Journal of Software: Evolution and Process

WILEY

Software: Evolution and Process

RESEARCH ARTICLE - EMPIRICAL OPEN ACCESS

Security Assessment of Private Package Repositories: An Experience on Acc-Py at CERN

Michele Lizzit[1] | Francesco Pinzauti[2] | Marino Miculan[1] | Vincenzo Riccio[1]

[1]University of Udine, Udine, Italy | [2]CERN, Meyrin, Switzerland

Correspondence: Michele Lizzit (michele.lizzit@uniud.it)

Received: 19 January 2026 | **Revised:** 30 June 2026 | **Accepted:** 17 July 2026

ABSTRACT

The introduction of package managers had a huge impact on software development, as they facilitate dependency management and the access to reusable code components. However, reliance on centralized repositories introduces security risks, as they are increasingly exploited in supply chain attacks. To reduce these risks, many companies rely on private package managers and repositories. Although public package repositories have been extensively studied, private ones remain underexplored. This paper presents a security comparison of private package repositories versus their public counterparts, through our experience on PyPI and CERN's Acc-Py. We perform a comprehensive security assessment of both repositories, complemented by discussions with the CERN development team. Using open-source static analyzers, we find that Acc-Py hosts packages with fewer potential security issues than those on the broad PyPI ecosystem. However, it remains susceptible to dependency confusion attacks due to namespace collisions with PyPI. Our dynamic analysis technique identifies telemetry collection as a privacy-monitoring trade-off. Our study provides valuable insights for analyzing and strengthening the security of private repositories, addressing their unique security challenges and attack surfaces.

1 | Introduction

Package managers are indispensable for modern software development, as they facilitate the installation and updating of software components [1]. By automating package discovery, installation, and management through centralized repositories (*package indexes*), they enable developers to integrate third-party libraries and share code across projects [2, 3]. Package managers and their corresponding repositories, such as `pip`/PyPI [4] and `npm`/npm registry [5], significantly accelerate development by streamlining efficient dependency management and promoting code reuse [6].

The openness and widespread adoption of package repositories make them extremely valuable but also prime targets for malicious actors, creating substantial security vulnerabilities [7]. Software supply chain attacks, which inject harmful code into libraries or manipulate distribution channels, pose an increasing threat to developers and organizations [8–12]. A striking example is the December 2022 dependency confusion attack against PyTorch, where attackers exploited PyPI to prioritize a malicious `torchtriton` package over the legitimate internal version, enabling data exfiltration via DNS [13]. Such incidents underscore the severe security implications that drive researchers and industry practitioners to explore mitigation, detection, and prevention strategies [14–17], engaging with administrators and contributors to gain a deeper understanding of real-world security challenges [18].

To enhance software supply chain security and maintain control over software distribution, organizations increasingly rely on *private* package managers and repositories [19, 20]. However, despite extensive research on public package repositories, the security properties of private package managers remain largely unexplored. Existing studies primarily focus on attacker behavior in open ecosystems, leaving open questions about whether

curated, access-controlled repositories meaningfully mitigate supply chain risks, and which vulnerabilities persist despite governance mechanisms.

In this paper, we conduct a comprehensive security assessment of *Acc-Py*, the private package manager designed by the European Organization for Nuclear Research (CERN) to enforce governance over Python dependencies [21]. Acc-Py plays a vital role in CERN's complex computing infrastructure, supporting critical software for data analysis, experiment control, and large-scale simulations [22]. Acc-Py is representative of private package repositories deployed in large organizations, sharing threat models with enterprise and research infrastructures. Like other private repositories, it acts as a controlled gateway between external third-party dependencies and internal software, implementing policies to mitigate risks such as supply chain attacks, dependency confusion, and unauthorized software distribution. We present our security assessment of Acc-Py by exploring potential attack vectors that have been previously overlooked, conducting static, metadata, and dynamic analyses to comprehensively evaluate its security posture. To establish a comparative baseline, we also apply these analyses to PyPI. In addition, we engage in collaborative discussions with CERN engineers to gain insight into their vulnerability assessment and remediation processes. Our study addresses the following research questions:

RQ1. (Comparative static analysis) *What are the quantitative and qualitative differences in the security issues identified through static analysis between packages in PyPI and those in Acc-Py?*

RQ2. (Dependency confusion vulnerabilities) *To what degree is Acc-Py susceptible to dependency confusion vulnerabilities?*

RQ3. (Comparative dynamic analysis) *How do the security issues revealed during dynamic analysis of package installations in Acc-Py compare to those observed in PyPI?*

RQ4. (Expert evaluation and mitigation strategies) *How do CERN engineers evaluate the identified security issues within Acc-Py, and what mitigation strategies do they propose?*

Our analysis revealed that Acc-Py has a comparatively robust security posture relative to PyPI, evidenced by a lower density of potential security issues detected by static analysis tools. Acc-Py remains vulnerable to dependency confusion attacks because most internally used package names are available to be registered on PyPI, a risk stemming from inadequate enforcement of proper configuration despite its secure framework and well-documented guidelines. Dynamic analysis found no evidence of malicious payloads during package installation, but revealed telemetry logging in Acc-Py, raising considerations regarding the balance between monitoring and user privacy. CERN engineers prioritized the dependency confusion risk over telemetry concerns and engaged in discussions regarding mitigation strategies, underscoring their dedication to continuous improvement of CERN's software ecosystem.

Our findings offer practical and transferable insights into the security of private package managers, enabling organizations to enhance the resilience of their internal software ecosystems. In particular, this paper advances the understanding of security risks in private software package repositories by making the following contributions:

- We provide the first empirical characterization of security properties in a large-scale private Python package repository, addressing a gap in existing software supply chain research that has predominantly focused on public ecosystems.
- We propose and validate a comparative assessment methodology that combines static analysis, metadata inspection, dynamic installation monitoring, and expert review, enabling systematic comparison between private and public package repositories.
- We demonstrate that repository governance and curation substantially reduce the prevalence of potential security issues, but do not inherently prevent dependency confusion attacks, which primarily arise from configuration and naming practices rather than implementation flaws.
- We show how expert-driven interpretation of automated analysis results refines risk assessment and informs effective mitigation strategies, highlighting the limitations of tool-only evaluations in private repository environments.

2 | Acc-Py: CERN's Package Management

CERN, a global leader in particle physics research, operates experiments generating massive datasets that necessitate sophisticated software infrastructure [23]. Python plays a pivotal role in CERN's data analysis and control systems. Due to these specific requirements, CERN develops ad hoc software packages or modifies existing ones, as public repositories often lack the necessary functionalities. To ensure strict control, security, and governance over package management, CERN employs *Acc-Py*, a curated software distribution framework, rather than relying solely on PyPI [21, 22].

Acc-Py stores, manages, and distributes Acc-Py packages while providing controlled access to external dependencies via PyPI. It comprises: a PEP-503 compliant repository for CERN-specific packages; client-side tools for controlled access; and a server that intermediates between internal clients and external repositories, enforcing CERN's security and governance policies. This internal ecosystem optimizes integration with CERN's infrastructure, ensures stability through rigorous version control, mitigates supply chain attacks, and enforces compliance. Acc-Py prioritizes *internal* resources to prevent dependency confusion when accessing external packages [11, 24, 25]. A continuous deployment pipeline subjects each package to automated validation, build, and release, including static security and compatibility checks, as well as functional and integration tests. Acc-Py also utilizes a centralized logging system (ELK stack) for real-time monitoring, anomaly detection, and unauthorized modification prevention [26, 27].

Given Acc-Py's design for enhanced Python package security over PyPI, this study assesses its resilience against real-world

threats and attack vectors, contributing to the understanding of supply chain security in high-stakes environments.

3 | Benchmark Dataset

We collected a comprehensive dataset of packages from both Acc-Py and PyPI in order to perform a comparative security assessment of the private repository under study and its public counterpart. To enable a comprehensive and meaningful comparison, we constructed two distinct subsets of PyPI packages. **PyPI Top 10k** includes the 10,000 most downloaded packages from PyPI [28], representing those most widely adopted in practice. **PyPI 10%** is a random sample of approximately 10% of all packages available from the official PyPI index [29], that captures the diversity and general security posture of the ecosystem.

This dual dataset design for PyPI allows us to balance representativeness and prominence: The random sample provides a baseline for common practices across the ecosystem, whereas the popularity-based subset targets packages with higher exposure and risk, which are more likely to be used in production systems and targeted by attackers. At the same time, popular packages often benefit from better maintenance practices, higher code quality, and more frequent updates, due to their larger user base and greater community scrutiny. For **Acc-Py**, we obtained all 558 available packages from the index available internally at CERN.

