    "echo \"ISSUE TITLE: ${{github.event.issue.title}}\"",
    1
)
```

Figure 6.1 Example of a COMMAND_INJECTION security issue detected in the workflow of Figure 2.4.

The evaluation phase parses each GHA workflow to *i*) execute the security checks detailed in Table 6.1 to detect security vulnerabilities and misconfigurations, and *ii)* evaluate the exploitability of the findings by identifying the events and preconditions that allow triggering the corresponding security issue.

The result of the evaluation is a set of tuples containing the detected security issue, some information related to the issue (i.e., the name of the job and the affected step, the position of the issue, and the print of the affected line), the event triggering the issue (i.e., the entry point), and an evaluation on its exploitability. If an issue belongs to a job triggered by multiple events, the evaluation will contain a separate tuple for each event to enable the filtering of specific results.

```
1      ( security_issue_type,
2        triggering_event,
3        job_name,
4        step_name,
5        step_position,
6        issue_line,
7        exploitability_score )
```

Listing 6.1 Structure of a security issue identified by the Workflow Security Evaluation phase.

Listing 6.1 provides the structure of a tuple, while Figure 6.1 shows an example of a tuple for the scenario depicted in Figure 2.4.

The rest of this section will detail the evaluation of the exploitability of events and the type of security issues supported by the methodology.

Execution of Security Checks.

The execution of the security checks listed in Table 6.1 enables the identification of a set of security issues targeting the workflow under test. The methodology labels an identified security issue into two different groups, i.e.:

Vulnerability. A flaw in the workflow that is directly exploitable. For example, a misuse of the `github.context` API enables attackers to execute command injection attacks.

Misconfiguration. A configuration error that makes the environment, the CR, or the SSC vulnerable. For example, the improper configuration of `tag` variable inside a workflow may lead to the import of an outdated third-party workflow.

In detail, the security issues belonging to the Confidential Data Disclosure and the Command Injection categories can be identified using pattern verification techniques.

Secrets can be used in workflows through the `secrets` context [98]. A particular secret is accessed declaring the context and then its name. Secrets are expected to be accessed inside of an environment. Indeed, when they are accessed from outside, secrets can be exfiltrated by an attacker (e.g., printed in a log or sent to a remote host) that has access to the workflow. Then, the security checks for Secrets consist of assessing if secrets are used in the environment part of the workflows (`SC-1`) and, if it is not the case, if they are manipulated in order to generate confidential data (`SC-2`). Indeed, `SC-2` allows assessing when a secret is involved in a computation and then exposed to potential exfiltration.

Similarly, the methodology can verify the presence of command injection attacks in workflows by evaluating the usage of `github` context [91] in run steps. In detail, the syntax `github.`$*$ is used to access variables contained in the GitHub Context. This context contains many subcontexts linked to different aspects of the workflows. For example, the `github.event` context contains all the information of the triggering event. Some parts of the `github` context can be manipulated externally by an actor. Hence, an attacker can inject a command into a vulnerable instruction through this context. For example, in the scenario of Figure 2.4, the attacker is able to exploit the run step through the issue title contained in `github.event.issue.title`. For this reason, the methodology evaluates the use of `github` subcontexts that can be exploited by an attacker to affect the workflow run (`SC-3`).

The command injection issues are further distinguished between *conditional* and *unconditional*. The first case consists of command injection attacks in a branch of a conditional statement, thereby requiring the attacker to trigger the appropriate branch to exploit the vulnerability. In the unconditional case, the vulnerable statement will be executed directly.

To evaluate the security checks belonging to the Third-party Workflows category, the methodology relies on the GitHub API to obtain the latest version of the specific TP workflow and compare it with the version requested by the workflow under test. If the used TP workflow is out-of-date (`SC-4`) or the workflow under test does not pin a specific commit hash (i.e., using the `<wf_name>@<commit_hash>` syntax) (`SC-5`), the methodology marks the issue as a security misconfiguration.

Finally, the methodology evaluates the workflow under test to check if the permissions declared in the workflow follow the principle of least privilege. Following the least privilege principle, the methodology checks if the required permission matches the capabilities required by the workflow (`SC-6`).

Exploitability Score

The only way for an attacker to execute a workflow and, thus, exploit a security issue is by triggering one or more events defined in the workflow. To this aim, the methodology evaluates the potential entry points by assessing the configuration of workflow events and the type of conditions that enables their activation (`SC-7`).

Events are influenced by filters [92] that can be applied to specialize the triggering actions. For example, an *Issues* event can have up to 16 activity types (e.g., opened, deleted, labeled, ...). To this aim, the use of filters directly affects the exploitability of a workflow.

The methodology classifies events using three security levels based on the complexity an attacker has to face to stimulate a particular event. I consider the complexity as the number and type of preparatory activities to get the required assets (e.g., privileges, knowledge) to trigger an event.

Restricted (1) The attacker needs to achieve a multi-stage attack to be able to trigger the specific workflow event. The attacker can succeed only with the help, whether intentionally or not, of one of the owners of the repositories. For example, an attacker cannot trigger a push event unless a repository owner grants him the maintainer status.

Supervised (2) The attacker must match particular conditions in order to trigger the event. For example, a pull request can be triggered, but the maintainer has to accept it and consequently start the workflow.

Unsupervised (3) The attacker does not need any granted permission by the repository owners. He can trigger the workflow at any time by means of external action. For example, an issue opening event can be triggered by any GitHub user with access to the repository.

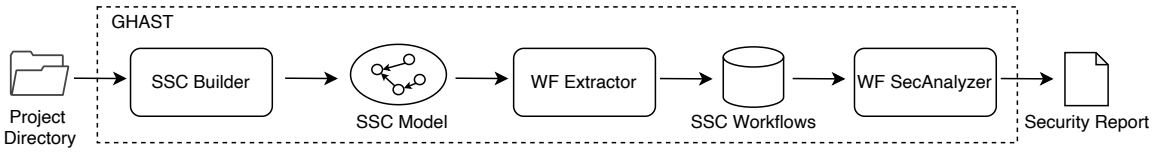


Figure 6.2 The GHAST architecture.

6.2 Prototype Implementation

I implemented the proposed methodology in a command-line tool - the GitHub Actions Security Tester (GHAST) - composed of three modules, as depicted in Figure 6.2. The source code of GHAST is publicly available[1].

The first module (*SSC Builder*) extracts the model of the SSC starting from a folder containing the source code of SUT. The second one (*WF Extractor*) extracts the list of code repositories from the SSC model. It extracts the set of GHA workflows from these collected repositories. The last module (*WF SecAnalyzer*) executes the workflow security evaluation described in Section 6.1.2 to produce the final security report.

For the *SSC builder*, I rely on a state-of-the-art research tool for the rebuilding and modeling of software supply chains [35]. In detail, the module requires the folder containing the SUT as input, then it recursively identifies the SSC assets, including code repositories. The output model is a direct graph structure stored in a Neo4J Graph database [147].

The *WF Extractor* queries the SSC model to extract the set of repositories that uses GHA workflows. Then, the module scans each repository using the GitHub API and downloads all the workflow YAML files contained in the corresponding `.github/workflows` folder. The result of this operation is the collection of workflows used in the SSC.

Finally, the *WF SecAnalyzer* evaluates each workflow by applying the security checks listed in Table 6.1. To do so, the module parses each YAML file to extract the following GHA elements:

- *triggering-events*, for the computation of the exploitability score (`SC-7`). These elements also contain event filters that are included in the score computation.
- *Runs*, for the evaluation of command injection and confidential data disclosure issues (`SC-1`, `SC-2`, `SC-3`).
- *TP-workflows*, for the evaluation of commit pinning and the workflow versions (`SC-4`, `SC-5`).

[1] https://github.com/Mobile-IoT-Security-Lab/GHAST

- *Permissions*, for the evaluation of permissions both at workflow level and at job level (`SC-6`).

```
"<work_test>":
{
  "events": ["issues",3],
  "issues": [
    ["MISCONF_PERM_GLOBAL", "<job_A>",...],
    ["OUTDATED_WF", "<job_A>",...],
    ["CI_ACTOR","<job_B>",...],
    ["OUTDATED_WF", "<job_A>",...],
    ["OUTDATED_WF", "<job_B>",...]
  ]
}
```

Listing 6.2 Extract of a sample security report produced by GHAST.

The output of the evaluation is a security report in JSON format that contains - for each repository (and workflow) of the SSC - the set of security issues identified by the tool. Listing 6.2 shows an extract of an output file. In the example, the tool identified five issues in the workflow `work_test`, associated with the event `issues`, i.e., a command injection exploiting the actor username, a misconfiguration of the permissions, and three outdated third-party workflows. The exploitability score of the event triggering those finding is 3, i.e., unsupervised.

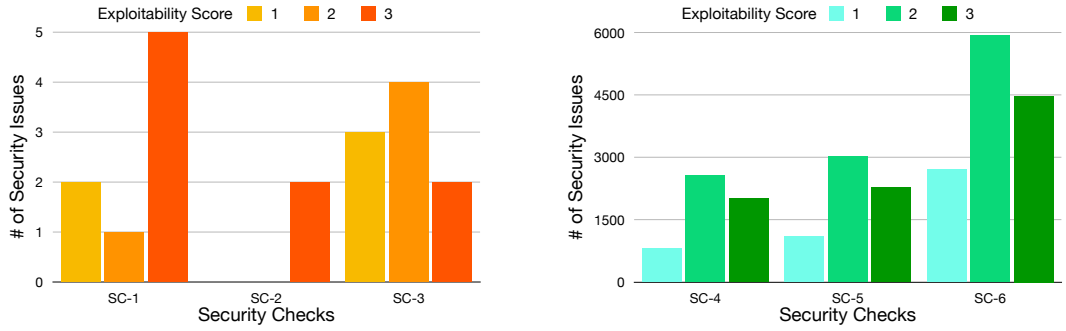
6.3 Experimental Evaluation

I conducted an experimental campaign to test the applicability and efficacy of GHAST in the wild on software repositories available on GitHub. In detail, I ran the tool against 50 public repositories randomly taken from the top 100 Python-based projects on GitHub in July 2022. The repositories are actively maintained and used by the community (e.g., the dataset has an average number of stargazers per repository above 20k). Table 6.2 details the dataset's characteristics regarding the number of stargazers, forks, contributors, open issues, and commits.

The tool extracts the SSC for each repository, identifies the CRs compatible with GHA workflows, and processes the security report. The experiments were hosted on a virtual machine running Ubuntu 20.04 with 8 processors and 32GB RAM.

	MIN	MAX	AVG
Stargazers	4,365	191,386	22,227
Forks	307	36,334	4,708
Contributors	3	30	25
Issues	1	1,707	232
Commits	40	52,887	4,729

Table 6.2 Number of stargazers, forks, contributions, issues, and commits (min, max, and avg values) of the dataset.



(a) Vulnerability Issues identified.

(b) Misconfiguration Issues identified.

Figure 6.3 Issues for different exploitability scores.

6.3.1 Experimental Results

GHAST was able to reconstruct the SSC of the 50 projects obtaining 646 unique CRs in approximately 14 hours. Then, the tool extracted 131,168 GHA workflows and executed the security checks.

The analysis identified 24,905 security issues, i.e., 20 security vulnerabilities and 24,885 misconfigurations. All the findings have been directly reported to the maintainers of the repositories.

Security Vulnerabilities. Figure 6.3a reports the distribution of security vulnerabilities identified using `SC-1`, `SC-2`, and `SC-3` according to their exploitability score.

The security checks on Secrets `SC-1` and `SC-2` reported 10 jobs that are not using the `secrets` context inside an environment. Such behavior enables jobs to expose those secrets, e.g., by printing in a log output or sending them to an external host. Also, it is worth noticing that most of these vulnerabilities (i.e., 7 out of 10) can be triggered using unsupervised events.

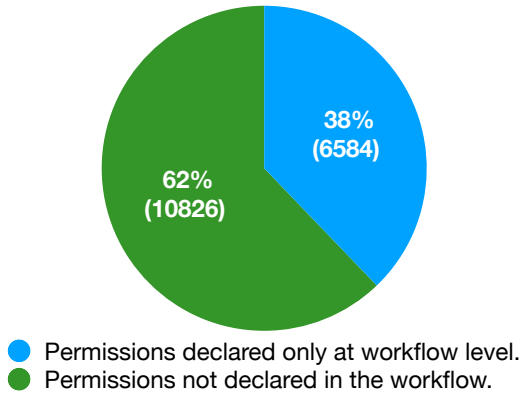


Figure 6.4 Distribution of Misconfigured Permissions (SC-6).

GHAST also reported 9 workflows vulnerable to command injection attacks (`SC-3`), of which two do not need any granted permission by the repository owner to be exploited (score equals to 3). Also, 3 out of 9 are *unconditional* command injections, i.e., that the affected step is not included in any conditional branch, thereby easing the exploitation process.

Misconfigurations. For `SC-4` and `SC-5`, I considered both checks passed when the workflow uses the commit hash of the latest version available of the third-party workflow. In particular, I found that this condition has never been verified on our sample set, meaning that even if a workflow uses the `latest` tag, it is still vulnerable to tag-reuse attacks. The experimental campaign allowed the identification of 5,384 steps that do not use the latest version of a third-party workflow. Also, the evaluation of `SC-5` resulted in 6,388 steps that do not pin a specific version of a third-party workflow.

Finally, `SC-6` reported several misconfigurations in workflow permissions definition. In detail, Figure 6.4 shows that 38% of workflows declared the permissions granted for their execution at the workflow level instead of for each job (as suggested by the security guidelines [93]). Furthermore, 62% of workflows do not declare any specific permission, thereby allowing the execution of the workflow with the default permissions assigned to the `GITHUB_TOKEN`.

6.3.2 Manual Validation of the Vulnerabilities

I manually revised the 20 identified vulnerabilities to verify the presence of false positives. To conduct this analysis, I downloaded the affected repositories locally and instantiated a local GHA Runner for the corresponding workflows to reproduce the exact conditions to

test the exploitability of the attack. The result of manual analysis confirmed that all the vulnerabilities identified by GHAST are true positives.

Listing 6.3 shows an anonymized GHA workflow containing a command injection issue triggered by a pull request event reported by GHAST. The workflow belongs to a repository with more than 10,000 stars, 500 forks, and 3,000 commits.

