Uncovering Similar but Different Packages in PyPI and Potential Security Threats

SUNHA PARK, Korea University, Republic of Korea
SOOJIN HAN, Dongduk Women's University, Republic of Korea
SEUNGHOON WOO, Korea University, Republic of Korea

In this study, we present a large-scale, in-depth study of package replication in PyPI. As a vital platform, PyPI streamlines Python package distribution for developers. However, beyond small-scale code cloning, we observe that many replicated packages exist on PyPI, which duplicate most of the codebase from existing packages. Such replication not only confuses developers but also propagates known vulnerabilities and enables the creation of new malicious packages. To address this issue, we comprehensively examine the characteristics and potential threats of replicated packages. Using one-third of the entire PyPI repository (200K packages), we investigate replication from three perspectives: replication of popular packages, vulnerable packages, and malicious packages. Our experiments reveal three critical findings about package replication in PyPI: (1) by identifying 1,361 replicated packages of the top 3K popular projects, we show that replication frequently redistributes substantial portions of existing packages under different maintainers; (2) by uncovering 256 previously unknown replicated vulnerable packages, we demonstrate that replication creates vulnerability blind spots that current detection tools rarely catch; (3) by analyzing 3,883 known malicious packages, we found that 186 (4.79%) replicated popular ones, and this pattern further led us to identify seven previously unknown replicated malicious packages, highlighting its role as an attack vector for malware distribution through minor modifications and code injection.

CCS Concepts: • **Security and privacy** → **Software security engineering**; • **General and reference** → **Empirical studies**.

Additional Key Words and Phrases: Python package replication, Vulnerability propagation, Malware detection.

ACM Reference Format:
Sunha Park, Soojin Han, and Seunghoon Woo. 2026. Uncovering Similar but Different Packages in PyPI and Potential Security Threats. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE210 (July 2026), 22 pages. https://doi.org/10.1145/3808217

1 Introduction

PyPI has grown rapidly as the central platform for distributing and reusing Python packages, hosting hundreds of thousands of projects widely used by developers worldwide [3]. Although the platform encourages code reuse and modular development, not all reuse patterns are safe [18, 44]. Beyond simple code cloning, where developers copy parts of existing projects, there is also *package replication*, where most of a package's codebase is duplicated and uploaded as a new package.

From an engineering perspective, this can provide high efficiency by reducing development time and costs, but at the same time, package replication may introduce security risks [14]. Malicious package replication can be exploited as the entry point for attacks such as typosquatting, which is already widely recognized as posing significant risks by misleading developers [19, 28]. Moreover,

Authors' Contact Information: Sunha Park, Korea University, Seoul, Republic of Korea, sunhap@korea.ac.kr; Soojin Han, Dongduk Women's University, Seoul, Republic of Korea, hsjsstn@gmail.com; Seunghoon Woo, Korea University, Seoul, Republic of Korea, seunghoonwoo@korea.ac.kr.

ACM 2994-970X/2026/7-ARTFSE210
https://doi.org/10.1145/3808217

even without malicious intent, packages replicated from those containing vulnerabilities can unintentionally propagate those security issues further [4, 32, 40].

Given these concerns, prior work has examined the security of PyPI. Research on malicious package detection has ranged from rule-based [9, 20] to machine learning–based approaches [11, 22, 35, 46], and the construction of public malicious package datasets [13, 30]. Other studies have investigated vulnerabilities and maintenance issues in Python packages [1, 37]. However, their goal is to identify insecure packages in PyPI, which is rather distant from detecting replicated packages or analyzing replication to uncover insecure packages. Beyond PyPI, prior work in other ecosystems has explored code clones [15, 16, 21, 34, 44], orphan vulnerabilities [31, 32], shrinkwrapped clones in npm [45], and large-scale duplication on GitHub [23]. Nevertheless, no large-scale study has systematically examined package replication in the Python ecosystem from a security perspective.

In this study, we aim to closely analyze this problem and address the following research questions.

RQ1. Popularity analysis. How prevalent are replicated packages in PyPI, and what are their fundamental characteristics (*e.g.*, naming and code similarity)?

RQ2. Vulnerability analysis. How widespread are vulnerabilities in replicated packages, and what are their primary characteristics (*e.g.*, CWE categories and CVSS scores)?

RQ3. Malware analysis. What are the characteristics of replicated malicious packages? Can these characteristics help identify unknown replicated malicious packages?

Unlike traditional code clones that involve reusing only small fragments of code, we define a *replicated package* as a new package that is created by reusing a substantial portion of an existing package's codebase (*e.g.*, two packages are considered highly replicated when their code similarity exceeds 90%; see Section 2.4). Although prior research has discussed partial code reuse, the identification and characterization of replicated packages and their security implications have not been thoroughly examined. We address these gaps by answering the three key research questions.