We used `pip` for downloading and installing packages, as it is the only package manager officially supported at CERN and the most susceptible installation method to malware, according to Guo et al. [30]. We developed a script to retrieve the latest version of each package from PyPI and Acc-Py, fetching a list of package names from the endpoint and downloading the corresponding packages. Table 1 reports the information about the packages downloaded from Acc-Py and PyPI, respectively. The average lines of code per package are comparable, indicating structural similarity between the two datasets. We also report the median size of the packages. Acc-Py packages are larger than those in the random PyPI sample and smaller than the most popular PyPI packages, mirroring the pattern observed for lines of code. We also include data on the number of active users (PyPI's numbers refer to the entire repository).

TABLE 1 | Acc-Py vs. PyPI: dataset statistics.

Statistic	Acc-Py	PyPI top10k	PyPI 10%
Total number of packages	558	10,000	55,813
Total lines of code	931,005	113,046,417	86,636,608
Average lines of code per package	1668.47	11,304.64	1552.26
Median package size (KB)	19.9	43.0	13.3
Index accessibility	Private	Public	Public
Active users	1000	910,515	910,515

4 | Static Analysis

A systematic comparison of potential security issues detected in PyPI and Acc-Py by static analysis tools provides valuable insights into the security landscape of CERN's Python ecosystem. Although static analyzers do not confirm actual vulnerabilities, the volume and nature of the issues they flag serve as useful indicators of code quality, developer burden, and recurring security anti-patterns, as demonstrated in the literature [31]. Moreover, they effectively reflect the triaging workload for engineers, which is an important operational concern. By comparing the two considered environments, we investigate whether the controlled nature of Acc-Py results in distinct security anti-patterns compared with the more diverse ecosystem of PyPI. Our comparison helps highlight whether particular types of potential security issues are more prevalent in the broader ecosystem or specific to enterprise projects, offering actionable guidance for CERN engineers in prioritizing their security reviews and tooling efforts. Therefore, we seek to answer the following research question:

RQ1. (Comparative static analysis) *What are the quantitative and qualitative differences in the security issues identified through static analysis between packages in PyPI and those in Acc-Py?*

4.1 | Experimental Design for RQ1

We employ two static analyzers, i.e., Bandit and GuardDog. These tools were selected with CERN engineers based on the following inclusion criteria:

- They are *open-source*, ensuring accessibility;
- They target the Python programming language;
- They exhibit diversity in their analysis approaches;
- They have been widely used in the literature, demonstrating their reliability [17, 32–35];
- They are regularly updated, ensuring relevance;
- They have been suggested by CERN engineers.

Bandit[1] is designed to identify security vulnerabilities in Python code. It relies on the analysis of abstract syntax trees (ASTs) to detect coding patterns that may introduce security risks. Bandit systematically inspects the code by parsing it into its AST representation and then detecting patterns that may pose security risks. These patterns are based on known insecure coding patterns, which are typically derived from security best practices, common weaknesses (e.g., CWE), and real-world security incidents.

GuardDog[2] is developed for static analysis of source code and packaged applications. It focuses on security vulnerabilities and compliance issues by applying a set of heuristics, inspired by the literature, on package source code through Semgrep rules [36] and on package metadata using Python or Javascript heuristics.

We acknowledge that static analyzers typically produce a high rate of false positives. However, because this rate is expected to

remain roughly constant across the repositories, we do not analyze it in detail at this stage. Instead, the volume and types of findings serve as a proxy for the triaging burden faced by organizations relying on these tools, which is within the scope of this study. In RQ4, we refine our analysis focusing on Acc-Py findings, that have been manually reviewed in collaboration with CERN engineers.

We apply both static analyzers to each package in our dataset to compare the number and types of findings (i.e., the potential security issues identified by the tools) across the two package repositories. The methodology and detailed results are presented in the following subsections.

4.2 | Results of RQ1

4.2.1 | Bandit

Bandit detects security-related code smells. We report occurrences per million lines of code **(Occ./MLOC)**, giving both the total and unique counts (the latter removes duplicate reports within a package). Normalizing by MLOC enables fair comparison across repositories of different sizes (e.g., PyPI vs. Acc-Py). In general, a lower Occ./MLOC value suggests that a repository has relatively less vulnerable code, possibly due to stronger security practices or higher code quality. Table 2 presents the frequency of each type of potential security issue detected in Acc-Py, PyPI 10%, and PyPI Top 10K. Acc-Py exhibits a higher proportion of security issues related to validation and permission issues, notably ID 20 and 25. Specifically, the *request_with_no_cert_validation* security issue is more prevalent in Acc-Py. This is likely caused by the frequent need for packages to access internal resources secured by TLS certificates signed by an internal certification authority (CA), rather than a public one. Developers may opt to disable certificate verification for convenience, rather than embedding internal CA certificates within the package, despite the inherent security risks. Furthermore, Acc-Py shows a higher occurrence rate of security issues associated with bad programming patterns, such as ID 1 and 4. Although these patterns are not necessarily indicative of direct security flaws, they can contribute to vulnerabilities. These two patterns occur with a high rate, but exhibit a much lower rate of unique Occ./MLOC, suggesting that few packages rely heavily on `assert` and `try-except-pass` patterns, therefore significantly increasing the total rate of Occ./MLOC. Overall, Acc-Py exhibits a higher total number of potential security issues compared with the most popular PyPI packages, but a slightly lower number of unique findings than the PyPI 10% dataset.

Figure 1 reports the results of a further analysis on the relevance of the static analyzer's findings, by grouping them by severity level as assigned by Bandit. When compared against a random sample of PyPI packages (i.e., PyPI 10%), Acc-Py demonstrates a reduced ratio of high- and medium-severity Occ./MLOC, suggesting a potentially stronger security posture. Conversely, when compared against the same dataset, Acc-Py exhibits a substantial increase in low-severity Occ./MLOC, which may reflect a higher incidence of false positives, typical of static analyzers. This trend reverses when the comparison is limited to the most popular PyPI packages (i.e., PyPI Top 10k), suggesting that widely adopted open-source projects tend to exhibit higher code quality and stronger security hygiene. This may reflect the benefits of broader community scrutiny, frequent maintenance, and stricter contribution practices typically associated with high-profile projects.

4.2.2 | GuardDog

GuardDog aims to identify potentially malicious packages by applying a set of heuristics to both the package source code and its metadata. In our analysis, we consider only GuardDog heuristics that are applicable to both Acc-Py and PyPI. Specifically, we exclude the following metadata-based checks: *repository_integrity_mismatch*, *unclaimed_maintainer_email_domain*, and *potentially_compromised_ email_domain*. In fact, these checks focus on maintainer profiles in the package index, rather than the package content. Because such metadata-based checks are either unavailable or irrelevant in most private repositories (such as Acc-Py), we omit these heuristics from our evaluation to ensure a fair and consistent comparison across repositories.

Table 3 presents the results of the analysis performed by GuardDog on Acc-Py and PyPI. Because some GuardDog heuristics (e.g., *single_python_file*) target package metadata rather than source code, normalizing their occurrences by MLOC would be inappropriate. Therefore, we report such metrics relative to the number of packages only (**Occ./package**), while reporting their values of Occ./MLOC as not applicable (*N/A*). We also note that, across all three sets of packages, code-related findings are never detected more than once per package. Therefore, we do not make a distinction between findings per package and unique findings per package, as we previously did for *Bandit*.

As for PyPI, the most prevalent heuristic, *empty_information*, was identified in 9788 packages within the PyPI 10% dataset, indicating a widespread absence of descriptive metadata, which may suggest poorly maintained software. *single_python_file* (6 769 occurrences in PyPI 10%) further highlights the presence of numerous packages consisting of a single script, a common trait in minimalistic or potentially obfuscated implementations. The detection of *typosquatting* in 2175 cases underscores a substantial risk where package names closely mimic those of well-known libraries, increasing the likelihood of unintentional installations by users. Additionally, *bundled_binary* (1043 occurrences in PyPI 10%) suggests a significant number of packages include compiled binaries, which may introduce security concerns due to hidden malicious payloads. Security-oriented heuristics, such as *shady-links* (423 occurrences in PyPI 10%), *cmd-overwrite* (288 occurrences), and *code-execution* (286 occurrences), further demonstrate the presence of packages exhibiting behaviors associated with unauthorized command execution, suspicious external links, and install-time script modifications.

Instead, across all Acc-Py packages GuardDog detects only five security issues. Among these, the most frequently triggered heuristic is *shady-links*, which generates findings in three distinct packages. This heuristic flags URLs within packages that point to domains with suspicious extensions, including URL shorteners that can obscure the true destination

TABLE 2 | RQ1: occurrences of potential security issues found by Bandit.