```
1  name: Pull Request Validation
2  on:
3    pull_request:
4      types: [opened, synchronize, reopened, edited]
5  jobs:
6    <anon-job-name>:
7      name: -------------
8      runs-on: ubuntu-latest
9      steps:
10     - name: <anon-step-name>
11       uses: actions/checkout@v2
12       with:
13         ref: ${{github.event.pull_request.head.sha}}
14         fetch-depth: 0
15     ...
16     - name: <anon-step-name>
17       run: |
18         cat << EOF | egrep -qsi '^disable-check:.*\<commit-count\>'
19         ${{github.event.pull_request.body}}
20         EOF
```

Listing 6.3 Excerpt (anonymized) of a workflow vulnerable to a command injection attack.

In such a workflow, it is possible to replicate a similar attack that affects the workflow in Figure 2.4 to initiate a reverse shell and access the Runner executing the workflow.

In detail, I crafted a payload to exploit the interpolation of the `github.event.pull_request.body` variable in the workflow, and I submitted it as a new pull request in the target repository, as shown in Figure 6.5. If a maintainer accepts the pull request, she will trigger the vulnerable workflow. When the GHA reaches the run step of Listing 6.3, the Runner executes rows 17-20 using bash.

In this case, the run step contains an `here documents redirection` [109] that allows command substitution [108]. Hence, the GHA Runner executes the malicious payload

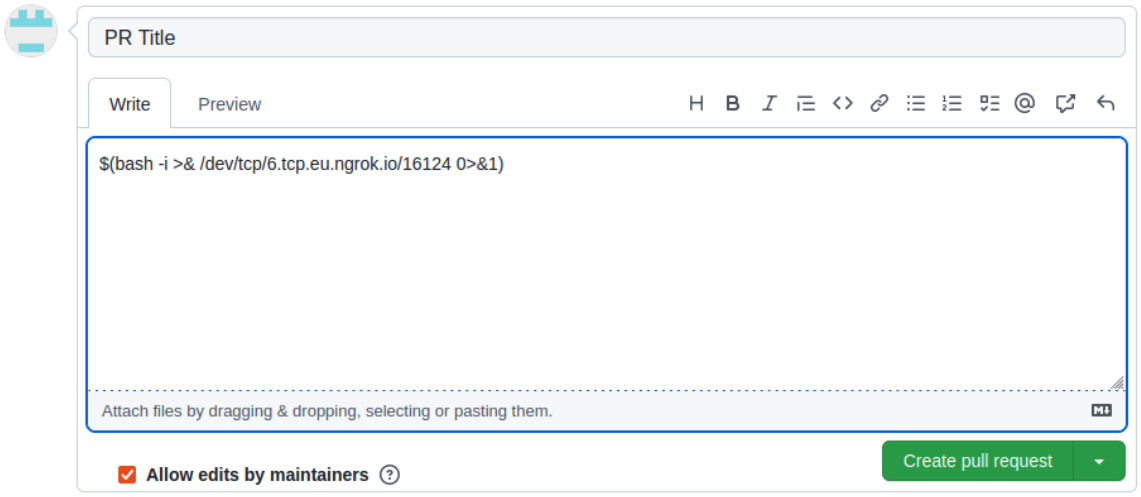


Figure 6.5 Creation of a malicious pull request on the target repository.

from the pull request and opens a TCP connection towards a remote server on port 16147. Figure 6.6 shows the listener active on local port 10000 mapped to the remote server, where the reverse shell is open. At this point, the attacker has complete access to the Runner instance, e.g., she can execute any command with Runner's privileges.