Id	Security issue	PyPI 10% Occ./ MLOC tot (unique)	PyPI Top 10k Occ./ MLOC tot (unique)	Acc-Py Occ./MLOC tot (unique)
1	assert_used	4170.5 (93.2)	5374.5 (32.3)	9062.2 (218.0)
2	blacklist	582.4 (106.2)	242.3 (27.1)	546.7 (68.7)
3	request_without_timeout	176.5 (45.4)	41.0 (6.0)	84.9 (15.0)
4	try_except_pass	141.5 (34.7)	86.8 (12.6)	227.7 (37.6)
5	subprocess_without_shell_true	127.3 (36.1)	52.3 (11.0)	105.3 (17.2)
6	hardcoded_sql_expressions	80.0 (11.0)	32.0 (3.2)	16.1 (3.2)
7	start_process_with_partial_path	92.1 (24.0)	19.8 (5.4)	12.9 (9.7)
8	hardcoded_password_string	87.8 (23.1)	71.2 (9.4)	25.8 (15.0)
9	start_process_with_a_shell	79.4 (15.0)	7.8 (2.1)	33.3 (11.8)
10	popen_with_shell_true	40.1 (13.2)	8.7 (3.0)	9.7 (7.5)
11	hashlib	33.0 (15.5)	19.6 (6.7)	6.4 (2.1)
12	hardcoded_tmp_directory	31.3 (9.1)	13.5 (3.1)	17.2 (7.5)
13	exec_used	24.3 (9.0)	11.2 (4.4)	7.5 (5.4)
14	hardcoded_password_funcarg	23.4 (5.7)	22.7 (2.3)	4.3 (2.1)
15	hardcoded_password_default	13.5 (5.9)	8.2 (2.5)	1.1 (1.1)
16	hardcoded_bind_all_interfaces	31.7 (7.4)	32.2 (2.2)	4.3 (4.3)
17	yaml_load	12.5 (6.5)	3.0 (1.3)	6.4 (3.2)
18	django_mark_safe	4.7 (1.6)	2.7 (0.6)	0.0 (0.0)
19	try_except_continue	15.9 (5.4)	3.9 (1.9)	6.4 (1.1)
20	request_with_no_cert_validation	7.3 (2.2)	1.2 (0.3)	26.9 (5.4)
21	jinja2_autoescape_false	6.0 (4.0)	2.2 (1.4)	5.4 (4.3)
22	tarfile_unsafe_members	6.0 (3.3)	3.5 (1.6)	1.1 (1.1)
23	any_other_func_with_shell_true	7.3 (1.7)	3.3 (1.0)	0.0 (0.0)
24	start_process_with_no_shell	3.2 (2.4)	1.4 (0.9)	4.3 (3.2)
25	set_bad_file_permissions	4.5 (1.7)	1.7 (0.7)	18.3 (4.3)
26	paramiko_calls	2.0 (0.9)	0.3 (0.2)	0.0 (0.0)
27	ssh_no_host_key_verification	1.9 (1.3)	0.3 (0.2)	1.1 (1.1)
28	django_extra_used	0.3 (0.2)	0.5 (0.1)	0.0 (0.0)
29	flask_debug_true	0.9 (0.8)	0.0 (0.0)	0.0 (0.0)
30	ssl_with_no_version	0.7 (0.5)	0.3 (0.3)	0.0 (0.0)
31	use_of_mako_templates	0.7 (0.4)	0.3 (0.1)	0.0 (0.0)
32	django_rawsql_used	0.1 (0.0)	0.1 (0.0)	0.0 (0.0)
33	weak_cryptographic_key	0.3 (0.1)	0.2 (0.0)	0.0 (0.0)
34	ssl_with_bad_version	0.1 (0.0)	0.1 (0.0)	0.0 (0.0)
35	linux_commands_wildcard_inj	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
36	snmp_insecure_version_check	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
37	logging_config_insecure_listen	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
	Total	5809.5 (487.4)	6068.9 (144.2)	10,235.2 (450.1)

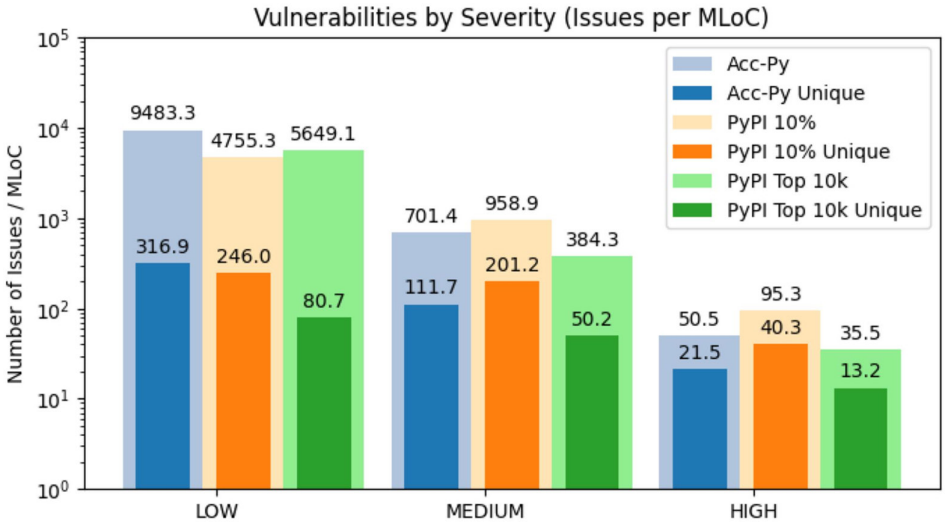

FIGURE 1 | RQ1: average number of occurrences of potential security issues per MLOC found by Bandit on Acc-Py and PyPI.

TABLE 3 | Occurrences of potential security issues found by GuardDog.

Id	Security issue	PyPI 10% Occ./ MLOC (Occ./pkg)	PyPI top10k Occ./ MLOC (Occ./pkg)	Acc-Py Occ./MLOC (Occ./pkg)
1	empty_information	N/A (.188)	N/A (.051)	N/A (–)
2	single_python_file	N/A (.129)	N/A (.117)	N/A (–)
3	typosquatting	N/A (.040)	N/A (.018)	N/A (–)
4	shady-links	15.70 (.026)	3.11 (.036)	3.22 (.006)
5	obfuscation	12.93 (.021)	4.18 (.049)	—
6	bundled_binary	N/A (.019)	N/A (.023)	N/A (–)
7	release_zero	N/A (.012)	N/A (.000)	N/A (–)
8	code-execution	6.74 (.011)	0.76 (.009)	1.07 (.002)
9	deceptive_author	N/A (.009)	N/A (.003)	N/A (–)
10	clipboard-access	3.30 (.005)	0.54 (.006)	1.07 (.002)
11	cmd-overwrite	3.14 (.005)	0.15 (.002)	1.07 (.002)
12	bidirectional-characters	2.20 (.004)	0.19 (.002)	—
13	dll-hijacking	1.29 (.002)	0.17 (.002)	—
14	exec-base64	0.75 (.001)	0.11 (.001)	—
15	exfiltrate-sensitive-data	0.58 (.001)	0.23 (.003)	1.07 (.002)
16	silent-process-execution	0.28 (.000)	0.03 (.000)	—
17	download-executable	0.10 (.000)	—	—
	Total	47.01 (0.47)	9.48 (0.32)	7.52 (0.01)

and services commonly associated with data exfiltration, such as file-sharing platforms and messaging APIs. We classify this type of finding as a supply chain integrity risk. Other heuristics, such as *exfiltrate-sensitive-data* and *clipboard-access*, each observed once, suggest that certain packages attempt to read or exfiltrate sensitive user information, posing a risk to data confidentiality. Finally, *code-execution* and *cmd-overwrite* were each identified once, highlighting instances where packages execute OS commands or modify installation behavior within the *setup.py* file. As shown in Figure 2, the key difference between the two repositories identified by GuardDog is that PyPI presents a broader risk landscape with occurrences across multiple heuristics. In contrast, Acc-Py appears more secure, exhibiting on average fewer potential security issues per package when compared with the general PyPi index; this advantage however shrinks substantially when compared only against the most popular PyPi packages and does not hold true when compared only on specific issue categories (i.e.

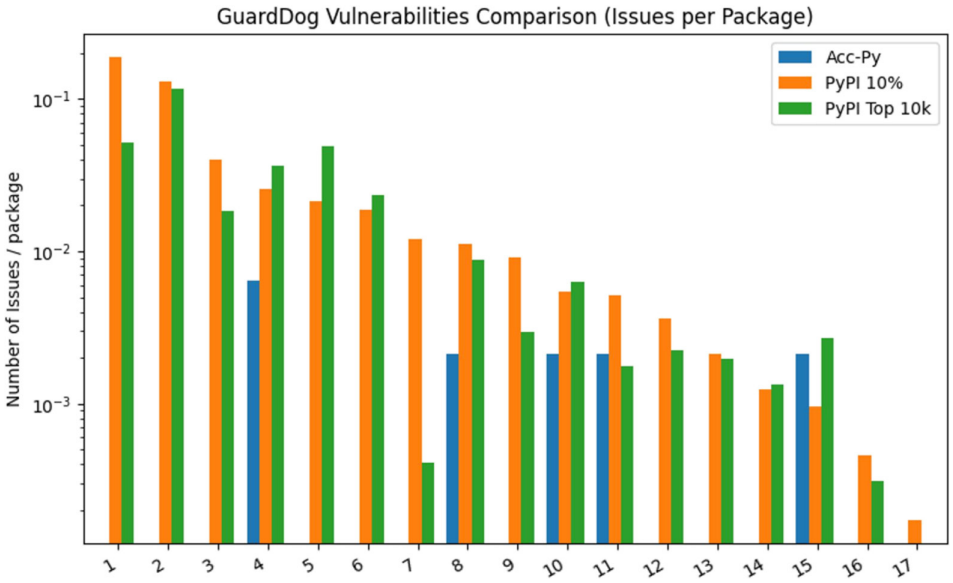

FIGURE 2 | Average number of occurrences of potential security issues per package found by GuardDog on Acc-Py and PyPi.

shady-link, code-execution, clipboard-access, cmd-overwrite, and exfiltrate-sensitive-data). Still, the analysis suggests that the management of URLs within packages requires careful scrutiny to determine if they are true positives.

4.2.3 | Comparative Evaluation

Our analysis reveals a nuanced picture of the security posture of Acc-Py compared with PyPI. GuardDog reports fewer potential threats in Acc-Py, likely due in part to the absence of publicly exposed metadata and a more controlled publishing environment. Bandit highlights a higher density of code-related findings in Acc-Py than in the most popular PyPI packages, but less than the random sample of PyPI packages. These results suggest that, although private repositories may benefit from tighter access control and reduced exposure to metadata-based risks, they do not necessarily exhibit superior code-level security.

Overall, **GuardDog** reports fewer potential security issues in Acc-Py compared with both PyPI samples, suggesting a lower exposure to metadata- and behavior-based code patterns commonly associated with malicious packages. This may reflect the more controlled and curated nature of private repositories. This contrasts with the findings from **Bandit**, where Acc-Py exhibited a higher density of code-level findings than the most popular PyPI packages. This divergence highlights the complementary nature of the two tools and the need to interpret security posture across multiple dimensions: code quality, ecosystem exposure, and metadata transparency.

> **RQ1:** *Static analysis reveals that Acc-Py generally exhibits a more secure posture, reporting fewer potential security issues per MLOC than the broader PyPI ecosystem, but still more than the most popular PyPI packages. Acc-Py still shows relatively high occurrences of specific findings, particularly those related to supply chain integrity, validation, and permission problems. This highlights the importance of adopting complementary security practices and tools, especially for private ecosystems that may lack the scale and community-driven quality control seen in popular open-source projects.*

> **Key Insight 1:** *Comparing a private repository with its public counterpart through multiple and diverse static analyzers provides valuable insights into the security posture of the enterprise repository, helping to assess its strengths and highlight areas for improvement.*

5 | Dependency Confusion

Dependency confusion is a supply chain attack that exploits package managers' resolution policies. In this attack, an adversary publishes a malicious package with the same name as an internally maintained one, but on a public repository. If the package manager prioritizes public over private packages, the malicious package can be inadvertently installed. This mechanism may lead to the unintended execution of malicious code, enabling actions such as data exfiltration. Therefore, introducing packages into public indexes with names identical to those of internal packages can potentially lead to security breaches, particularly for organizations that maintain private repositories alongside public counterparts. We focus on dependency confusion as a key contribution of this study, due to its high risk in hybrid ecosystems with private repositories and public counterparts. This threat has already been recognized by CERN, highlighting the practical importance of analyzing namespace collisions and their security implications in real-world settings [37]. In this section, we explore how dependency confusion vulnerabilities impact Acc-Py. We answer the following question:

RQ2. (Dependency confusion vulnerabilities) *To what degree is Acc-Py susceptible to dependency confusion vulnerabilities?*

5.1 | Quantitative Analysis for RQ2

We quantitatively evaluated the extent to which packages in Acc-Py are susceptible to dependency confusion attacks. Specifically, we analyzed how many package names in Acc-Py have a corresponding package or placeholder package on PyPI. The presence of an Acc-Py package with the same name on PyPI increases the risk of unintentional dependency resolution from an external source, making it a potential vector for dependency confusion attacks.

Our analysis found that only 2.7% of Acc-Py packages have a counterpart on PyPI. This suggests that the attack surface for dependency confusion attacks is large, as most package names used internally are available to be registered on PyPI and could be claimed by attackers. Of these PyPI packages, four of them are a shallow clone of an Acc-Py package; eight of them are the original package version, which was forked and published as an Acc-Py package; four packages are mirrors of an internal Acc-Py package, that was pushed to PyPI in order to make it publicly available. We also note that the Acc-Py official documentation page of CERN's Accelerators and Beam Physics Computing (ABP) group explicitly recommend using placeholders [37], where they are referred to as *shallow clones*, and provide detailed instructions for CERN developers on how to build and publish them. Despite this, our results confirm that this practice is not thoroughly followed within the organization.

5.2 | Real-World Validation for RQ2

To evaluate the impact of dependency confusion on Acc-Py, we uploaded a shallow clone on pypi.org, with the same name as one of the packages on Acc-Py but with a higher version number. The target package was chosen randomly among the least popular packages, as to create the least disruption. In the shallow clone, we introduced telemetry to evaluate whether our packages were being installed. Because package managers often prioritize the latest available version, this approach allows us to test when hosts using CERN's Acc-Py dependency resolution mechanisms prioritize the external package over the internal one. To measure the impact of our simulated attack, we introduced a benign modification to the SETUP.PY file of our package as follows.

```
...
def _post_install():
    p = base64.b64encode(socket.getfqdn().encode())
    url = f'https://our-domain.tld/image.png?{p.decode()}'
    with urllib.request.urlopen(url) as response:
        data = response.read()
...
```

Specifically, upon package installation, the modified package initiates an HTTP request to our backend server. This request includes the hostname of the machine. The backend, implemented as a Flask application, logs the source IP address of the request and the hostname of the machine.

To assess if dependency confusion is a viable attack vector within the Acc-Py ecosystem, we monitored all incoming requests to determine if any originated from CERN's internal network. A successful request from within CERN would confirm that Acc-Py was susceptible to dependency confusion, revealing a potential security risk. Before deploying our validation experiment, we ensured compliance with applicable legal and ethical guidelines. Additionally, the package was promptly removed as soon as the analysis was over. No sensitive data beyond minimal identifiers (i.e., encoded hostname and IP address) were collected. Afterwards, we followed the ISO/IEC 29147:2018 guidelines on responsible disclosure of vulnerabilities [38], and all findings were immediately shared with the relevant security team at CERN, who took appropriate action to mitigate the risk. Prior to deployment, we discussed our validation experiment with the Acc-Py team. We did not discuss the validation experiment or warned all CERN users in advance as this would have impacted the results. In coordination with CERN engineers, we estimated the impact of the test to be minimal and not a concern.

Our analysis, which ran over a span of 96 h, recorded a total of six installation requests, all of which originated from hosts within the CERN General Purpose Network (GPN). This demonstrates that the Acc-Py repository is vulnerable to dependency confusion attacks.

5.3 | Qualitative Analysis for RQ2

We evaluated the Acc-Py installation rules of the Acc-Py framework [37], verifying that its default installation is not susceptible to dependency confusion. This is because this framework merges public and private repositories on the server side, preventing the installer client from making the selection. The merging rules enforce a strict preference for in-house packages over public ones, regardless of version [22]. This is consistent with best practices for mitigating dependency confusion attacks [39].

However, dependency confusion is still possible, as the default installation rules of Acc-Py are not automatically enforced on the internal GitLab pipelines, the CERN network, or CERN laptops. Users can freely configure their Python installation to use the Acc-Py repository without installing the entire Acc-Py framework and its default security settings. Nevertheless, Acc-Py is enabled by default in Virtual Private Clouds (VPCs) and in the Technical Network, where all critical operations take place. Additionally, the Technical Network is isolated from the Internet, inherently protecting it from this type of attacks.

> **RQ2:** *Acc-Py was found to be vulnerable to dependency confusion attacks. This vulnerability does not stem from the structure of the framework, which we found to be secure and with well-documented usage guidelines, but rather from the lack of enforcement of proper configuration at the organizational level.*

> **Key Insight 2:** *When maintaining private repositories with public counterparts, conflicting namespaces can create vulnerabilities, highlighting the need for organizational-level controls to enforce proper configuration and mitigate security risks such as dependency confusion.*

6 | Dynamic Analysis

To uncover dynamic malicious behaviors that evade static analysis, we analyze package installation processes, focusing on actions like retrieving external payloads or connecting to Command and Control (C2) servers. In RQ3, we measure how often these behaviors appear in Acc-Py compared with PyPI:

RQ3. (Comparative dynamic analysis) *How do the security issues revealed during dynamic analysis of package installations in Acc-Py compare to those observed in PyPI?*

6.1 | Experimental Design for RQ3

We perform dynamic, install-time analysis to detect malicious behaviors (e.g., payload downloads or C2 connections) because it is prevalent and critical in the PyPI ecosystem, as reported by Guo et al. [30]. We use a reproducible Docker-based environment, based on the official Python 3.11 Docker image, simulating a typical Python user setup. It includes tcpdump and common build dependencies (`setuptools`, `wheel`). Our procedure has two phases.