```
nc -lv 10000
bash: cannot set terminal process group (678): Inappropriate ioctl for device
bash: no job control in this shell
runner@fv-az445-461:~/work/wfSecurityAnalysis/wfSecurityAnalysis$ ls
ls
README.md
main.py
runner@fv-az445-461:~/work/wfSecurityAnalysis/wfSecurityAnalysis$ ls -s
ls -s
total 8
4 README.md
4 main.py
runner@fv-az445-461:~/work/wfSecurityAnalysis/wfSecurityAnalysis$ whoami
whoami
runner
runner@fv-az445-461:~/work/wfSecurityAnalysis/wfSecurityAnalysis$
```

Figure 6.6 Attacker terminal with the reverse shell to the affected Runner.

6.4 Related Work

To the best of our knowledge, the research work targeting the evaluation of GHA in the context of the software supply chain is limited. On the one hand, several organizations like ENISA [80], NIST [60], and OWASP [151] discussed the security issues in the Supply

Chain, giving a focus on SSCs and the impact of using insecure third-party software in the development pipeline. Also, several technical reports investigated attacks targeting SSCs of popular software like Solarwind [121] and Log4J [165].

On the other hand, the scientific community mainly investigated the problem of ensuring the integrity of the final software in SSCs that include different actors (e.g., code repository platforms, software library binaries, and distribution networks). Several works [132, 115] propose the use of reproducible builds to create an independently verifiable path from source to binary code. To cope with the integrity problem in the last part of the development process, Vu et al. in [205] show how the software can be compromised between the production of source code and the building process and how it is possible to mitigate this security problem. Considering the entire CI/CD pipeline, the in-toto framework [199] proposes holistic integrity enforcement of a software supply chain. It gathers cryptographically verifiable information about the chain to accomplish its objectives.

To our knowledge, only two open–source projects explicitly target the evaluation of code repositories: GitGat [175] and GitHub Workflow Auditor [198]. The first is a project released in June 2022 that uses the Open Policy Agent (OPA) [55] to evaluate security policies for GitHub's organization, repositories, and user accounts. GitHub Workflow Auditor is a command line tool released by Tinder in July 2022 for the security assessment of workflows. The tool scans a specific organization, user, or repository to detect potential security issues in secrets usage and external inputs. Unlike GHAST, GitHub Workflow Auditor can neither reconstruct the SSC of a project nor extract all the workflows associated with different CRs to detect security issues. Also, the tool does not provide any security evaluation of Third-party Workflows, Workflow Permissions, and Triggering Events. Also, it is worth noticing that both tools were released in Q2 2022, thereby suggesting a growing interest in the topic.

Koishybayev et al. [130] studied the security of GitHub CI workflows in parallel with our research. In detail, their work identified four security properties (permissions, privileges, code controls, and secrets) affecting workflows in GitHub CI and other VCS platforms. Also, they released a PoC tool called GWChecker to assist in analyzing GHA workflows. As for the other works on code repositories, their research focuses on single workflows, and neither takes into account nor models the dependencies among GHA workflows, thereby lacking an evaluation of the workflows on the entire software supply chain of the SUT. In addition, the authors do not consider events and their exploitability to evaluate the impact of vulnerabilities and security misconfigurations of GHA workflows.

Following the publication of GHAST, Muralee et al. [145] release ARGUS. The tool is a sophisticated version of GHAST using inter-representation of code to perform static analysis

on GitHub Actions. While is out-performs GHAST, it also confirm the results I obtained on GitHub Actions workflows in the experimental evaluation.

6.5 Limitations and Discussion

Limitations. The proposed methodology obtained promising results during the experimental evaluation and allowed us to assess the applicability and efficacy of GHAST. In addition, the manual validation of the identified vulnerabilities confirms the reliability of the approach.

Still, our solution suffers from some limitations. First, the security evaluation methodology has inherited limits. Although some security controls (e.g., SC-4 and SC-5) are based on the definition syntax of GHA workflows, having an intrinsically low rate of false positives. Other SCs (e.g., SC-1 and SC-2) are based on regular expressions and pattern matching techniques that cannot keep into account other information, like, e.g., the workflow execution context. As a consequence, this could, in principle, lead to potential false positives.

In our work, I tried to reduce the rate of false positives by thoroughly reviewing the GHA documentation and real-world GHA workflows. This approach allowed us to catch some potential corner cases and forge a heuristic to evaluate the SCs. However, it is still possible that GHAST does not identify deviant cases; therefore, I argue that an assessment in the wild may lead to false positives.

Moreover, the evaluation of the GHA workflows depends on correctly identifying all the software repositories involved in the SSC and the associated GHA workflows. To this aim, an error in the parsing process of the SUT can lead to not identifying a subset of CRs and the related workflows, thereby leading to potential false negatives. In this respect, GHAST is based on the Sunset Framework [35] and shares the same parsing limitations.

Vulnerability Disclosure and Security Implications. I disclosed all the identified vulnerabilities to the owners of the affected repositories via email. Each email contained a description of the vulnerability, the potential impact of its exploitation, and the GHAST report. At the time of writing, 6 out of 20 repository owners acknowledged our notifications within a period of 1 month after the disclosure.

As with any security assessment tool, attackers can use GHAST to discover vulnerable projects. When releasing a tool, there is a fundamental trade-off between helping repository owners versus facilitating the attackers.

Given the increased attention to software supply chain security and the existence of similar publicly available tools (e.g., [130] [175], and [198]), I feel that the benefits to

repository owners for publicly releasing GHAST outweigh the harms. Attackers have the resources and capabilities to replicate GHAST, whereas many repository owners do not. Also, GHAST does not explicitly report how to exploit security misconfigurations. Finally, publicly releasing GHAST will also encourage further research and would help repository maintainers (e.g., tools to patch vulnerable workflows automatically).

6.6 Conclusion

I investigated the security issues affecting GHA workflows to understand their security impact on the software supply chain. I produced an analysis of GHA aspects impacting the security of repositories. Then, I leveraged our analysis into a methodology for automatically assessing the presence of security issues in workflows.

I implemented the methodology in GHAST (GitHub Actions Security Tester). GHAST runs on a project source code, taking advantage of the *Sunset* SSC security framework to retrieve the SSC. Then, it extracts and analyzes the GitHub Actions workflows applying the security checks designed in the methodology.

Using GHAST against 50 open source projects produced relevant experimental results regarding security issues. I analyzed the results, providing an overview of the current security landscape of GHA workflows. I identified and manually revisioned the security vulnerabilities discovered through GHAST.

For future work, I plan to deepen the analysis of third-party workflow to *i)* better understand their involvement in the SSC, *ii)* extend GHAST with automatic security assessment of third-party workflow code.

Chapter 7

Privilege Life Cycle in Software Management Platform Automation Workflows

Modern software development is based on the DevOps development approach [76]. DevOps relies on a range of tools to improve both the development phase (Dev), e.g., code management, and the operational phase (Ops), e.g., deployment, monitoring, and logging [177, 143]. These tools help to ensure that software is delivered quickly, reliably, and with minimal risk of errors or downtime. Automation tools (e.g., Jenkins [124]) have been instrumental in sustaining the principles of DevOps over the years.

However, software management platforms recently introduced automation workflows directly on their repositories [128]. An *automation workflow* is a set of steps that automate the management of software deployments, updates, and other operations in a streamlined and consistent manner. Automation workflows are designed to execute a series of actions autonomously — i.e., a task — when the developer invokes or responds to an event — i.e., a triggering condition.

Automation workflows on software management platforms are a crucial component of the DevOps process, as they help to automate and streamline the software delivery pipeline. Indeed, the possibility of applying automation without relying on external services pushed the community to thrive on this feature [70].

Thus, workflows rapidly became an essential aspect of modern software development practices affecting development throughout the development process, from code writing to release.

Privileges are crucial for automation workflows, as they allow ensuring the security and integrity of the software development process. Automation workflows involve a range of tasks, such as building, testing, and deploying software, requiring access to sensitive resources. Privileges determine a user or entity's actions on a particular resource. An entity is a non-human actor that can interact with the software management platform, e.g., a cloud application. Automation workflows can expose these resources without appropriate privileges to unauthorized access or modification, leading to serious security and compliance issues, as discussed in [111, 214]. In particular, recent work on GitHub Workflows [36] highlighted the importance of detecting privilege misconfigurations in their security assessment. Also, the authors evaluated more than 100k workflows discovering that 62% of them do not follow the GitHub security guidelines regarding permissions [102], thereby increasing the chance of attacks against the repository resources, e.g., the code hosted in the repository and the build process.

The industry leaders for software management platforms are GitHub [100], GitLab [104], and BitBucket [22]. These software management platforms share many similarities. However, their privileges and permissions management mechanisms differ in some aspects. Over the year, the documentation for these mechanisms received updates and patches. It grew stratified and sparse, and obtaining the proper knowledge of managing privileges became challenging.

This Chapter aims to define *a rationalization of the privilege system used by automation workflows*. Defining the privilege life cycle in the automation workflows enabled us to categorize the fundamental steps involved in the configuration, authentication, and resolution of privileges during the life span of an automation workflow. I used the privilege life cycle to investigate and compare the different approaches to privilege management for automation workflows. Thanks to a clearer understanding of the privilege management involvement in automation workflows, I discuss the advantages and the potential pitfalls identified in the different approaches used on the software management platforms.

The Chapter is structured as follows. Section 7.1 describes the different stages of the privilege life cycle and how privileges and permissions are involved in automation workflows. In Section 7.2, I analyze the privilege life cycle handling of the three most widely used software management platforms. In Section 7.3, I discuss the security implications of privilege life cycle studied implementations. Section 7.4 presents related work regarding automation workflows in software management platforms. Finally, I conclude this work in Section 7.5.

7.1 Privilege Life Cycle in Automation Workflows

Software management platforms allow the creation, storage, and management of repositories. Software developers can store and manage a project's code and related assets in a repository. Repositories provide a structured way to organize and version control the code and assets, enabling collaboration between developers working on the same project. A repository can be created by a single user, namely the Owner, or by an Organization, i.e., a business-shared account. The repository owner allows other users to access the repository by assigning them specific privileges mapped as *roles*. According to the given role, other users can access privileged operations, e.g., writing on the repository or creating automation procedures called automation workflows.

An automation workflow is a procedure to automate tasks such as building, testing, and deploying software packages. Software management platforms enable developers to define workflows as a series of steps or tasks that will be executed on the repository. Because of its nature, an automation workflow can interact with contents hosted on the repository and elements external to the repository environment, e.g., container image repositories. All the operations affecting the repository are managed through the software management platform APIs. An automation workflow is usually described in one or more specification files written with human-readable data serialization languages (e.g., YAML [33]) or other programming languages like JavaScript and TypeScript.

Software management platforms offer various methods to configure privileges to automation workflow, e.g., with user roles, repository settings, or directly on automation workflow specification files. Privileges are expressed as permissions that can be granted to specific users or groups. Users' access to repository resources is regulated with permissions. Users with the necessary privileges can define permissions for other users and entities for automation workflow execution.

Permissions define both the capability for a user to trigger an automation workflow and which actions the workflow can execute. Indeed, a common approach is to run the workflow with the same privileges as the user that triggered the execution.

However, these privileges can be elevated or restricted depending on the other configuration methods available. For example, a user *A* with reading permission triggers the workflow *B*. However, a task in B requires the workflow to access the repository with *write* permission. The workflow will execute correctly if and only if the developer of B has included in the workflow specification file the *write* permission. If this is the case, *B* inherits the *read* permission from A and the *write* permission directly from its specification.

In the analysis, I modeled the involvement of privileges and permissions in the automation workflow life span with a life cycle composed of three different stages, i.e., *privilege configuration*, *automation workflow triggering*, and *privilege resolution*. Table 7.1 lists the stages and the operations happening in each stage. The rest of this section details the resulting life cycle.

Stages	**Operations**
(A) Privilege Configuration	*(A.1)* Roles configuration *(A.2)* Permissions definition
(B) Automation Workflow Triggering	*(B.1)* Trigger point stimulation *(B.2)* Authentication for triggering
(C) Privilege Resolution	*(C.1)* Permissions resolution

Table 7.1 Stages of Privilege Life Cycle in Automation Workflows.

7.1.1 Privilege Configuration

The first stage of the privilege life cycle deals with the configuration of privileges. An automation workflow executes successfully only if it has the set of permissions necessary to *(i)* invoke the API provided by the software management platform and *(ii)* access the repository resources.

Depending on the software management platform, permissions and privileges can be defined:

- *outside the automation workflow*. This approach relies on the concept of role. A role grants privileges associated with a set of permissions. Software management platforms allow dealing with roles at different levels of granularity, e.g., organization, groups of repositories, and single repositories.

- *inside the automation workflow*. In this case, developers can directly define permissions in the specification file that apply to the workflow or event limited to specific parts (e.g., at a job or step level).

In both ways, privileges granularity is an essential aspect in defining permissions. It depends on the specificity of resources; the more a resource is specific to a scope, the more the permission can be fine-grained.

During this stage, a developer should evaluate the expected automation workflow goals to understand which resources are needed by the workflow. A developer can obtain the required

knowledge through the software management platform documentation. In detail, it needs to understand how the software management platform *(i)* manages privilege roles, *(ii)* handles permissions and resources, and *(iii)* provides methods to modify privileges of workflows.

7.1.2 Automation Workflow Triggering

The automation workflow triggering stage regards identifying the starting conditions of automation workflows. The starting conditions can be divided into *event-driven triggers* and *manual triggers*.

Event-driven triggers collect all the events affecting the repository and its configuration. Those events include, for example, push and pull requests, issues opening, and timed events. In this case, the workflow activation depends on the privileges and permissions required for a user or entity to fire one of its triggering events successfully. For example, an automation workflow triggered by a push event needs a user or entity with enough permissions to perform the push on the repository.

Manual triggers allow users and entities to start an automation workflow on-demand. In this case, the successful start of the automation workflow depends on the privileges of the user or entity to access the workflow. Software management platforms use tokens to authenticate users and entities on the repositories and to determine their privileges. The most common tokens are *user tokens* and *access tokens*.

User tokens establish the user's identity during the automation workflow execution and match the permissions granted to the user or entity. Once authenticated, the automation workflows inherit the permissions granted to the user.

Access tokens, on the other hand, are specific to individual repositories and are single-purpose. They grant the permissions that were assigned at the time of their creation.

The permissions contained in the tokens are inherited by the workflow unless overwritten by the permissions defined directly in the workflow specification file.

In this stage, the developer should check *(i)* the starting conditions the software management platform supports, *(ii)* the available means of authentications, and *(iii)* how the workflow inherits or overwrites the permissions of the token.

7.1.3 Privilege Resolution

The pipeline's user or entity is authenticated and authorized to start the workflow during the triggering stage. The running automation workflow then interacts with the platform to access the resources required to execute the workflow steps using the available APIs.

GitLab		GitHub		BitBucket	
Role	**Description**	**Role**	**Description**	**Role**	**Description**
Owner	The Owner of a GitLab project or group has full access to all features and settings, including the ability to delete the project or group.	Owner	The Owner of a GitHub repository has full access to all features and settings, including the ability to transfer or delete the repository.	Admin	Users with this permission have full control over the repository, including the ability to create and delete branches, tags, and pull requests, and to modify repository settings.
Maintainer	Maintainers have similar access to Owners, but they cannot delete the project or group.	Collaborator	Collaborators have similar access to Owners, but they cannot transfer or delete the repository. This role is available only for organization level.	Create repository	Users with this permission can create new repositories within a project and become the Owner of those repositories. This role is available only for project level.
Developer	Developers have read and write access to the code, but they cannot modify project settings or invite others to the project.	Write	Users with write access can make changes to the repository, such as pushing commits and creating branches, but they cannot modify the repository's settings or add collaborators.	Write	Users with this permission can make changes to the repository, including creating and deleting branches and tags, and creating and merging pull requests.
		Triage	Users with triage access have the ability to manage issues and pull requests, such as assigning labels and milestones, but they cannot make changes to the repository's code or settings.		
Reporter	Reporters have read-only access to the project, and can view issues, merge requests, and code, but cannot make any changes.				
Guest	Guests have limited read-only access to the project, and can only view issues and merge requests that have been explicitly shared with them.	Read	Users with read access can view the repository's contents, but they cannot make any changes.	Read	Users with this permission can view the repository and its contents, including code, issues, and pull requests, but cannot make any changes.
Anonymous	Anonymous users have read-only access to the project, but do not need to be logged in to view it. This role is typically used for public projects that do not require authentication.				
				None	Users with this permission have no access to the repository.

Table 7.2 Comparison of privilege roles on software management platforms.

The platform evaluates the permissions of the authenticated user or entity to determine whether they have the necessary rights to access the requested resources. The resolution process is based on the policies adopted by the platform, which define the rules and criteria for granting or denying access to resources.