Dependency resolution and preparation: We resolve dependencies with `pipgrip` and download them using `pip download` into a shared `DOWNLOAD_DIR`. Because dependency retrieval does not execute package code, traffic from this phase is excluded from the attack analysis.

Installation monitoring and analysis: Each container runs WORKER.PY, which starts `tcpdump` and installs packages offline using `pip install` with `--no-build-isolation`, `--no-index`, and `--find-links` (pointing to `DOWNLOAD_DIR`) to ensure deterministic, offline installs. Network captures for each package and its dependencies are stored in `OUTPUT_DIR`.

Packages that fail to install (e.g. due to missing external system packages or system libraries) are excluded from further analysis. Upon completion of the installation of all dependencies, the network traffic captured by `tcpdump` for each package is stored in the corresponding folder within `OUTPUT_DIR`. The process is then repeated for the main package, whose network traffic is stored in a different file, allowing us to distinguish between traffic generated by the package itself, and that generated by its dependencies. We do not perform normalization by LOC, as install-time traffic is decoupled from code size. We empirically confirm this in the results below by showing that install-time traffic does not correlate with package size. We include TCP and UDP traffic, excluding other types of network activity that may introduce background noise unrelated to the installation process. This ensures that our analysis focuses on potentially relevant communication patterns, such as data exfiltration or connections to external servers. Our methodology isolates relevant network traffic generated by each package during installation, enabling us to identify install-time attacks.

6.2 | Results of RQ3

We characterize install-time traffic using two features: (1) the number of packets and (2) average packet size, considering TCP and UDP. Figure 3 illustrates a notable difference between the two ecosystems: in general, installations from Acc-Py tend to generate fewer and smaller packets compared with those from PyPI. To assess the statistical significance of this difference, we conduct a two-sample Hotelling's T^2 test [40], comparing the joint distributions of the two traffic features across the

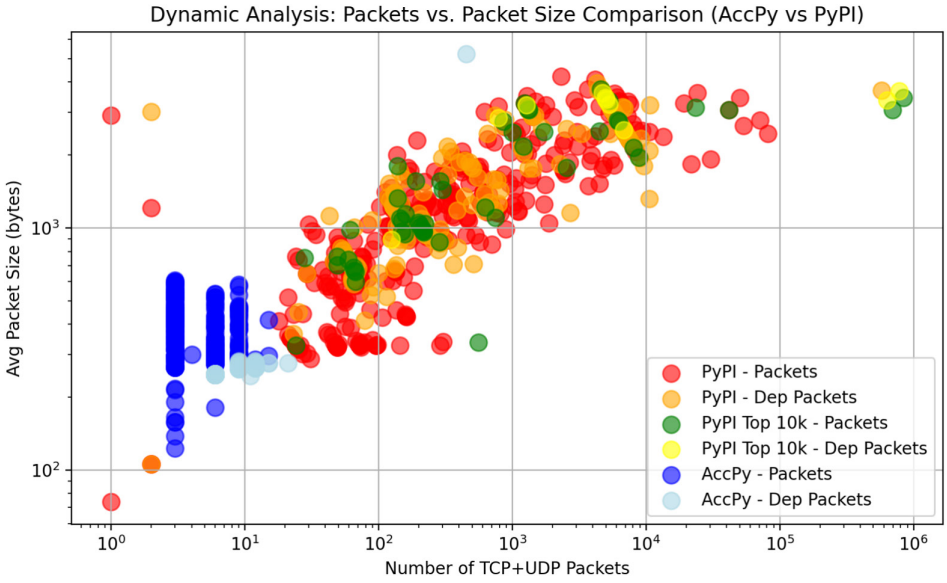

FIGURE 3 | Dynamic analysis results summary. Each dot represents the number and the average size of exchanged packets observed during the installation of a package (dark-colored dots) or its dependencies (light-colored dots).

two groups. The test yields $T^2 = 922$, with an effect size, measured as Mahalanobis distance between the group centroids of $D_m = 4.08$. This indicates a statistically significant divergence in the multivariate profiles of network traffic. The fact that Acc-Py packages generate less network traffic with smaller packets during installation suggests that they rely less on external resources, leading to fewer external dependencies being fetched during installation. With fewer packets exchanged, there is a reduced volume of outgoing network traffic, which may limit potential exposure to data interception or exploitation. However, the security implications depend on the content of these packets, i.e., the nature and sensitivity of the transmitted data.

We also confirm that install-time traffic is decoupled from package size. We do so by measuring the byte size of every package in all datasets and testing if install-time traffic depends on package size. The two are uncorrelated within every dataset, with Spearman coefficients between −0.04 and +0.01 for Acc-Py, PyPI Top 10k, and PyPI 10%. This confirms that the observed traffic does not depend on package size, and is instead driven by install-time behavior such as telemetry.

To investigate potential dynamic threats, we further analyzed network traffic generated during the installation of Acc-Py packages manually using Wireshark, focusing on packet destinations and communication protocols. Our analysis revealed no evidence of malicious payload downloads or backdoor installations for remote control. However, we identified that 326 out of 431 successfully analyzed Acc-Py packages collect *telemetry data* during installation, accounting for the previously observed traffic volume discrepancies. The collected telemetry data includes potentially sensitive user information, such as hostname and username, without explicitly informing users during the installation process. This data collection is designed to enable proactive monitoring, debugging, and auditing of system configurations, in order to support compatibility maintenance and efficient issue resolution, and is authorized under the CERN Legal Framework [41].

> **RQ3:** Dynamic analysis revealed that Acc-Py packages generate fewer network traffic during installation compared with PyPI, relying less on external resources at install time. We found no evidence of malicious payload downloads or backdoor installations in Acc-Py. However, 75% of the Acc-Py packages exfiltrate telemetry data without informing the user, raising privacy concerns.

> **Key Insight 3:** The implementation of private package repositories within organizations could imply a tighter monitoring of user activities. This practice introduces a trade-off between security and privacy, requiring a delicate balance between gaining essential operational insights and safeguarding user privacy.

7 | Expert Analysis

Gaining insight into how CERN engineers evaluate and address the security concerns identified in Acc-Py is essential to our research. This collaboration serves a dual purpose: first, to assess the relevance of our findings to CERN's security practices and their potential to enhance Acc-Py's protocols; second, to disseminate the engineers' proposed solutions, which can provide valuable guidance to other practitioners and organizations facing analogous security challenges. Therefore, we consider the following research question:

RQ4. (Expert evaluation and mitigation strategies) *How do CERN engineers evaluate the identified security issues within Acc-Py, and what mitigation strategies do they propose?*

7.1 | Review of the Static Analysis Findings

The results of our static analysis (Section 4) require manual validation to accurately assess the presence and severity of true positives. To this aim, we conducted a detailed review of the static analysis findings for Acc-Py in collaboration with the CERN team. The review process considered all findings produced by Bandit and GuardDog. To make the review tractable, warnings were first grouped by rule type and affected package, and then assessed by the engineers according to (i) whether the finding represented a true positive, (ii) whether it was expected or intentional behavior within the Acc-Py ecosystem, and (iii) its potential security impact. Through this process, we identified only one security-critical issue on a single package, which was exfiltrating sensitive data. Upon being informed, the CERN engineers promptly addressed and resolved the issue by removing the package from the Acc-Py repository, as it was a legacy artifact.

7.2 | Live Demonstration

We involved experts from CERN in a study on dependency confusion and install-time vulnerabilities, i.e., our main findings in Sections 5 and 6. We conducted a 15-min live demonstration for the expert team, followed by an open Q&A discussion to address questions and clarify key points. The audience included security specialists and software engineers responsible for Acc-Py, thus the participants had direct experience with dependency management and security policies at CERN. We intentionally avoided collecting feedback during the live demonstration to prevent participants influencing each other. In this demonstration, we highlighted the main vulnerabilities identified in RQ2 and RQ3. Specifically, we illustrated how dependency confusion attacks operate and their potential impact on Acc-Py, along with an analysis of the risks associated with telemetry collection during installation.

To facilitate a practical discussion, we presented possible mitigation strategies, whereas participants were encouraged to elaborate alternative ones and highlight any limitations in those we described.

To mitigate the risk of *dependency confusion* attacks we proposed modifying the release pipelines for Acc-Py packages by publishing a minimal placeholder package of each package on PyPI. This strategy prevents malicious actors from uploading identically named packages to PyPI, which could otherwise be mistakenly installed instead of the intended internal versions.

The placeholders will contain only the minimal necessary content, along with telemetry to monitor retrievals and gather basic user environment details. However, we also highlighted that this measure would publicly expose CERN's internal package names, which could reveal sensitive information.

To counter *typosquatting*, an attack vector which consists in registering names similar to popular packages [42], we recommended implementing package name checks at the index level. This would allow index managers to reject submissions with overly similar names or issue warnings when users attempt to install packages resembling well-known ones.

To enhance transparency and user control over *telemetry* in Acc-Py, we proposed two measures to balance necessary data collection with user satisfaction and privacy compliance. First, when executing `pip install`, users should be informed that telemetry data is collected. This notice should specify the types of data gathered and its purpose (e.g., understanding user environments, diagnosing issues, and improving performance and security). Secondly, we suggested to introduce an environment variable, allowing users to disable telemetry collection by setting it before installation; this last suggestion has already been implemented by CERN before our work was made public.

7.3 | Questionnaire

Following the presentation, we administered to CERN engineers (BE-CSS group) a questionnaire on dependency confusion attacks and telemetry, to assess both the perceived severity of the identified vulnerabilities and the effectiveness of the proposed solutions [43]. To develop our questionnaire, we performed iterative tests. In particular, one of the authors designed the questions, whereas the others reviewed and tested the questionnaire. This process was essential to identify potential issues, biases and ambiguities [44]. We refined the questionnaire until we were fully satisfied with its clarity and effectiveness, by performing a total of two rounds. In designing our questions, we ensured that the questionnaire was both clear and concise while avoiding ambiguous language. To address potential misunderstandings of technical terms, we included a reference to the definition of dependency confusion attack, besides providing practical examples during the demonstration. The questionnaire included both Likert scale questions, enabling quantitative analysis with nuanced insights, and open-ended responses, enabling qualitative feedback. To ensure respondents clearly understood their choices, we explicitly defined the two extremes of the Likert scale, i.e., scale anchors. Given the small and highly specialized expert pool, as well as the interactive nature of our demonstration, we opted not to include general knowledge verification questions. This decision was made to focus on the assessment of participants' perceptions and concerns rather than testing their knowledge.

The questionnaire was structured into three sections: *Demographic Information*, *Dependency Confusion Attack*, and *Telemetry*. Each section was presented on a separate page, and participants could revise their responses by navigating back to previous pages before submission. The answers were automatically saved, allowing participants to complete the questionnaire in multiple sessions, but a final submission was required to confirm their answers.

The **Demographic Information** section helped contextualize responses based on the roles and levels of experience of the participants. The section included three open-ended questions:

Q1. Role at CERN

Q2. Years of experience in total

Q3. Years of experience at CERN

The **Dependency Confusion Attack** section aimed to assess the participants' prior knowledge of this type of issue and their perception of its relevance and severity within the Acc-Py environment. The questions included:

Q4. Prior to this demonstration, did you have any knowledge of how dependency confusion attacks work?

Q5. Prior to this demonstration, were you aware that the Acc-Py environment could be targeted by dependency confusion attacks?

Q6. How relevant do you believe dependency confusion attacks are for Acc-Py?

Q7. How severe do you find dependency confusion attacks?

Q8. Do you think that the solution to counter dependency confusion attacks we proposed is useful?

Q9. Please share any additional thoughts or comments you have on this topic

The **Telemetry** section aimed to assess participants' prior knowledge of potential risks related to telemetry collection and their perception of its relevance and severity. The questions included:

Q10. Prior to this demonstration, were you aware of the potential privacy problems associated with telemetry?

Q11. Prior to this demonstration, were you aware of all the informations collected by Acc-Py?

Q12. How relevant do you believe issues with telemetry are for Acc-Py?

Q13. Do you think the current usage of telemetry in Acc-Py might be a problem? If so, why?

We distributed the questionnaire via CERN's internal messaging platform immediately after the presentation. The experts had one month to respond. To ensure *transparency*, we adopted the following measures: (1) we clearly communicated the duration of the questionnaire in advance, (2) we used a robust online survey tool provided by Microsoft and already in use at CERN to facilitate response analysis and prevent errors in administration, (3) we explicitly stated that all responses would be treated as confidential to encourage honest and uninhibited feedback while at the same time we obtained consent to share aggregated responses, and (4) we informed participants that

they could reach out for clarifications regarding any aspect of the questionnaire.

7.4 | Interview With Acc-Py Maintainers

To gain better insights into the design decisions that lead to the current security posture of Acc-Py, we further interviewed the core maintainers of Acc-Py at CERN. Through these interviews, we were able to assess the pipeline that Acc-Py packages follow before inclusion in the index. Specifically, packages are scanned by the `ruff` checker, hygene hooks (`trailing-whitespace`, `end-of-file-fixer`, `check-yaml`, and `add-trailing-comma`) and `mypy`; although these tools do not feature security scanning capabilities, they are used to ensure code quality standards across the Acc-Py repository. Packages that fail any of these checks are not allowed to be included in the index. It is important to note that this pipeline applies only to packages that utilize the recommended template `acc-py-gitlab-ci-templates`; packages that, due to developer choices, do not follow said template are not subject to these checks.

Security-oriented reviews of packages pushed to the index are performed manually by the Acc-Py repository core maintainers. Although CERN does not currently use Bandit or GuardDog as part of a pre-screening pipeline, these tools were evaluated in this study as a possible way to reduce the manual review burden, and their adoption is currently under evaluation at CERN.

Other limitations are known, for example, there are no safeguards in place to prevent users from uploading packages directly to the Acc-Py index, without a proper deployment pipeline. This means that a malicious user, or a malicious actor with access to user credentials is able to upload arbitrary packages to Acc-Py. The Acc-Py server however checks that the uploading user is an authenticated CERN user. Finally, the server strictly checks for metadata, anti-collision and package ownership before publishing, and these checks cannot be bypassed by a regular CERN user.

7.5 | Results of RQ4

Demographic. As reported in Table 4, six CERN engineers responded to the questionnaire within the given timeframe. They hold different levels of experience, both within and outside CERN.

TABLE 4 | General information.

ID	Time (min)	Role	Exp Tot (CERN)
1	32	Full-time engineer	<4 (<4)
2	5	Full-time engineer	<4 (<4)
3	8	Student intern	<4 (<4)
4	18	Full-time engineer	10+ (4 – 10)
5	19	Full-time engineer	10+ (4 – 10)
6	6	Full-time engineer	10+ (4 – 10)

Dependency Confusion Attack. As shown in Figure 4, most experts had prior knowledge of dependency confusion attacks but were unaware that Acc-Py could be affected by such vulnerabilities. They rated the severity of dependency confusion attacks as extremely high (8.5 out of 10 on average) and considered them quite relevant to Acc-Py (6.67 out of 10 on average). All respondents acknowledged the usefulness of our proposed solution, i.e., publishing placeholders of Acc-Py packages on PyPI. Although the solution was unanimously deemed indeed beneficial, experts raised the following concerns in the free-response section:

- Policy violation: A major concern is that the solution may violate PyPI's policies. Although similar techniques have been used by large companies[3], relying on a potentially non-compliant approach is not considered sustainable;
- Exposure of internal package names: The approach could expose internal package names to external entities;
- Name conflicts: The solution does not address cases where internal packages share names with existing external ones.

Moreover, the experts provided useful feedback by proposing the following alternative solutions:

- Network-level rewrite: A long-term solution would involve modifying network policies, so that all traffic directed to the public index is rerouted to the internal one. However, implementing such a change is unlikely in the near future due to both technical and company governance challenges. This solution would resemble a man-in-the-middle (MITM) scenario between company computers and PyPI and may require non-standard solutions to ensure that TLS connections succeed, e.g. by deploying suitable trusted X.509 certificates on each computer.
- Blocking conflicting packages: A more immediate mitigation strategy is to block any PyPI package that shares a name with an internal package, thereby reducing the risk of dependency confusion. This can be achieved by intercepting all requests to PyPI, by means of a suitable proxy (which again acts as a MITM).

Telemetry. In contrast to dependency confusion, only four out of six experts were aware of the privacy concerns related to telemetry and the specific data collected by Acc-Py. However, they generally perceived this as a minor issue within the CERN ecosystem. The experts provided the following feedback:

- Need for better documentation on telemetry data collection: Experts emphasized that such practices should be explicitly communicated to users;
- Legal compliance: Logging and telemetry practices comply with CERN's Data Protection Policy. The Office of Data Privacy has reviewed and approved these practices, confirming their alignment with CERN's specific regulations (OC11);