The policies implemented by the platform dictate the precedence order of permission levels. The evaluation process may involve multiple levels of permissions. The platform evaluates each permission level in the order defined by the policies to determine the level of access granted to the user or entity that triggered the pipeline.

In this stage, it is necessary to understand how the software management platform applies privilege resolution, considering precedence orders and granting access to resources.

7.2 Software Management Platforms and their approaches to privilege-granting

Software management platforms provide methods for managing the privilege life cycle in automation workflows. This section analyzes the privilege life cycle management for GitHub, Gitlab, and BitBucket software management platforms.

7.2.1 GitHub

GitHub provides *GitHub Actions* [49, 101] as automation technology on its platform. GitHub Actions was released in 2019 and gained attention because of its workflow definition and usage flexibility. A workflow is defined by a YAML file containing a set of operations called jobs. Each job will execute a sequence of tasks called steps. GitHub Actions also support the use and integration of external workflows, called reusable workflows, which allow developers to integrate third-party automation workflows to execute already defined jobs. Reusable workflows can be defined using YAML, JavaScript/TypeScript, or Docker.

(A.1) Roles configuration. GitHub distinguishes between personal repositories and organization-managed repositories. Personal repositories have only two roles: *(i)* the owner and *(ii)* collaborators. Organization-managed repositories, instead, have multiple roles that are (from the least privileged to the most privileged): *(i)* read, *(ii)* triage, *(iii)* write, *(iv)* maintain, and *(v)* admin (see Table 7.2). Additional roles can be created with the Enterprise plan offered by GitHub, allowing for a more granular permissions control for roles. Organizations can set base permissions for all organization members. Base permissions are automatically granted to users when added to the organization.

(A.2) Permissions definition. GitHub organizes repositories in scopes managing specific resources (e.g., file resources are in the *contents* scope), as reported in Table 7.3.

GitHub allows configuring workflow permissions over GitHub resource scopes according to two privilege levels, i.e., *restrictive* and *permissive*.

The former grants read permissions in the repository for the *contents*, and *packages* scopes only to the automation workflow. The permissive privilege level, instead, grants read and write permissions over all repository scopes.

More granular permissions can be declared in the workflow specification file. Scopes and corresponding permission values are declared in the workflow specification file through the `permissions` keyword.

Scope	**Description**
actions	Grants permission to access the Actions API, including the ability to create and manage secrets. The read permission allows reading Actions metadata and logs, while the write permission allows starting and canceling workflows and approving and rejecting workflow runs.
checks	Grants permission to read and write check runs and check suites for a repository.
contents	Grants permission to read and write repository contents, including files and directories.
deployments	Grants permission to read and write repository deployments.
id-token	Grants permission to read and write ID tokens for a repository, which can be used to authenticate with GitHub APIs.
issues	Grants permission to read and write issues for a repository.
discussions	Grants permission to read and write discussions for a repository.
packages	Grants permission to read and write packages within a repository or organization.
pages	Grants permission to read and write GitHub Pages for a repository.
pull-requests	Grants permission to read and write pull requests for a repository.
repository-projects	Grants permission to read and write repository projects for a repository.
security-events	Grants permission to read and write security events for a repository.
statuses	Grants permission to read and write commit statuses for a repository.

Table 7.3 Descriptions of available scopes for permissions in GitHub Actions.

Possible permission values for scopes are *read*, *write*, and *none*. The *read* value provides access to the resource. The *write* value enables the user to modify the resource. The *none* value disables all activities on the resource.

The permissions can be specified at workflow and job levels inside the workflow specification file, reducing the permission scope over the resource even more.

(B.1) Triggering points stimulation. GitHub Actions workflows support different types of triggers. Triggering events can be defined for an entire workflow inside the workflow specification file. In detail, there are 36 possible events [103] that can trigger a workflow. Filtering events depending on operations (e.g., creation of issues event) and branches (e.g., push on specific branch) is also possible. Manual triggering is enabled through a specific event, i.e., `workflow_dispatch`. Automation workflows triggered by the `workflow_dispatch` event starts when a user or an entity requests a specific API endpoint associated with the workflow. Workflow triggering can also be scheduled using the `crontab` syntax [86].

(B.2) Authentication for triggering. The user or entity needs to authenticate on GitHub to trigger an automation workflow to allow the platform to evaluate the possession of the required privileges to fire a starting event. This policy also applies to manual triggers because they are mapped as a specific event by the GitHub platform (i.e., `workflow_dispatch`).

The authentication to events is managed through two types of access tokens, i.e., the *user token* and the *fine-grained access token*. The former is associated with the user's role, and thus, it inherits the privileges related to her. The latter, instead, are tokens containing permissions defined during their creation.

(C.1) Permissions resolution. Once the workflow has been triggered, GitHub computes the privileges involved in the run. This activity considers all the permissions specified in the *privilege configuration* stage.

In detail, the workflow permissions are initially set to the base permissions declared in the repository settings (i.e., *restrictive* or *permissive* mode). Base permissions can be elevated or restricted by permissions defined in the automation workflow specification file. Hence, GitHub grants the finest-grained permissions defined in the *privilege configuration* stage. For example, a workflow needs to push changes to a branch in your repository. Still, the base permissions are set to *restrictive* mode, allowing the workflow only to read the repository content. In this case, it is necessary to elevate the permissions granted to the automation workflow during its execution. To do so, it is possible to declare such higher permissions directly in the automation workflow specification file.

7.2.2 GitLab

GitLab automation technology takes the name of *CI/CD pipelines* [106]. A CI/CD pipeline workflow comprises *stages* and *jobs*. Stages usually represent development pipeline phases (e.g., build, production, test). Jobs represent the action to be taken on the repository. Stages are executed sequentially, and their order is defined at the beginning of the workflow. When all jobs in the stage succeed, the workflow moves to the next stage.

A job is composed of sequential bash commands. A runner executes these commands using the repository root as the working directory.

GitLab can be installed on-premises on personal servers. For this reason, permissions can be granted at three levels of granularity: instance-wide (i.e., the GitLab instance on the server), group, or project level.

(A.1) Roles configuration. GitLab applies a role-based authorization policy. It defines five roles, from the least privileged to the full privileged: *(i)* Guest, *(ii)* Reporter, *(iii)* Developer, *(iv)* Maintainer, *(v)* Owner (see Table 7.2). Moreover, GitLab has a special role, the *Administrator*. This is the sole role able to deal with the management of the GitLab platform instance.

(A.2) Permissions definition. Permissions can be defined through the GitLab settings. Roles enabled for permission granting depend on the granularity level. Instance-wide permissions and privileges can be managed only by the Owner role. The Owner and Maintainer roles deal with group permissions, while the Owner, Maintainer, and Developer roles can manage project permissions.

Project owners can configure more granular permissions for automation workflow execution using *Protected Branches* and *Protected Tags*. These settings allow project owners to restrict automation workflow execution for certain branches or tags to specific users or groups with defined permission levels. Moreover, it is possible to modify project-level permissions scope for automation workflows. In particular, the *public pipelines* and *pipeline visibility* options change the visibility of workflows for low-privilege roles.

GitLab does not allow managing permissions in the workflow definition file, as GitHub does. Then, automation workflows only rely on the user role granted during this phase.

(B.1) Triggering point stimulation. GitLab automation workflows are triggered by push and merge request[1] events by default. Also, it is possible to configure additional triggering events for CI/CD pipelines through dedicated webhooks [107]. Thanks to such a mechanism, GitLab provides a triggering-by-event approach similar to the one offered by GitHub. Web-

[1] equivalent to pull request in GitHub

hooks can also be involved in the manual triggering of automation workflows. The webhook URL can be embedded in external services, integrating the automation workflow. Moreover, GitLab API provides specific endpoints for automation workflow triggering.

(B.2) Authentication for triggering. Users or entities triggering a workflow with an API call have to include an access token. The token is evaluated to authenticate the user or entity, and the workflow inherits permissions associated with the token during execution.

GitLab enables automation workflow triggering through *trigger tokens*. These are single-purpose tokens that can be used to trigger a workflow with specific permissions.

Using trigger tokens, users can initiate workflow execution without requiring specific roles or permissions. This feature helps integrate automation workflow with external services or scripts. It allows these services to trigger the workflow without requiring full access to the GitLab project.

(C.1) Permissions resolution. GitLab evaluates privileges based on the membership relation [105]. Hence, the memberships to specific groups and projects play a crucial role in defining which permissions the automation workflow has. The precedence of permissions can be summarized in the following four points:

- Access level precedence. If a user is granted access directly at a higher access level than her group membership, then the higher access level takes precedence.
- Higher access levels precedence. If a user has been granted multiple access levels explicitly, then the highest access level takes precedence.
- Inherited access levels precedence (over no access). If a user has access to a project or group through inheritance, such as being a group member with access, then that access level takes precedence over having no access.
- Narrower access levels precedence (over broader access levels): if a user has been granted access at different levels of granularity, such as access to a specific project versus access to all projects in a group, then the more specific access level takes precedence.

The permissions evaluation policy guarantees users the most permissive access level possible, considering their roles.

7.2.3 BitBucket

BitBucket Pipelines is a continuous integration and deployment service built into the BitBucket platform. It allows developers to automate their code's building, testing, and deploy-

ment. Pipelines use YAML configuration files to define workflows and support a range of programming languages and tools. It integrates with other Atlassian [27] products such as Jira [29] and can also be extended through third-party plugins [28]. Pipelines offer teams a flexible and scalable solution to streamline their development process and improve their productivity.

As for GitLab, Users can interact with three levels, so privileges are granted on three levels of granularity: global, project, and repository. The global level concerns the BitBucket platform instance, while projects are groups of repositories.

(A.1) Roles configuration. BitBucket provides roles at the global level, the project level, and the repository level. Global-level privileges deal with management activities on the BitBucket instance, like creating projects and managing users and groups of users. These roles are, from the least privileged to the full privileged: *(i)* BitBucket User, *(ii)* Project Creator, *(iii)* Admin, and *(iv)* System Admin.

Concerning the project and repository levels, BitBucket allows the management of privileges directly associated with permissions. In detail, the permissions that can be associated with a user are, from the least privileged to the full privileged: *(i)* Read, *(ii)* write, and *(iii)* Admin (see Table 7.2).

(A.2) Permissions definition. Permissions can be defined through the BitBucket settings. This activity is restricted to the Admin role. An Admin user carries on the permissions granting activity for the scope of its role (i.e., a repository Admin manages permissions only for the repository, while a project Admin manages permissions for all the repositories in the project).

More granular permissions can be defined through the use of these features: *Deployment permission options* [26], *deployment variables* [24], and *branch permissions* [23].

The *Deployment permission options* feature provides two options. With *Admin restrictions*, repository owners can set privileges to trigger a deployment pipeline only for trusted users. With this feature, users need high-privilege roles to trigger critical pipelines that could accidentally or intentionally deploy malicious or buggy code into production. *Branch restrictions* enable controlling deployment to critical environments, like production. Repository owners specify the branches allowed to deploy to production. This feature prevents branches requiring fewer privileges to interact with them from deploying their content in critical environments.

Securing *deployment variables* prevents critical data from being stolen from execution logs during pipeline execution. Apart from the execution logs, the only place where deploy-

ment variables are available for displaying is in the repository settings. BitBucket imposes users to have at least the Admin role to access this repository section.

Using *branch permissions*, repository owners can set pipeline restrictions on specific branches. This feature prevents changes to the branch content and itself (e.g., deleting the branch). The permissions can be set on the branch at different granularity. Permissions can be set on branch types (e.g., development and production branches) or branch patterns (i.e., branches identified by regular expressions on their name).

Bitbucket Pipelines does not currently include access control as a configurable option in the automation workflow specification file.

(B.1) Triggering point stimulation. BitBucket Pipelines defines start conditions for automation workflows [25]. In detail, a pipeline can be triggered: *(i)* when a commit is pushed to any branch (default), *(ii)* when a commit is pushed to specific branches, *(iii)* when a pull request is created or updated and targeting specific branches, *(iv)* when a tag is created, *(v)* or manually by the user. *(i)* to *(iv)* are triggers by event.

BitBucket Pipelines does not allow the definition of other events for the automation workflows triggering. Automation workflows can be manually triggered when they present the `custom` property in their body. This property is comparable to the `workflow_dispatch` event on GitHub. Then, requesting a specific BitBucket API endpoint, the workflow is triggered.

(B.2) Authentication for triggering. BitBucket relies on user access tokens for authentication. The automation workflow runs with the permissions granted to the user and hence to the token. Because of the triggering events and the direct triggering policies, a user needs at least the permissions to write on the repository to initiate an automation workflow.

(C.1) Permissions resolution. In BitBucket Pipelines, permissions for automation workflow execution are determined by the permissions assigned to users and groups within the repository. If there is a conflict or overlap in permissions between different users or groups, BitBucket will prioritize the access level with the higher permission.

For example, suppose a user is a member of multiple user groups within the repository, and each group has different access levels. In that case, BitBucket will use the group's access level with the highest permission level to determine the user's access level during pipeline execution.

7.3 Discussion

Attacks targeting code repositories exploiting automation workflows are a real-world problem, as the scientific and industrial community pointed out e.g., in [36, 129, 186]. Although privilege management strategies cannot prevent these attacks as a whole, the role of permissions is mandatory for attack mitigation and prevention.

The introduction of role-based authentication and permissions by software management platforms aims to reduce the probability of triggering and exploiting vulnerabilities in automation workflows. Although, the task of granting privileges and roles to users is entirely transferred to the repository's owners and the workflows' developers. This is a critical task, and the time and competencies required to deal with repositories with many contributors should be more manageable.

All software management platforms support the specification of fine-grained privileges to mitigate privilege misuse. This allows developers to define the smallest number of resources involved in the privilege and the permissions to be granted.

Our analysis shows that the platform offering the finest granularity of permissions is GitHub. In particular, GitHub allows developers to define permission directly in the automation workflow specification file. This approach has two main implications. First, it breaks the separation between privileges management and automation workflow definition, shifting responsibilities to the workflow creator. Then, it provides a public mapping between resources and permission, enhancing the visibility of resources and permissions involved in the automation workflow. The sole use of roles for privilege management blurs the understanding of permissions involved in automation workflows.

Relying only on roles, a user role has to be elevated to guarantee the expected results from an automation workflow. This approach overcomes potential malfunction in the automation workflow. However, because of the coarse granularity of roles, the elevation of the role grants access to more resources than the ones necessary to execute the automation workflow. Consequently, this overprivileged user can access a larger attack surface. GitLab and BitBucket apply this approach to their automation workflows.

Software management platforms implement repositories groups for privileges, roles, and users. This feature enables the transferability of user privileges from higher levels, i.e., the group, to lower levels, i.e., the single repository. This process adds a layer of complexity to the privilege resolution phase. Thus, additional awareness is required during the privilege configuration phase to avoid unexpected overprivileged users. Complex membership mechanisms, such as the one implemented by GitLab [105], make tracking the inheritance of

privileges among repositories, groups, and projects difficult. This lack of visibility can make detecting and responding to security incidents challenging.

Moreover, as visible in Table 7.2, I notice that the distribution of roles on software management platforms is not uniform. For example, BitBucket tends to have more highly privileged roles; GitHub, outside of organizations, has just two highly privileged roles. The lack of low and intermediate roles can lead to more highly privileged users causing security concerns.

Overall, from this study, I argue the need for a security-first approach to privilege management on software management platforms.

This lack is highlighted by the missing strategies to identify potential flaws from misconfigured privileges and permissions. Thus, all three software management platform documentation recommend achieving the principle of least privilege for automation workflow execution. However, no mechanism exists to help or force users to obtain such a principle. In particular, the privilege configuration and the privilege resolution phases could integrate procedures to verify and even enforce the principle of least privilege or at least control over permissions.

7.4 Related Work

Software management platforms and automation workflows gathered attention in the last years. In particular, GitHub with its GitHub Actions and GitLab with the CI/CD Pipelines. The advantage of automation workflows concerning their use for pull request management has been pointed out in [203, 208]. Works as [128, 50, 70, 201] studied the adoption, involvement, and evolution of automation workflows in software development. Automation workflows have also been analyzed referring to specific research fields [127, 44]. The study in [110] reports statistics on how the usage decrease of CI services, like Travis CI [123], coincides with the rise of GitHub Actions that, in eighteen months, became the dominant automation technology.

The variety of automation workflow applications and their growing adoption arose interest in the security community. In particular, the authors in [36] provided a methodology for the security assessment of GitHub Actions and conducted an in-the-wild assessment of GitHub Action workflows to demonstrate the presence of several security issues in public GitHub repositories. In [129], authors analyzed the security of GitHub Actions and other CI/CD automation pipelines by conducting an empirical assessment of automation workflows. Both

works unveil the presence of potential misconfigurations in the privileges of automation workflows.

However, to the best of our knowledge, our work represents the first rationalization of privilege handling for automation workflows on the three major software management platforms.

7.5 Conclusion

In this work, I defined the privilege life cycle in automation workflows. I investigate how software management platforms implement the steps of the life cycle. This analysis highlighted how the methods used by software management platforms are similar under specific aspects. For example, platforms are coherent in triggering automation workflows, relying on access tokens and APIs. Although significant differences have been identified. For example, the involvement of privileged roles in the automation workflows. From a security point of view, improvements are necessary to guarantee a stronger security position. In particular, the complexity and heterogeneity of privilege management mechanisms and the lack of proper documentation support increase the probability of security violations related to misconfiguration and over privilege.

Security should be more central during the privilege life cycle in automation workflows. In particular, the *privilege configuration* stage of the privilege life cycle should contain compliance verification and enforcement methods. The study discussed in this Chapter aims to encourage the discussion on privilege management on software management platforms, focusing on automation workflows. The significant advantages of automation workflows on software management platforms risk being dangerous without proper privilege management methodologies.

Chapter 8

The Road Ahead

This Thesis proposes many advancements aiming to provide a more secure software supply chain without adding burden on developers. The software supply chain security topic is now vamping everywhere in the world, and it requires researchers to keep up in the years to come. This last Chapter discusses some possible next steps following the same separation of Thesis contribution: transparency, automation, and software trust.

8.1 Transparency

The last empirical studies highlighted the impact of software composition analysis in software supply chain security [142]. Part I gives a glimpse into the complexity of an SSC, discussing the many entry points available to attackers. Developing a framework dealing with this complexity is an ineffective effort. Research is focusing on smaller challenges to address the bigger problem. SBOM emerged as a tool for enabling transparency. The simplicity of the concept, a list of the ingredients for software, facilitate its adoption. However, producing a complete and correct SBOM is not an easy task. As shown in this Thesis there are many obstacles on the way to a fully accurate SBOM. Among these: are the heterogeneity of approaches adopted by software ecosystems in dependency management and the complete reliance on generation tools in metadata files.

An extension of this Thesis could be extending research to language-agnostic solutions using state-of-the-art technology as a translation layer. Using code analysis tools, such as Joern [6], it should be possible to focus the composition analysis on the structural elements of the code. Previous works show that bare static analysis cannot create complete SBOMs [211, 163, 31]. Investigating and improving SBOM generation tools with dynamic approaches is a

key step toward better SBOMs. This step should be taken considering also performances to push developers for adoption.

8.2 Software Trust

Reproducible builds are achievable goals for most software ecosystems, as shown in this Thesis. Causes of non-determinism are often inserted through configurations [167], for this reason, tools must be carefully designed to handle these configurations to provide reproducibility. However educating developers in making builds reproducible is an important goal, security must be thought of as a by-design property. Pushing in this direction, research should provide tools that enable reproducibility without requiring specific configuration to developers. OpenSSF conducted a survey on reproducible builds feasibility, highlighting the ecosystems currently dealing with provenance information. This information would be used to obtain build dependencies and then enable reproducibility. This Thesis shows that once the build dependencies are assessed, most software ecosystems have reproducible builds. The gap should be filled by distributing buildinfo together with software avoiding unknown build dependencies. At that point, everyone can build source code with the same specification of the distributed artifact, both software and build dependencies, and test the integrity of the release.

8.3 Automation

Recent studies on GitHub Actions adoption and usage show how they are key components in the supply chain of many software projects [70, 128, 71, 47, 150]. Extending this Thesis there are three main points of study: 1) Docker-based GitHub Actions, 2) other platforms than GitHub, and 3) dynamic approaches.

Docker-based Actions use custom Docker images as running platform. It is difficult to design a static analysis methodology since there is no a priori knowledge of the image content. That is, the image can run every operation that can be run on a traditional machine.

While GitHub is the most used version control system, others have different technologies implementing different policies. This Thesis provided a study on the permissions life cycle in GitLab and BitBucket, showing how differently access control policies are implemented depending on the platform.

Static approaches proved to be functional, however, there is a need for a more sophisticated approach to deploy efficient solutions. Dynamic analysis would largely improve

the security toolkit. On one hand, a dynamic approach allows for abstracting from the implementation of the automation technology. On the other hand, more detailed information on the security posture of complex CI/CD pipelines involving tens of elements. Dynamic analysis presents many challenges. GitHub Actions should be sandboxed to increase the testing coverage without breaking external resources. Thus, an Action usually interacts with the associated repositories and other third-party elements. Testing the Action in a local environment would not prevent it from communicating with these resources, causing potential harness. Developing an isolated sandbox would enable testing of Actions covering most use cases without the risk of interacting with real assets.

8.4 Conclusion

This thesis has advanced the understanding and security of modern software supply chains by addressing three critical pillars: transparency, trust, and automation. The rapidly increasing complexity of software ecosystems necessitates innovative tools and methodologies to identify vulnerabilities, enhance the integrity of software components, and automate secure development practices.

Key contributions include the development of SUNSET, a framework that systematically models the software supply chain, assesses risks, and generates actionable security insights. By leveraging SBOMs and introducing PIP-SBOM, this work demonstrated improved accuracy in dependency security assessments, particularly for Python ecosystems. The empirical investigation into reproducible builds across major packaging ecosystems addressed challenges in trust and proposed actionable solutions to improve software integrity. Furthermore, the GHAST tool and an analysis of permission lifecycles in CI/CD workflows highlighted significant vulnerabilities in automation tools like GitHub Actions, offering practical recommendations for secure configurations.

Together, these contributions form a cohesive strategy for mitigating the threats faced by interconnected software supply chains. Future work will continue to explore emerging challenges in software supply chain security, such as dynamic threat modeling, the integration of AI-driven predictive analyses, and the standardization of secure automation practices. The foundational principles and methodologies presented in this thesis aim to inspire further innovation and collaboration in the pursuit of a transparent, trusted, and secure software ecosystem.

References

[1] Intrusion detection. https://owasp.org/www-community/controls/Intrusion_Detection. Accessed: 2024-9-5.

[2] bomber: Scans software bill of materials (SBOMs) for security vulnerabilities. https://github.com/devops-kung-fu/bomber.

[3] Why chainguard uses grype as its first line of defense for CVEs. https://www.chainguard.dev/unchained/why-chainguard-uses-grype-as-its-first-line-of-defense-for-cves. Accessed: 2024-9-5.

[4] Grype. https://www.cisa.gov/resources-tools/services/grype. Accessed: 2024-9-5.

[5] False negative. https://www.contrastsecurity.com/glossary/false-negative. Accessed: 2024-9-5.

[6] Joern - the bug hunter's workbench. https://joern.io/. Accessed: 2024-9-23.

[7] jqlang/jq: Command-line JSON processor. https://github.com/jqlang/jq. Accessed: 2024-9-5.

[8] kubeclarity: KubeClarity is a tool for detection and management of software bill of materials (SBOM) and vulnerabilities of container images and filesystems. https://github.com/openclarity/kubeclarity.

[9] Versioning - python packaging user guide. https://packaging.python.org/en/latest/discussions/versioning/. Accessed: 2024-9-3.

[10] resolvelib. https://pypi.org/project/resolvelib/. Accessed: 2024-9-5.

[11] Secure software supply chain center. URL https://s3c2.org/.

[12] sbom-scorecard: Generate a score for your sbom to understand if it will actually be useful. https://github.com/eBay/sbom-scorecard.

[13] sast-scan: Scan is a free & open source DevSecOps tool for performing static analysis based security testing of your applications and its dependencies. CI and git friendly. https://github.com/ShiftLeftSecurity/sast-scan.

[14] Using grype to scan container images for vulnerabilities. https://edu.chainguard.dev/chainguard/chainguard-images/working-with-images/scanners/grype-tutorial/, Jan. 1. Accessed: 2024-9-5.

[15] False positives and false negatives in information security. https://www.guardrails.io/blog/false-positives-and-false-negatives-in-information-security/, Aug. 2022. Accessed: 2024-9-5.

[16] Macaron: A Logic-based Framework for Software Supply Chain Security Assurance. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 29–37, Copenhagen Denmark, Nov. 2023. ACM. ISBN 9798400702631. doi: 10.1145/3605770.3625213. URL https://dl.acm.org/doi/10.1145/3605770.3625213.

[17] Dependency Graph SBoM export for older repository versions?, 2024. URL https://github.com/orgs/community/discussions/118612.

[18] C. J. Alberts, A. J. Dorofee, R. Creel, R. J. Ellison, and C. Woody. A Systemic Approach for Assessing Software Supply-Chain Risk. In *2011 44th Hawaii International Conference on System Sciences*, pages 1–8, Kauai, HI, Jan. 2011. ISBN 978-1-4244-9618-1. doi: 10.1109/HICSS.2011.36.

[19] Anchore. Grype. URL https://github.com/anchore/grype/.

[20] Argon. 2021 software supply chain security report. https://info.aquasec.com/argon-supply-chain-attacks-study, 2021.

[21] M. Arvan, L. Pina, and N. Parde. Reproducibility in computational linguistics: Is source code enough? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2350–2361, 2022.

[22] Atlassian. Bitbucket. https://bitbucket.org, . [Online; accessed 10-march-2023].

[23] Atlassian. Branch permissions. https://support.atlassian.com/bitbucket-cloud/docs/use-branch-permissions/, . [Online; accessed 10-march-2023].

[24] Atlassian. Variables and secrets: Bitbucket cloud. https://support.atlassian.com/bitbucket-cloud/docs/variables-and-secrets/, . [Online; accessed 10-march-2023].

[25] Atlassian. Pipeline start conditions. https://support.atlassian.com/bitbucket-cloud/docs/pipeline-start-conditions/, . [Online; accessed 10-march-2023].

[26] Atlassian. Deployment permissions now available in bitbucket pipelines. https://bitbucket.org/blog/deployment-permissions-now-available-in-bitbucket-pipelines, Dec 2022.

[27] Atlassian, Inc. Atlassian. https://www.atlassian.com, . [Online; accessed 10-march-2023].

[28] Atlassian, Inc. Jira. https://bitbucket.org/product/features/pipelines/integrations, . [Online; accessed 10-march-2023].

[29] Atlassian, Inc. Jira. https://www.atlassian.com/software/jira, . [Online; accessed 10-march-2023].

[30] R. Bajaj, E. Fernandes, B. Adams, and A. E. Hassan. Unreproducible builds: time to fix, causes, and correlation with external ecosystem factors. *Empirical Software Engineering*, 29(1):11, Feb. 2024. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-023-10399-4. URL https://link.springer.com/10.1007/s10664-023-10399-4.

[31] M. Balliu, B. Baudry, S. Bobadilla, M. Ekstedt, M. Monperrus, J. Ron, A. Sharma, G. Skoglund, C. Soto-Valero, and M. Wittlinger. Challenges of producing software bill of materials for java. *IEEE Security & Privacy*, 2023.

[32] A.-L. Barabási. *Network Science*. Cambridge University Press, 2016. ISBN 978-1-10707-626-6. URL http://networksciencebook.com/.

[33] O. Ben-Kiki, C. Evans, and B. Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008*, 5:11, 2009.

[34] G. Benedetti, O. Solarin, G. Tystahl, W. Enck, C. Kästner, A. Kapravelos, A. Merlo, and L. Verderame. Replication package: An empirical study on reproducible packaging in open-source ecosystems. URL https://osf.io/vmnsh/?view_only=776597f9dd604eff93b13299c8ce252c.

[35] G. Benedetti, L. Verderame, and A. Merlo. Alice in (software supply) chains: Risk identification and evaluation. In *Quality of Information and Communications Technology*, pages 281–295, Cham, 2022. Springer International Publishing. ISBN 978-3-031-14179-9. doi: 10.1007/978-3-031-14179-9_19.

[36] G. Benedetti, L. Verderame, and A. Merlo. Automatic security assessment of github actions workflows. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, SCORED'22, page 37–45, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450398855. doi: 10.1145/3560835.3564554. URL https://doi.org/10.1145/3560835.3564554.

[37] T. Bi, B. Xia, Z. Xing, Q. Lu, and L. Zhu. On the way to sboms: Investigating design issues and solutions in practice. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–25, 2024.

[38] J. R. Biden. Executive Order on Improving the Nation's Cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/, 2021.

[39] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, Seattle WA USA, Nov. 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950325. URL https://dl.acm.org/doi/10.1145/2950290.2950325.

[40] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Trans. Softw. Eng. Methodol.*, 30(4), jul 2021. ISSN 1049-331X. doi: 10.1145/3447245. URL https://doi.org/10.1145/3447245.

[41] R. Builds. Reproducible builds organization. https://reproducible-builds.org/, 2020.

[42] R. Builds. Source_date_epoch standard. https://reproducible-builds.org/specs/source-date-epoch/, 2020.

[43] S. Butler, J. Gamalielsson, B. Lundell, C. Brax, A. Mattsson, T. Gustavsson, J. Feist, B. Kvarnström, and E. Lönroth. On business adoption and use of reproducible builds for open and closed source software. *Software Quality Journal*, 31(3):687–719, Sept. 2023. ISSN 0963-9314, 1573-1367. doi: 10.1007/s11219-022-09607-z. URL https://link.springer.com/10.1007/s11219-022-09607-z.

[44] F. Calefato, F. Lanubile, and L. Quaranta. A preliminary investigation of mlops practices in github. In *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '22, page 283–288, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394277. doi: 10.1145/3544902.3546636. URL https://doi.org/10.1145/3544902.3546636.

[45] M. Canesche, R. Leissa, and F. M. Q. Pereira. Preparing reproducible scientific artifacts using docker. *arXiv preprint arXiv:2308.14122*, 2023.

[46] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 14–14, 2007. doi: 10.1109/MSR.2007.14.

[47] G. Cardoen, T. Mens, and A. Decan. A dataset of GitHub actions workflow histories. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 677–681, New York, NY, USA, Apr. 2024. ACM.

[48] S. Cass. The Top Programming Languages 2024 . https://spectrum.ieee.org/ibm-quantum-computer-2668978269, 2024.

[49] C. Chandrasekara and P. Herath. *Introduction to GitHub Actions*, pages 1–8. Apress, Berkeley, CA, 2021. ISBN 978-1-4842-6464-5. doi: 10.1007/978-1-4842-6464-5_1. URL https://doi.org/10.1007/978-1-4842-6464-5_1.

[50] T. Chen, Y. Zhang, S. Chen, T. Wang, and Y. Wu. Let's supercharge the workflows: An empirical study of github actions. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 01–10, 2021. doi: 10.1109/QRS-C55045.2021.00163.

[51] C. Cimpanu. Backdoored python library caught stealing ssh credentials. https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/, May 2018.

[52] S. Cofano, G. Benedetti, and M. Dell'Amico. SBOM generation tools in the python ecosystem: an in-detail analysis. *arXiv [cs.CR]*, Sept. 2024.

[53] C. Collberg and T. A. Proebsting. Repeatability in computer systems research. *Communications of the ACM*, 59(3):62–69, 2016.

[54] Compiler-dev. Javac determinism. https://mail.openjdk.org/pipermail/compiler-dev/2023-December/025215.html, 2023.

[55] O. P. A. contributors. Open policy agent. https://www.openpolicyagent.org, 2022.

[56] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2017. ISBN 978-81-203-4007-7 81-203-4007-8.

[57] R. Cox. Perfectly reproducible, verified go toolchains. https://go.dev/blog/rebuild, 2023.

[58] R. Cox. Timeline of the xz open source attack. https://research.swtch.com/xz-timeline, Apr. 2024.

[59] A. Cumming. Open source intelligence (osint): Issues for congress, 2007.

[60] Cybersecurity and I. S. Agency. *Defending Against Software Supply Chain Attacks*. 2021. URL https://www.cisa.gov/sites/default/files/publications/defending_against_software_supply_chain_attacks_508_1.pdf.

[61] Cybersecurity & Infrastructure Security Agency. SBOM FAQ. https://www.cisa.gov/resources-tools/resources/sbom-faq, 2024.

[62] Darkport Technologies Limited. shhgit. https://github.com/eth0izzle/shhgit.

[63] X. De Carné De Carnavalet and M. Mannan. Challenges and implications of verifiable builds for security-critical open-source software. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 16–25, New Orleans Louisiana USA, Dec. 2014. ACM. ISBN 978-1-4503-3005-3. doi: 10.1145/2664243.2664288. URL https://dl.acm.org/doi/10.1145/2664243.2664288.

[64] Debian. Buildinfofiles. https://wiki.debian.org/ReproducibleBuilds/BuildinfoFiles, 2018. URL https://wiki.debian.org/ReproducibleBuilds/BuildinfoFiles.

[65] A. Decan, T. Mens, M. Claes, and P. Grosjean. On the Development and Distribution of R Packages: An Empirical Analysis of the R Ecosystem. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, pages 1–6, Dubrovnik Cavtat Croatia, Sept. 2015. ACM. ISBN 978-1-4503-3393-1. doi: 10.1145/2797433.2797476. URL https://dl.acm.org/doi/10.1145/2797433.2797476.

[66] A. Decan, T. Mens, and M. Claes. On the topology of package dependency networks: a comparison of three programming language ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, pages 1–4, Copenhagen Denmark, Nov. 2016. ACM. ISBN 978-1-4503-4781-5. doi: 10.1145/2993412.3003382. URL https://dl.acm.org/doi/10.1145/2993412.3003382.

[67] A. Decan, T. Mens, M. Claes, and P. Grosjean. When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 493–504, Suita, Mar. 2016. IEEE. ISBN 978-1-5090-1855-0. doi: 10.1109/SANER.2016.12. URL http://ieeexplore.ieee.org/document/7476669/.

[68] A. Decan, T. Mens, and M. Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2–12, Klagenfurt, Austria, Feb. 2017. IEEE. ISBN 978-1-5090-5501-2. doi: 10.1109/SANER.2017.7884604. URL http://ieeexplore.ieee.org/document/7884604/.

[69] A. Decan, T. Mens, and P. Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24:381–416, 2 2019. ISSN 15737616. doi: 10.1007/s10664-017-9589-y.

[70] A. Decan, T. Mens, P. R. Mazrae, and M. Golzadeh. On the use of github actions in software development repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245, 2022. doi: 10.1109/ICSME55016.2022.00029.

[71] A. Decan, T. Mens, and H. Onsori Delicheh. On the outdatedness of workflows in the github actions ecosystem. *Journal of Systems and Software*, 206:111827, 2023. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2023.111827.

[72] Deepbits. Evaluating and Benchmarking SBOM Generators: A Systematic Approach. https://www.deepbits.com/whitepaper/1, 2023.

[73] devrandom. Gitian: a secure software distribution method. https://github.com/devrandom/gitian-builder, 2011.

[74] J. Dietrich, S. Rasheed, A. Jordan, and T. White. On the security blind spots of software composition analysis, 2023.

[75] M. Dowd, J. McDonald, and J. Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[76] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *IEEE Software*, 33(3): 94–100, 2016. doi: 10.1109/MS.2016.68.

[77] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *IEEE Softw.*, (3): 94–100, may 2016. ISSN 0740-7459. doi: 10.1109/MS.2016.68. URL https://doi.org/10.1109/MS.2016.68.

[78] W. Enck and L. Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Secur. Priv.*, 20 (2):96–100, Mar. 2022.

[79] EU. Eu cyber resilience act, 2022. URL https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act.

[80] European Union Agency for Cybersecurity. *ENISA threat landscape for supply chain attacks*. Publications Office, 2021. URL https://data.europa.eu/doi/10.2824/168593.

[81] European Union Agency for Cybersecurity. *ENISA threat landscape for supply chain attacks*. Publications Office, LU, 2021. URL https://data.europa.eu/doi/10.2824/168593.

[82] FIRST.ORG, Inc. CVSS. https://www.first.org/cvss/.

[83] C. Flynn. PyPI Stats. https://pypistats.org/packages/__all__.

[84] B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative Inquiry*, 12(2):219–245, 2006. doi: 10.1177/1077800405284363. URL https://doi.org/10.1177/1077800405284363.

[85] M. Fourné, D. Wermke, W. Enck, S. Fahl, and Y. Acar. It's like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 1527–1544, May 2023. doi: 10.1109/SP46215.2023.10179320.

[86] Free Software Foundation. Crontab. https://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html.

[87] B. Gates. Memo from bill gates. https://news.microsoft.com/2012/01/11/memo-from-bill-gates/, 2012.

[88] A. Gaurav. Enable repeatable package restores using a lock file. https://devblogs.microsoft.com/nuget/enable-repeatable-package-restores-using-a-lock-file/, 2018.

[89] S. M. Ghaffarian and H. R. Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.*, (4), aug 2017. doi: 10.1145/3092566. URL https://doi.org/10.1145/3092566.

[90] GitHub. GitHub actions. https://docs.github.com/en/actions, 2022.