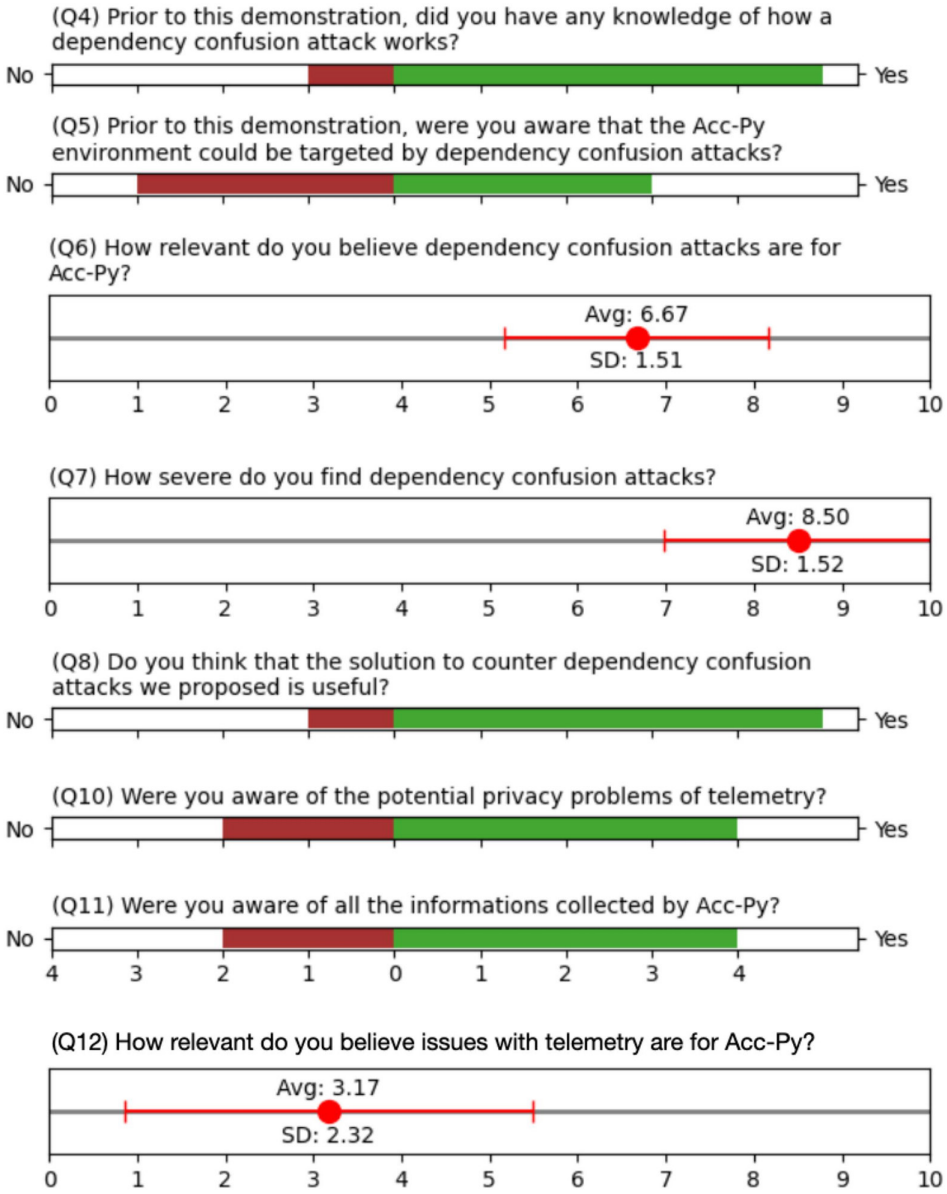

FIGURE 4 | Questionnaire responses summary.

- Awareness and transparency: Despite being legally compliant, telemetry practices require greater transparency and improved user awareness regarding the nature and extent of data collection.

> **RQ4:** *The feedback from the experts highlights that dependency confusion is widely recognized as a critical security issue, whereas concerns about telemetry are more nuanced. There is a shared commitment within the team to address security concerns and strengthen the Acc-Py ecosystem.*

> **Key Insight 4:** Enhancing the security of private package repositories necessitates tailored solutions that are specific to each organization's unique infrastructure and operational context. Consequently, effective security measures cannot be developed in isolation. The synergy between security researchers and repository maintainers represents a crucial socio-technical asset.

8 | Threats to Validity

Construct validity: The validation of true positives was based on expert knowledge, which may introduce subjective bias. Experts may overlook security issues, thus introducing false positives. However, the vulnerability triage and mitigation procedure we followed closely aligns CERN's one. The qualitative analysis of the questionnaire responses was performed using thematic analysis, a well-established technique for aggregating open-ended answers. All the researchers reviewed separately and discussed the categorizations to enhance reliability. This approach mitigated subjective interpretation.

Internal validity: The dynamic analysis tool was carefully tested by our team and validated by CERN engineers, and was deployed within Docker images to ensure reproducibility and isolation. However, potential implementation errors or environment-specific factors could still influence our results. Our comparative static analysis may be influenced by confounding factors such as package size, category, or development practices. Due to limited metadata availability for Acc-Py (e.g., number of contributors, usage context), we could not construct fully matched package pairs. We mitigate this threat by approximating control for these factors through the use of two PyPI subsets: a random 10% sample to capture general ecosystem diversity, and the top 10 K packages to reflect more mature and widely used projects. Additionally, we consider code size (MLoC) in our static analysis to reduce the impact of structural differences. For the dynamic analysis, we further tested whether package-size confounds install-time traffic; we did not control for package functionality, which remains a residual threat.

External validity: The static analyzers used in our study were selected to be diverse and representative of those commonly employed by practitioners and researchers. However, the choice of specific tools may limit the generalizability of our findings. Further studies incorporating additional tools could extend the applicability of our results to a broader set of security analysis techniques. Our analysis was performed on subsets of PyPI packages to maintain feasibility. These complementary subsets were deemed representative because their combination can capture both the diversity of the overall ecosystem and the maintenance quality of widely used packages. However, our conclusions may not fully generalize to the entire PyPI ecosystem.

Reproducibility: Our experimental data is publicly available [45].

9 | Related Work

The security of software package managers has been extensively studied due to its increasing relevance. Related research highlighted various attack vectors and evaluated detection methodologies and mitigation strategies [3, 14, 42, 46]. These contributions studied widely used ecosystems such as PyPI and npm in isolation; instead, we focused on the comparison of the security posture of Acc-Py, a specialized repository of Python packages at CERN, to PyPI, the official package index for Python. In this way, we were able to identify security challenges unique to private package repositories such as Acc-Py.

Duan et al. [42] proposed MALOSS, a comparative framework for assessing the security of PyPI, npm, and RubyGems. Their approach combines metadata analysis with static and dynamic analysis to detect registry abuse, discovering seven new malicious PyPI packages confirmed by maintainers. Like their study, we highlight the risks of package manager abuse, emphasize the need for stronger security measures, and involve human experts to validate our findings.

Guo et al. [30] conducted an empirical study on the PyPI ecosystem, analyzing source code and proposing a dataset of malware packages. Although their study focused solely on malicious PyPI code, we compared Acc-Py and PyPI at the package level. Our study confirms some of their findings, such as the effectiveness of static analyzers in detecting malicious patterns but with a high false positive rate. We were inspired by their work in analyzing install-time attacks, especially those exploiting the `setup.py` file, which they identified as a common vulnerability and attack vector in PyPI.

Zahan et al. [18] also focused on security attacks on package managers. Differently from us, they focused on the npm ecosystem, proposing a framework to prioritize weak link signals (i.e., indicators of security weaknesses) within the npm supply chain based on package metadata. Like our study, they discovered vulnerabilities in install scripts, highlighting the risks associated with execution during package installation. They also involved practitioners through a survey to discuss findings and propose additional weak link signals.

Vu et al. [3] conducted an empirical study on the PyPI repository, evaluating existing malware detectors on their benchmark dataset. Their findings highlight that these tools tend to generate an unmanageable rate of false positives for administrators. They also conducted interviews with repository administrators and contributors, emphasizing the importance of collaboration between security researchers and repository administrators for improving security. Our research, focusing on CERN's Acc-Py repository, aligns with their conclusions, demonstrating that such collaboration is crucial for assessing and enhancing the security of software ecosystems.

Zheng et al. [17] focus on dynamic code poisoning detection for NPM and PyPI by executing packages in a sandbox environment, applying fuzz testing, and reducing false positives. Also their work has been implemented in a real-world industrial setting, i.e., the Ant Group. In contrast, our study compares the security of private and public repositories, focusing on vulnerabilities like dependency confusion and misconfigurations and offering a complementary approach to securing software supply chains.

In summary, although prior research mostly focused on large repositories like PyPI and npm, our work examines the specialized Acc-Py index. By comparing it with PyPI, we identify both common and unique security challenges, emphasizing the importance of ecosystem-specific strategies and researcher-practitioner collaboration to strengthen supply chain security.

10 | Conclusions and Future Work

This study presented a comprehensive security assessment of Acc-Py, CERN's private package manager. Through static and metadata analysis, we demonstrated that Acc-Py has a strong security posture, as emerges from the comparison with PyPI. We identified potential vulnerabilities to dependency confusion and highlighted the importance of transparent telemetry collection practices. Collaborative expert feedback led to the discovery and resolution of an undocumented vulnerability, underscoring the value of joint security evaluations.

Beyond the specific case of Acc-Py, this research offers broader insights into private package repository security. First, the comparative analysis of private and public repositories through static and dynamic analysis is a valuable methodology for evaluating security posture. Secondly, the inherent risk of namespace collisions underscores the necessity of organizational-level controls to prevent dependency confusion. Additionally, the enhanced governance of private ecosystems introduces a critical trade-off between security monitoring and user privacy. Future research will extend the methodology and insights gained from this study to a broader spectrum of private package managers.