[91] GitHub. GitHub contexts - github. https://docs.github.com/en/actions/learn-github-actions/contexts#github-context, 2022.

[92] GitHub. Using filters. https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#using-filters, 2022.

[93] GitHub. Security hardening for github actions. https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions, 2022.

[94] GitHub. Security hardening for GitHub actions: Restricting permissions for tokens. https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#restricting-permissions-for-tokens, 2022.

[95] GitHub. Security hardening for GitHub actions: Using secrets. https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#using-secrets, 2022.

[96] GitHub. Security hardening for github actions: Using third-party actions. https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#using-third-party-actions, 2022.

[97] GitHub. Reusing workflows. https://docs.github.com/en/actions/using-workflows/reusing-workflows, 2022.

[98] GitHub. GitHub contexts - secrets. https://docs.github.com/en/actions/learn-github-actions/contexts#secrets-context, 2022.

[99] GitHub. Automatic token authentication. https://docs.github.com/en/actions/security-guides/automatic-token-authentication, 2022.

[100] GitHub, Inc. Github. https://github.com, . [Online; accessed 10-march-2023].

[101] GitHub, Inc. Github actions. https://docs.github.com/en/actions, . [Online; accessed 10-march-2023].

[102] GitHub, Inc. Security hardening for github actions. https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions, . [Online; accessed 10-march-2023].

[103] GitHub, Inc. Events that trigger workflows. https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows, . [Online; accessed 10-march-2023].

[104] GitLab B.V. Gitlab. https://gitlab.com, . [Online; accessed 10-march-2023].

[105] GitLab B.V. Gitlab - members of a project. https://docs.gitlab.com/ee/user/project/members/index.html, . [Online; accessed 10-march-2023].

[106] GitLab B.V. Gitlab ci/cd pipelines. https://docs.gitlab.com/ee/ci/pipelines/, . [Online; accessed 10-march-2023].

[107] GitLab B.V. Gitlab - webhooks. https://docs.gitlab.com/ee/user/project/integrations/webhooks.html, . [Online; accessed 10-march-2023].

[108] GNU. Bash reference manual - command substitution. https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Command-Substitution, 2020.

[109] GNU. Bash reference manual - here documents. https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Here-Documents, 2020.

[110] M. Golzadeh, A. Decan, and T. Mens. On the rise and fall of ci services in github. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 662–672, 2022. doi: 10.1109/SANER53432.2022.00084.

[111] Q. Gong, J. Zhang, Y. Chen, Q. Li, Y. Xiao, X. Wang, and P. Hui. Detecting malicious accounts in online developer communities using deep learning. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, page 1251–1260, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369763. doi: 10.1145/3357384.3357971. URL https://doi.org/10.1145/3357384.3357971.

[112] D. Goodin. What we know about the xz utils backdoor that almost infected the world. ars Technica, Apr. 2024. https://arstechnica.com/security/2024/04/what-we-know-about-the-xz-utils-backdoor-that-almost-infected-the-world/.

[113] P. Gorski, L. Lo Iacono, S. Wiefling, and S. Möller. Warn if secure or how to deal with security by default in software development? 08 2018.

[114] P. Goswami, S. Gupta, Z. Li, N. Meng, and D. Yao. Investigating the reproducibility of npm packages. pages 677–681. Institute of Electrical and Electronics Engineers Inc., 9 2020. ISBN 9781728156194. doi: 10.1109/ICSME46990.2020.00071.

[115] P. Goswami, S. Gupta, Z. Li, N. Meng, and D. Yao. Investigating the reproducibility of npm packages. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 677–681, 2020. doi: 10.1109/ICSME46990.2020.00071.

[116] Graphviz Authors. Graphviz. https://graphviz.org/.

[117] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu. An empirical study of malicious code in pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, 2023.

[118] A. Halbritter and D. Merli. Accuracy evaluation of SBOM tools for web applications and system-level software. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, New York, NY, USA, July 2024. ACM.

[119] M. Hashemi. Calendar versioning — CalVer. https://calver.org/. Accessed: 2024-9-3.

[120] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios. Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering*, 27, 9 2022. ISSN 15737616. doi: 10.1007/s10664-021-10071-9.

[121] T. Herr, W. Loomis, E. Schroeder, S. Scott, S. Handler, T. Zuo, and Atlantic Council of the United States. *Broken trust: lessons from Sunburst*. 2021. ISBN 978-1-61977-168-0. URL https://www.atlanticcouncil.org/in-depth-research-reports/report/broken-trust-lessons-from-sunburst/. OCLC: 1244810221.

[122] Hex-Rays. Ida Decompiler. https://hex-rays.com/decompiler/.

[123] Idera, Inc. Travis ci. https://www.travis-ci.com/. [Online; accessed 10-march-2023].

[124] P. Jenkins and J. Cassou. *Jenkins*. Gerd Hatje, 1963.

[125] P. E. Kaloroumakis and M. J. Smith. Toward a knowledge graph of cybersecurity countermeasures. 2021.

[126] M. Keshani, T.-G. Velican, G. Bot, and S. Proksch. Aroma: Automatic reproduction of maven artifacts. *Proc. ACM Softw. Eng.*, 1(FSE), jul 2024. doi: 10.1145/3643764. URL https://doi.org/10.1145/3643764.

[127] A. Y. Kim, V. Herrmann, R. Barreto, B. Calkins, E. Gonzalez-Akre, D. J. Johnson, J. A. Jordan, L. Magee, I. R. McGregor, N. Montero, K. Novak, T. Rogers, J. Shue, and K. J. Anderson-Teixeira. Implementing github actions continuous integration to reduce error rates in ecological data collection. *Methods in Ecology and Evolution*, 13 (11):2572–2585, 2022. doi: 10.1111/2041-210X.13982.

[128] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude. How do software developers use github actions to automate their workflows? In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 420–431, 2021. doi: 10.1109/MSR52588.2021.00054.

[129] I. Koishybayev, A. Nahapetyan, R. Zachariah, S. Muralee, B. Reaves, A. Kapravelos, and A. Machiry. Characterizing the security of github CI workflows. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2747–2763, Boston, MA, Aug. 2022. USENIX Association. ISBN 978-1-939133-31-1.

[130] I. Koishybayev, A. Nahapetyan, R. Zachariah, S. Muralee, B. Reaves, A. Kapravelos, and A. Machiry. Characterizing the security of github CI workflows. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2747–2763, Boston, MA, Aug. 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/koishybayev.

[131] P. Ladisa, H. Plate, M. Martinez, and O. Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526, Los Alamitos, CA, USA, may 2023. IEEE Computer Society. doi: 10.1109/SP46215.2023.00010.

[132] C. Lamb and S. Zacchiroli. Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software*, 39(2):62–70, Mar. 2022.

[133] H. Levsen, C. Lamb, and M. Rizzolo. reprotest. https://salsa.debian.org/reproducible-builds/reprotest, 2016. URL https://salsa.debian.org/reproducible-builds/reprotest.

[134] H. Levsen et al. Reproducible Debian overview. https://tests.reproducible-builds.org/debian/reproducible.html, 2023.

[135] B. Liu, L. Shi, Z. Cai, and M. Li. Software vulnerability discovery techniques: A survey. In *2012 Fourth International Conference on Multimedia Information Networking and Security*, pages 152–156, 2012. doi: 10.1109/MINES.2012.202.

[136] M. Logan. GitHub action runners: Analyzing the environment and security in action. https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/github-action-runners-analyzing-the-environment-and-security-in-action, 2022.

[137] Mackenzie Jackson. Codecov supply chain breach - explained step by step. https://blog.gitguardian.com/codecov-supply-chain-breach/.

[138] Maltego Technologies. Maltego. https://www.maltego.com/.

[139] K. Manikas. Revisiting software ecosystems research: A longitudinal literature study. *Journal of Systems and Software*, 117:84–103, 2016. ISSN 0164-1212. doi: 10.1016/j.jss.2016.02.003.

[140] A. Maven. Configuring for reproducible builds. https://maven.apache.org/guides/mini/guide-reproducible-builds.html, 2024.

[141] K. Merrill, Z. Newman, S. Torres-Arias, and K. R. Sollins. Speranza: Usable, privacy-friendly software signing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 3388–3402, 2023.

[142] D. Meyer and H. Plate. 2024 dependency management report. https://www.endorlabs.com/lp/2024-dependency-management-report. Accessed: 2024-9-23.

[143] M. Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, 2014. doi: 10.1109/MS.2014.58.

[144] R. Monat, A. Ouadjaout, and A. Miné. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In C. Drăgoi, S. Mukherjee, and K. Namjoshi, editors, *Static Analysis*, volume 12913, pages 323–345. Springer International Publishing, Cham, 2021. ISBN 978-3-030-88805-3 978-3-030-88806-0. doi: 10.1007/978-3-030-88806-0_16. URL https://link.springer.com/10.1007/978-3-030-88806-0_16. Series Title: Lecture Notes in Computer Science.

[145] S. Muralee, I. Koishybayev, A. Nahapetyan, G. Tystahl, B. Reaves, A. Bianchi, W. Enck, A. Kapravelos, and A. Machiry. {ARGUS}: A Framework for Staged Static Taint Analysis of {GitHub} Workflows and Actions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6983–7000, 2023. ISBN 978-1-939133-37-3.

[146] National Security Agency. Ghidra. https://ghidra-sre.org/.

[147] Neo4j. Neo4j. https://neo4j.com/.

[148] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford. CHAINIAC: Proactive Software-Update transparency via collectively signed skipchains and verified builds. In *Proceedings of the 26th USENIX Security Symposium (Sec'17)*, pages 1271–1287, Aug. 2017.

[149] NTIA. The minimum elements for a software bill of materials (SBOM).

[150] H. Onsori Delicheh, A. Decan, and T. Mens. Quantifying security issues in reusable JavaScript actions in GitHub workflows. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 692–703, New York, NY, USA, Apr. 2024. ACM.

[151] OWASP. *OWASP Software Component Verification Standard*. 2020. URL https://owasp.org/www-project-software-component-verification-standard/.

[152] OWASP Foundation, Inc. Owasp dependency-check. https://owasp.org/www-project-dependency-check/.

[153] S. Ozkan. NVD leaves thousands of vulnerabilities without analysis data. https://securityscorecard.com/blog/national-vulnerability-database-nvd-leaves-thousands-of-vulnerabilities-without-analysis-data/, 2024.

[154] I. Pashchenko, D.-L. Vu, and F. Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1513–1531, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417232. URL https://doi.org/10.1145/3372297.3417232.

[155] S. Peisert, B. Schneier, H. Okhravi, F. Massacci, T. Benzel, C. Landwehr, M. Mannan, J. Mirkovic, A. Prakash, and J. B. Michael. Perspectives on the solarwinds incident. *IEEE Security & Privacy*, 19(2):7–13, 2021.

[156] M. Perry. Deterministic builds part two: Technical details. https://blog.torproject.org/deterministic-builds-part-two-technical-details/, Oct. 2013.

[157] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*, pages 507–517. IEEE, 2019.

[158] pip developers. More on Dependency Resolution. https://pip.pypa.io/en/stable/topics/more-dependency-resolution/, 2024.

[159] H. Plate, S. E. Ponta, and A. Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–420, 2015. doi: 10.1109/ICSM.2015.7332492.

[160] S. E. Ponta, H. Plate, and A. Sabetta. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 449–460, 2018. doi: 10.1109/ICSME.2018.00054.

[161] S. E. Ponta, H. Plate, and A. Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25: 3175–3215, 9 2020. ISSN 15737616. doi: 10.1007/s10664-020-09830-x.

[162] T. Preston-Werner. Semantic versioning 2.0.0. https://semver.org/. Accessed: 2024-9-3.

[163] M. F. Rabbi, A. I. Champa, C. Nachuma, and M. F. Zibran. Sbom generation tools under microscope: A focus on the npm ecosystem. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pages 1233–1241, 2024.

[164] radareorg. Radare2. https://rada.re/.

[165] Radware. Log4shell: Critical log4j vulnerability. https://www.radware.com/security/threat-advisories-and-attack-reports/log4shell-critical-log4j-vulnerability/, 2021.

[166] G. A. Randrianaina, D. E. Khelladi, O. Zendra, and M. Acher. Options matter: Documenting and fixing non-reproducible builds in highly-configurable systems. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 654–664, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705878. doi: 10.1145/3643991.3644913. URL https://doi.org/10.1145/3643991.3644913.

[167] G. A. Randrianaina, D. E. Khelladi, O. Zendra, and M. Acher. Options matter: Documenting and fixing non-reproducible builds in highly-configurable systems. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 654–664, New York, NY, USA, Apr. 2024. ACM.

[168] Z. Ren, H. Jiang, J. Xuan, and Z. Yang. Automated localization for unreproducible builds. In *Proceedings of the 40th International Conference on Software Engineering*, pages 71–81, Gothenburg Sweden, May 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180224. URL https://dl.acm.org/doi/10.1145/3180155.3180224.

[169] Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie. Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 527–538, San Diego, CA, USA, Nov. 2019. IEEE. ISBN 978-1-72812-508-4. doi: 10.1109/ASE.2019.00056. URL https://ieeexplore.ieee.org/document/8952375/.

[170] Z. Ren, S. Sun, J. Xuan, X. Li, Z. Zhou, and H. Jiang. Automated patching for unreproducible builds. In *Proceedings of the 44th International Conference on Software Engineering*, pages 200–211, Pittsburgh Pennsylvania, May 2022. ACM. ISBN 978-1-4503-9221-1. doi: 10.1145/3510003.3510102. URL https://dl.acm.org/doi/10.1145/3510003.3510102.

[171] reproducible builds.org. diffoscope in-depth comparison of files, archives, and directories. https://diffoscope.org/, 2014. URL https://diffoscope.org/.

[172] ReproducibleBuilds. Reproduciblebuilds. https://reproducible-builds.org/.

[173] Revenera. The 2022 state of the software supply chain report. https://info.revenera.com/SCA-RPT-OSS-License-Compliance-2022/, 2022.

[174] G. Rousseau, R. Di Cosmo, and S. Zacchiroli. Software provenance tracking at the scale of public source code. *Empirical Software Engineering*, 25(4):2930–2959, July 2020. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-020-09828-5. URL https://link.springer.com/10.1007/s10664-020-09828-5.

[175] Scribe. Gitgat. https://github.com/scribe-public/gitgat, 2022.

[176] T. Segura. GitHub actions security best practices. https://blog.gitguardian.com/github-actions-security-cheat-sheet/, 2022.

[177] M. Shahin, M. Ali Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5: 3909–3943, 2017. doi: 10.1109/ACCESS.2017.2685629.

[178] A. Sharma, M. Wittlinger, B. Baudry, and M. Monperrus. Sbom.exe: Countering dynamic code injection based on software bill of materials in java, 2024. URL https://arxiv.org/abs/2407.00246.

[179] Y. Shi, M. Wen, F. R. Cogo, B. Chen, and Z. M. Jiang. An Experience Report on Producing Verifiable Builds for Large-Scale Commercial Systems. *IEEE Transactions on Software Engineering*, 48(9):3361–3377, Sept. 2022. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2021.3092692. URL https://ieeexplore.ieee.org/document/9465650/.

[180] ShiftLeftSecurity. slscan. https://slscan.io/.

[181] J. F. Shobe, M. Y. Karim, M. B. Zanjani, and H. Kagdi. On mapping releases to commits in open source systems. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 68–71, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328791. doi: 10.1145/2597008.2597792. URL https://doi.org/10.1145/2597008.2597792.

[182] Shodan. Shodan. https://www.shodan.io/.

[183] Snyk Limited. Snyk open source. https://snyk.io/.

[184] M. Stanek. Secure by default - the case of tls. 2017. doi: 10.48550/arXiv.1708.07569.

[185] Z. Steindler. How to Make Programming Language Package Repositories More Secure, 2024. URL https://openssf.org/blog/2024/07/31/how-to-make-programming-language-package-repositories-more-secure/.

[186] StepSecurity, Inc. Stepsecurity - securerepo. https://github.com/step-security/secure-repo#1-automatically-set-minimum-github_token-permissions. [Online; accessed 10-march-2023].

[187] Synopsis. 2023 ossra report. URL https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2023.pdf.

[188] The Linux Foundation. SLSA. https://slsa.dev/.

[189] The MITRE Corporation. CVE. https://cve.mitre.org/, .

[190] The MITRE Corporation. MITRE Common Weakness Enumeration. https://cwe.mitre.org/, .

[191] The MITRE Corporation. CWE-20: Improper Input Validation. https://cwe.mitre.org/data/definitions/20.html, .

[192] The MITRE Corporation. CWE-478: Missing Default Case in Switch Statement. https://cwe.mitre.org/data/definitions/478.html, .

[193] The MITRE Corporation. CWE-798: Use of Hard-coded Credentials. https://cwe.mitre.org/data/definitions/798.html, .

[194] The MITRE Corporation. CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). https://cwe.mitre.org/data/definitions/89.html, .

[195] The MITRE Corporation. CWE VIEW 699: Software Development. https://cwe.mitre.org/data/definitions/699.html, .

[196] The MITRE Corporation. MITRE ATT&CK. https://attack.mitre.org/, .

[197] K. Thompson. Reflections on trusting trust. *Commun. ACM*, pages 761—-763, 1984.

[198] Tinder. gh-workflow-auditor. https://github.com/TinderSec/gh-workflow-auditor, 2022.

[199] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1393–1410, Santa Clara, CA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

[200] S. Torres-Arias, D. Geer, and J. S. Meyers. A viewpoint on knowing software: Bill of materials quality when you see it. *IEEE Secur. Priv.*, 21(6):50–54, Nov. 2023.

[201] P. Valenzuela-Toledo and A. Bergel. Evolution of github action workflows. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 123–127, 2022. doi: 10.1109/SANER53432.2022.00026.

[202] I. van den Berk, S. Jansen, and L. Luinenburg. Software ecosystems: a software ecosystem strategy assessment model. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, page 127–134, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450301794. doi: 10.1145/1842752.1842781. URL https://doi.org/10.1145/1842752.1842781.

[203] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. ESEC/FSE 2015, page 805–816, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786850.

[204] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta. Typosquatting and combosquatting attacks on the python ecosystem. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 509–514, 2020. doi: 10.1109/EuroSPW51379.2020.00074.

[205] D.-L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. Sabetta. Lastpymile: identifying the discrepancy between sources and packages. ESEC/FSE 2021, page 780–792, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3468592.

[206] J. Wallen. Scan container images for vulnerabilities with grype. https://thenewstack.io/scan-container-images-for-vulnerabilities-with-grype/, May 2022. Accessed: 2024-9-5.

[207] D. Wermke, J. H. Klemmer, N. Wöhler, J. Schmüser, H. S. Ramulu, Y. Acar, and S. Fahl. " always contribute back": A qualitative study on security challenges of the open source supply chain. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1545–1560. IEEE, 2023.

[208] M. Wessel, J. Vargovich, M. A. Gerosa, and C. Treude. Github actions: The impact on the pull request process, 2022. https://arxiv.org/abs/2206.14118.

[209] S. Winter, C. S. Timperley, B. Hermann, J. Cito, J. Bell, M. Hilton, and D. Beyer. A retrospective study of one decade of artifact evaluations. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 145–156, 2022.

[210] D. Yan, Y. Niu, K. Liu, Z. Liu, Z. Liu, and T. F. Bissyandé. Estimating the attack surface from residual vulnerabilities in open source software supply chain. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 493–502, 2021. doi: 10.1109/QRS54544.2021.00060.

[211] S. Yu, W. Song, X. Hu, and H. Yin. On the correctness of metadata-based sbom generation: A differential analysis approach. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 29–36, 2024. doi: 10.1109/DSN58291.2024.00018.

[212] N. Zahan, Y. Acar, M. Cukier, W. Enck, C. Kastner, A. Kapravelos, D. Wermke, and L. Williams. S3C2 summit 2023-11: Industry secure supply chain summit. Aug. 2024.

[213] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta. Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 471–482, 2021. doi: 10.1109/ICSME52107.2021.00048.

[214] Y. Zhang, Y. Fan, S. Hou, Y. Ye, X. Xiao, P. Li, C. Shi, L. Zhao, and S. Xu. Cyber-guided deep neural network for malicious repository detection in github. In *2020 IEEE International Conference on Knowledge Graph (ICKG)*, pages 458–465, 2020. doi: 10.1109/ICBK50248.2020.00071.

Appendix A

Appendices

A.1 Causes of reproducibility issues

Listings A.1-A.5 are examples of dynamic metadata that causes packages not to be reproducible. We categorize the causes as follows.

- **File Ordering:** RubyGems allows the developer to dynamically specify the list of files to be included within the package (e.g., Listing A.4). Hence, the order of the files depends on the method used to obtain them. Many packages use `git ls-files` command list files (e.g., Listing A.5). Unfortunately, the order depends on the system locale and, therefore, is not deterministic. More generally, this example shows how variations in the build infrastructure can trigger the same unreproducibility problem. Varying both the file order and the locale results in the same reproducibility issue.

- **Locales:** As shown in the file ordering example, locales can affect build reproducibility in multiple ways. Some of the examples we collected show that the locale affects reproducibility Other examples include: i) when text is inserted inside of a file during the build, ii) when functions are used to collect textual elements (as the file ordering example shows), and iii) when a file is read during the build process.

- **Umask:** Developers sometimes perform file manipulation through dynamic metadata (e.g., Listing A.1). Permissions are also modified following system policies when files are created, copied, or moved. For PyPI, most of the reproducibility issues caused by umask are in archive metadata, as happens for timestamps in Section 5.2.2.

- **Time and Timezone:** Time-related functions are sometimes used when creating dynamic metadata. For example, the setuptools PyPI build backend allows characterizing

the version with the date (Listing A.2), and the RubyGems date macro initializes a metadata value to the current date (Listing A.3).

```
1 if /gem/i =~ $0
2 ‘ruby scripts/mkchangelog.rb ChangeLog.txt‘
3 end
4
5
6 $gemspec = Gem::Specification.new do |s|
7
8   s.name     = "nice-ffi"
```

Listing A.1 Creating or modifying files during the build process using arbitrary scripts or function may cause unreproducibility issues since the non-determinism of the used umask value (line 2).

```
1 [egg_info]
2 tag_build = .post
3 tag_date = 1
```

Listing A.2 Reproducibility can be affected by using time-dependent release tags version (3).

```
1 Gem::Specification.new do |s|
2   s.name        = '...'
3   s.version     = '...'
4   s.date = @date
5   s.description = '...'
6   s.authors     = ["..."]
7   s.email       = '...'
8   s.homepage    = 'https://github.com/.../...'
9 end
```

Listing A.3 Inserting date in metadata (line 4) cause unreproducibility issues. Date is inserted with day-wise granularity.

```
1 Gem::Specification.new do |s|
2   s.name        = '...'
3   s.version     = '...'
4   s.date        = '...'
5   s.summary     = File.read("README.markdown").split(/===+/)[1].strip.split("\n")[0]
6   s.description = s.summary
7   s.authors     = ["..."]
8   s.email       = '...'
```