Acknowledgements

Open access publishing facilitated by Universita degli Studi di Udine, as part of the Wiley - CRUI-CARE agreement.

Funding

This work was supported by the project PNRR M4 C2 I1.3 "SEcurity and RIghts in the CyberSpace (SERICS)" (PE000 00014 - CUP H73C22000890001, D33C22001300002) PE7, funded by Next-Generation EU.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The replication package, including the data and artifacts used in this study, is publicly available [45].

Endnotes

[1]https://github.com/PyCQA/bandit, latest release: 1.8.3, February 2025.

[2]https://github.com/DataDog/guarddog, latest release: 2.4.0, February 2025.

[3]https://pypi.org/search/?q=yandex+AND+"dependency+confusion"&o=-created

References

1. H. Muhammad, L. C. V. Real, and M. Homer, "Taxonomy of Package Management in Programming Languages and Operating Systems," in *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, (2019): 60–66.

2. R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why Do Developers Use Trivial Packages? An Empirical Case Study on npm," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, (2017): 385–395.

3. D.-L. Vu, Z. Newman, and J. S. Meyers, "Bad Snakes: Understanding and Improving Python Package Index Malware Scanning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (IEEE, 2023), 499–511.

4. Y. Peng, R. Hu, R. Wang, C. Gao, S. Li, and M. R. Lyu, "Less Is More? An Empirical Study on Configuration Issues in Python Pypi Ecosystem," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, (2024): 1–12.

5. F. R. Cogo, G. A. Oliva, and A. E. Hassan, "An Empirical Study of Dependency Downgrades in the npm Ecosystem," *IEEE Transactions on Software Engineering* 47, no. 11 (2019): 2457–2470.

6. D. Spinellis, "Package Management Systems," *IEEE Software* 29, no. 2 (2012): 84–86.

7. R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab, "On the Impact of Using Trivial Packages: An Empirical Case Study on npm and PyPI," *Empirical Software Engineering* 25 (2020): 1168–1204.

8. 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)*, (2019): 1393–1410.

9. M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, Dimva 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17* (Springer, 2020), 23–43.

10. 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)* (IEEE, 2023), 1509–1526.

11. Y. Shen, X. Gao, H. Sun, and Y. Guo, "Understanding Vulnerabilities in Software Supply Chains," *Empirical Software Engineering* 30, no. 1 (2025): 1–38.

12. L. Williams, G. Benedetti, S. Hamer, et al., "Research Directions in Software Supply Chain Security," *ACM Transactions on Software Engineering and Methodology* (2024).

13. P. Ladisa, S. E. Ponta, A. Sabetta, M. Martinez, and O. Barais, "Journey to the Center of Software Supply Chain Attacks," *IEEE Security & Privacy* 21, no. 6 (2023): 34–49.

14. D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Towards Using Source Code Repositories to Identify Software Supply Chain Attacks," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, (2020): 2093–2095.

15. M. J. H. Faruk, M. Tasnim, H. Shahriar, M. Valero, A. Rahman, and F. Wu, "Investigating Novel Approaches to Defend Software Supply Chain Attacks," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (IEEE, 2022), 283–288.

16. M. Ohm and C. Stuke, "SoK: Practical Detection of Software Supply Chain Attacks," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, (2023): 1–11.

17. X. Zheng, C. Wei, S. Wang, et al., "Towards Robust Detection of Open Source Software Supply Chain Poisoning Attacks in Industry Environments," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, (2024): 1990–2001.

18. N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What Are Weak Links in the npm Supply Chain?," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, (2022): 331–340.

19. J. Nguyen Xuan, M. Dönszelmann, and B. Copy, "A C/C++ Build System Based on Maven for the LHC Controls System," in *Conf. Proc.*, Vol. 111010, (2011) WEPKS026.

20. A. Olaoye, "DevOps on Amazon Web Services (AWS)," *Beginning Devops on AWS for IOS Development: Xcode, Jenkins, and Fastlane Integration on the Cloud* (Springer, 2022), 45–83.

21. P. Elson, C. Baldi, and I. Sinkarenko, "Introducing Python as a Supported Language for Accelerator Controls at cern," *JACoW* 2022 (2022): 236–241.

22. I. Sinkarenko, P. Elson, W. Koorn, F. Iannaccone, and B. Copy, "JACoW: Towards a Flexible and Secure Python Package Repository Service," *JACoW ICALEPCS* 2023 (2023): THPDP067.

23. L. Evans, "The Large Hadron Collider," *Annual Review of Nuclear and Particle Science* 61, no. 1 (2011): 435–466.

24. P. Ladisa, M. Sahin, S. E. Ponta, M. Rosa, M. Martinez, and O. Barais, "The Hitchhiker's Guide to Malicious Third-Party Dependencies," in

Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, (2023): 65–74.

25. A. J. Jafari, D. E. Costa, A. Abdellatif, and E. Shihab, "Dependency Practices for Vulnerability Mitigation," (2023), arXiv preprint arXiv:2310.07847.

26. R. Appleyard and J. Adams, "Using the ELK Stack for CASTOR Application Logging at RAL," in *International Symposium on Grids and Clouds (ISGC)*, Vol. 15, (2015).

27. J. P. Gil, J. Reveco, and T.-C. Shen, "Operational Logs Analysis at Alma Observatory Based on ELK Stack," in *Software and Cyberinfrastructure for Astronomy IV*, Vol. 9913 (SPIE, 2016), 814–822.

28. P. S. Foundation, "Bigquery Datasets," (2025), Accessed: 2025-07-15.

29. Accelerator and B. P. C. P. At CERN, "Python Package Index (pypi)," (2025), Accessed: 2025-03-12.

30. 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)*. IEEE, (2023): 166–177.

31. N. Nagappan and T. Ball, "Static Analysis Tools as Early Indicators of Pre-Release Defect Density," in *Proceedings of the 27th International Conference on Software Engineering*, (2005): 580–586.

32. J. Ruohonen, K. Hjerppe, and K. Rindell, "A Large-Scale Security-Oriented Static Analysis of Python Packages in pypi," in *2021 18th International Conference on Privacy, Security and Trust (PST)* (IEEE, 2021), 1–10.

33. D. A. Kapustin, V. V. Shvyrov, and T. I. Shulika, "Static Analysis of Corpus of Source Codes of Python Applications," *Programming and Computer Software* 49, no. 4 (2023): 302–309.

34. T. Dunlap, S. Thorn, W. Enck, and B. Reaves, "Finding Fixed Vulnerabilities With Off-the-Shelf Static Analysis," in *2023 IEEE 8th European Symposium on Security and Privacy (Euros&p)* (IEEE, 2023), 489–505.

35. C. Huang, N. Wang, Z. Wang, et al., "{DONAPI}: Malicious {NPM} Packages Detector Using Behavior Sequence Knowledge Mapping," in *33rd Usenix Security Symposium (Usenix Security 24)*, (2024), 3765–3782.

36. semgrep Inc., "Semgrep Rules," (2025), Accessed: 2025-03-12.

37. Accelerator and B. P. C. P. At CERN, "Acc-Py Repository Packages and External Tools," (2025), Accessed: 2025-03-12.

38. ISO, "ISO/IEC 29147:2018 Information Technology Security Techniques Vulnerability Disclosure," (2025), Accessed: 2025-03-12.

39. Microsoft Azure, "3 Ways to Mitigate Risk When Using Private Package Feeds," (2020), https://azure.microsoft.com/mediahandler/files/resourcefiles/3-ways-to-mitigate-risk-using-private-package-feeds/3%20Ways%20to%20Mitigate%20Risk%20When%20Using%20Private%20Package%20Feeds%20-%20v1.0.pdf.

40. T. W. Anderson, T. W. Anderson, T. W. Anderson, T. W. Anderson, and E.-U. Mathématicien, *An Introduction to Multivariate Statistical Analysis*, vol. 2, (Wiley New York, 1958).

41. CERN, "Cern Legal Framework," (2025), Accessed: 2025-03-12.

42. R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages," Proceedings 2021 Network and Distributed System Security Symposium, (2020).

43. J. Linåker, S. M. Sulaman, R. Maiani de Mello, and M. Höst, "Guidelines for Conducting Surveys in Software Engineering," (2015), Lund University.

44. B. Kitchenham and S. L. Pfleeger, "Principles of Survey Research Part 4: Questionnaire Evaluation," *ACM SIGSOFT Software Engineering Notes* 27, no. 3 (2002): 20–23.

45. M. Lizzit, F. Pinzauti, M. Miculan, and V. Riccio, "Replication Package," (2025), https://github.com/cysecud/accpy-rep-pkg.

46. R. K. Vaidya, L. De Carli, D. Davidson, and V. Rastogi, "Security Issues in Language-Based Software Ecosystems," (2019), arXiv preprint arXiv:1903.02613.