```
1  Gem::Specification.new do |s|
2    s.name        = "..."
3    s.version     = ...
4    s.platform    = Gem::Platform::RUBY
5    s.authors     = ["..."]
6    s.email       = ["..."]
7    s.homepage    = "https://github.com/.../..."
8
9     s.files         = `find *`.split("\n").uniq.sort.select{|f| !f.empty? }
10    s.test_files    = `find spec/*`.split("\n")
11   s.executables   = []
12   s.require_paths = ["lib"]
13 end
```

Listing A.4 Listing files using non-deterministic functions cause unreproducibility issues. In line 9 this issue is addressed while in line 10 it is not.

```
9    s.homepage    = 'https://github.com/.../...'
10
11   s.files = `git ls-files`.split("\n")
12 end
```

Listing A.5 Locales can cause issues when reading from a file as happening at line 5. The output is not deterministic without a specific locale value set for the build process.

A.2 Package Managers Patches

We now describe how we patched the RubyGems and PyPI package managers to promote reproducible package builds.

A.2.1 RubyGems Patches

We patched the RubyGems package manager in several ways. Our first patch (Listing A.6) forces `SOURCE_DATE_EPOCH` to epoch 0. While this variable can be set in the build environment (Section 5.2.2), using a fixed value avoids the need for developers to document and communicate the value with packages. Setting it to 0 is similar to npm and Cargo's use of a standard value for all packages. Our second patch (Listing A.7) similarly sets the date to `nil`. Our third patch (Listing A.8) sorts several well-defined arrays of files (e.g., `@files` and `@extensions`). This post-processing ensures the values are listed in the same order, regardless of the environment used to derive the list. Our fourth patch (Listing A.9) sets the default encoding to `Encoding::UTF_8` to make the locale encoding deterministic. Our fifth

and final patch (Listing A.10) sets the default umask to `0o022` to ensure all files are created with the same umask.

```
    @lib/rubygems/package/tar_writer.rb
    def add_file(name, mode) # :yields: io
        ...

        header = Gem::Package::TarHeader.new name: name, mode: mode, size: size, prefix:
    prefix, mtime: Gem.source_date_epoch

        ...
        self
      end

    @lib/rubygems.rb
    def self.source_date_epoch
         - Time.at(source_date_epoch_string.to_i).utc.freeze
         + Time.at(0).utc.freeze
    end

```

Listing A.6 The `SOURCE_DATE_EPOCH` variable is used throughout the toolchain referring to the homonimous environment variable. Forcing the `SOURCE_DATE_EPOCH` variable to epoch 0 solves the timestamp issue in the `gem` archives without any action required by the developer.

```
    def date=(date)
         - @date = case date
         - when String then
         - if DateTimeFormat =  date
         - Time.utc($1.to_i, $2.to_i, $3.to_i)
         - else
         - raise(Gem::InvalidSpecificationException,
         - "invalid date format in specification: #date.inspect")
         - end
         - when Time, DateLike then
         - Time.utc(date.year, date.month, date.day)
         - else
         - TODAY
         - end
         + @date = nil
    end

```

Listing A.7 Removing the capability of returning a non-deterministic date value by the toolchain it is possible to avoid unreproducibility because of date in metadata.

```
    def normalize
        if defined?(@extra_rdoc_files) && @extra_rdoc_files
            @extra_rdoc_files.uniq
            @files ||= []
            @files.concat(@extra_rdoc_files)
        end

        - @files = @files.uniq if @files
        - @extensions = @extensions.uniq if @extensions
        - @test_files = @test_files.uniq if @test_files
        - @executables = @executables.uniq if @executables
        - @extra_rdoc_files = @extra_rdoc_files.uniq if @extra_rdoc_files
        + @files = @files.uniq.sort if @files
        + @extensions = @extensions.uniq.sort if @extensions
        + @test_files = @test_files.uniq.sort if @test_files
        + @executables = @executables.uniq.sort if @executables
        + @extra_rdoc_files = @extra_rdoc_files.uniq.sort if @extra_rdoc_files
    end
```

Listing A.8 The application of a sorting function to the list of files retrieved by the configuration file can prevent unreproducibility issues in RubyGems packages.

```
    def self.load(file)
        return unless file

        spec = @load_cache_mutex.synchronize { @load_cache[file] }
        return spec if spec

        return unless File.file?(file)

        code = Gem.open_file(file, "r:UTF-8:-", &:read)

        begin
         + Encoding.default_external = Encoding::UTF_8
         + Encoding.default_internal = Encoding::UTF_8
            spec = eval code, binding, file
    ...
```

```
16      end
17
```

Listing A.9 Making the locale encoding deterministic we avoid the build process to break because of unrecognized characters. Thus solving unreproducibility issues linked to that.

```
1       module Gem
2           RUBYGEMS_DIR = __dir__
3           ...
4
5           @default_source_date_epoch = nil
6
7           @discover_gems_on_require = true
8
9            + File.umask(0o022)
10
```

Listing A.10 Forcing the permission umask for the running process avoids unreproducibility issues caused by the execution of scripts with the user permission umask.

A.2.2 PyPI Patches

We patched the `pip` package manager to address reproducibility issues caused by umask in both archive metadata and dynamic metadata. The archive metadata patch (Listing A.11) consists of resetting the file metadata contained in the archive. We did that by adding a submodule intercepting the created archive and repackaging it. The patch works for both zip and tar archives, addressing both source and wheel PyPI distributions, and it also addresses issues caused by timestamps in the archive metadata, already solved through infrastructure configuration. The dynamic metadata patch for umask (Listing A.12) specifies the umask value used throughout the build process to ensure that all file manipulations have that specific umask value. Another patch applied into the `pip` code targets locale reproducibility issues. This patch (Listing A.12) sets the locale to a fixed value.

The frontend - backend architecture of the PyPI distribution model allows the use of the `pip` frontend to address some reproducibility issues. However, the build backend operations can compromise the build reproducibility offered by the frontend. We patched the PyPI `setuptools` and the `poetry` build backends in different ways. Our first patch (Listing A.13) set a fixed time value for `setuptools` tag date option. This is quite a common issue, also

affecting the setuptools package[1]. Our second patch (Listing A.14) set a fixed locale value during the package build in the `poetry` backend.

```
def reset_zip(zipinfo):
    zipinfo.date_time = (1980, 1, 1, 0, 0, 0)
    zipinfo.external_attr = 0o100644 << 16
    return zipinfo

def reset_tar(tarinfo, timestamp=503013770):
        if tarinfo.isdir():
            tarinfo.mode = 0o755
        else:
            tarinfo.mode = 0o644
        tarinfo.uid = tarinfo.gid = 0
        tarinfo.uname = tarinfo.gname = "root"
        tarinfo.mtime = timestamp
        return tarinfo
```

Listing A.11 The fix inserted a new submodule (i.e. `_archive.py`) in the `build` module for the repackaging of the created distributable archives. Since PyPI distributes both `zip` and `tar` archives for wheel and source distribution respectively we provided different resetting methods for both of them. For space reasons we display only the functions dealing with the resetting of metadata.

```
    def main(args: Optional[List[str]] = None) -> int:
    ...

    try:
         locale.setlocale(locale.LC_ALL, "")
         locale.setlocale(0, "")
         os.umask(0o022)
```

Listing A.12 reproducibility issues caused by locale and umask in dynamic metadata are solved by setting these properties to fixed values in the pip frontend

```
def tags(self) -> str:
    version = ''
    if self.tag_build:
        version += self.tag_build
    if self.tag_date:
         - version += time.strftime("%Y%m%d")
```

[1]https://github.com/pypa/setuptools/blob/1ed759173983656734c3606e9c97a348895e5e0c/setup.cfg#L160

```
+ version += "omitted to achieve reproducibility"
    return version
```

Listing A.13 The `setuptools` backend allows developers to characterize the version of a PyPI package based on the current date. When the `tag_date` field is set to 1 the current date is appended to the package version. This feature compromises the reproducibility of a package.

```
    try:
- pth_file = self._env.site_packages.write_text(pth_file, content,
encoding=locale.getpreferredencoding())
- pth_file = self._env.site_packages.write_text(pth_file, content,
encoding="UTF-8"
        self._debug(
            f"  - Adding <c2>{pth_file.name}</c2> to <b>{pth_file.parent}</b> for"
            f" {self._poetry.file.path.parent}"
        )
        return [pth_file]
```

Listing A.14 insertcaption here.

```
import os
from os.path import join, dirname

from setuptools import setup, find_packages

from version import get_version

 os.umask(0o022)

with open(join(dirname(__file__), 'README.rst'), encoding="utf-8") as f:
    README = f.read()
```

Listing A.15 Example of PyPI package where the developer tried to solve the reproducibility issues caused by the umask values in archive metadata by the use of the setup.py file.

A.3 Tables

Table A.1 `reprotest` variations, along with a brief description, used to identify reproducibility issues in packaging and compilation build processes.

Variation	Brief Description
Environment	Environment variables may affect the build used inside code and configuration files.
Fileordering	The order in which files are evaluated or inserted inside of a package.
Kernel	Kernel properties, like the OS platform, may influence the build process or be inserted inside of code.
Locales	The used encoding may affect the build process during I/O operations.
Exec_path	Changes in the `PATH` variable may affect the build process.
Time	The current time can be used inside of code or placed in metadata during the build process.
Timezone	As for time, the timezone can influence the current time, hence the build process.
Umask	Permission masks are applied to files inserted inside archives or during the build process to execute external scripts.

Table A.2 Each ecosystem points to a package manager. The package manager can be used to either package or build a project, based on the parameters passed through the CLI.

Ecosystem	Package Manager	Build Cmd. Pack.	Build Cmd Compilation
Cargo	`cargo`	`cargo package`	`cargo build`
Go	`go`	-	`go build`
Maven	`mvn`	`mvn package`	`mvn compile`
NPM	`npm`	`npm pack`	
PyPI	`pip`	`pip wheel`	
RubyGems	`gem`	`gem build`	