To this end, we constructed five datasets: (1) *popularity*, (2) *vulnerability*, (3) *malware*, (4) *recent*, and (5) *candidate* datasets (as of Aug. 2025). The candidate dataset includes the latest versions of approximately one-third of all PyPI packages (200K out of 670K) and is used to analyze replication trends and identify vulnerable packages. The popularity dataset covers the top 3K packages by downloads, while the vulnerability and malware datasets contain known vulnerable and malicious packages. The recent dataset was collected to evaluate the security of newly published packages (see Section 2.1). To identify replicated packages, we propose a framework combining embedding, clustering, and code similarity analysis. Packages are embedded using `CodeT5+` [41] (see Section 2.2), followed by dimensionality reduction with `UMAP` [25] and density-based clustering via `HDBSCAN` [24]. Replication is then confirmed through fine-grained code similarity analysis (see Section 2.4).

Our experiments reveal three critical findings: (1) *replication frequently redistributes substantial portions of existing packages*, often under different maintainers, (2) *replication creates vulnerability blind spots* that are difficult for current tools to detect, and (3) *replication provides an attack vector for malware distribution* through minimal modifications combined with malicious code injection.

From the candidate dataset, we identified 1,361 replicated packages derived from popular projects, 60% of which were maintained by different authors than the originals (see Section 4). To evaluate security implications, we mapped 1,072 known vulnerable packages to the candidate dataset and discovered 256 previously unknown replicated vulnerable packages; 35.14% were high or critical severity and had remained unmaintained for an average of 1,600 days. Existing dependency scanners missed over 98% of these cases, exposing a critical blind spot in current vulnerability management approaches that rely on package names and versions (see Section 5). We further analyzed malicious packages and found that, among 3,883 known malicious packages, 186 were replication-based variants with injected suspicious APIs (*e.g.*, `arbitrary code execution`). Applying our method to

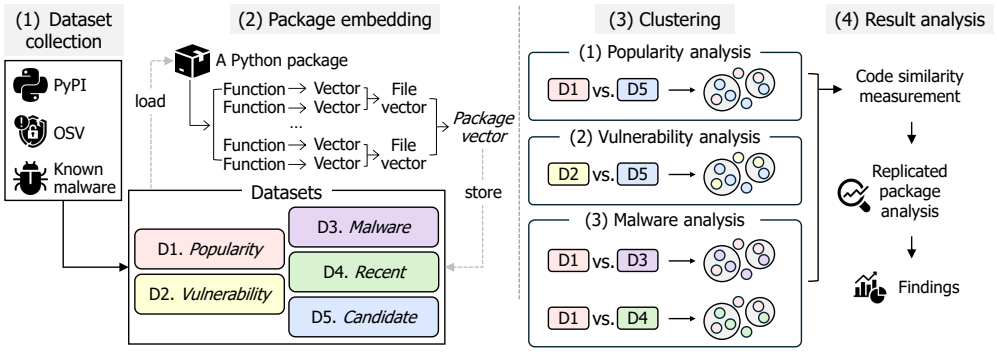


Fig. 1. Overview of the empirical study.

newly registered PyPI packages uncovered seven previously unknown malicious packages, all later removed after disclosure (see Section 6).

Contribution. This paper makes the following four main contributions.

- *Large-scale analysis.* To analyze overall insights and potential risks of package replication, we constructed five datasets and conducted large-scale experiments. In particular, the candidate dataset was built to represent PyPI, consisting of the latest versions of 200,737 repositories.
- *Thorough study.* We conducted a thorough and multi-faceted analysis of package replication. Our experiments did not rely solely on name or metadata similarity; instead, we incorporated clustering and code similarity techniques, and designed specialized experiments for vulnerability and malware detection to ensure a more rigorous evaluation.
- *Practicality.* Our experiments showed a practical impact by reporting and removing high-risk vulnerable and malicious packages. We also exposed the limits of existing techniques and proposed a replication-based method as a practical way to strengthen PyPI's security.
- *Open source.* Our source code and experimental results are available at: https://github.com/sunha21/pypi-replication-analysis.

2 Methodology

Figure 1 illustrates the high-level workflow of this study. To comprehensively analyze package replication, we constructed five datasets and embedded all packages to generate package-level vectors. These vectors were then clustered according to the analysis objectives, followed by additional code similarity analysis and other inspections to perform a deep analysis of package replication.

2.1 Dataset Collection

We construct the following five datasets: (1) *popularity*, (2) *vulnerability*, (3) *malware*, (4) *recent*, and (5) *candidate* datasets. Table 1 provides a quantitative overview of the collected datasets.

2.1.1 Popularity Dataset. The popularity dataset comprises the most popular packages in PyPI and becomes the reference point for identifying replications. To construct this dataset, we collected the codebases of the top 3,000 packages from PyPI (as of August 2025) based on the Top PyPI Packages dataset [38]. Each of these packages had over one million downloads in August 2025, making them a reasonable standard for popular packages. We used the official PyPI JSON API to download all versions of the packages and extracted only the Python source files. While we initially targeted 3,000 packages, some contained no Python source or no valid functions. After filtering, the dataset

Table 1. Overview of the five datasets collected for experiments.

Dataset	Popularity	Vulnerability	Malware	Recent	Candidate
#Packages	2,767	1,072	2,656	25,407	200,737
#Versions	160,317	70,280	3,883	36,008	200,737
#Lines of Code	8,701,101,354	4,628,177,354	7,291,220	196,760,716	1,020,502,964

comprises 2,767 packages, 160,317 versions, and 8,701,101,354 lines of code. Note that all dataset statistics reported in this section are measured after filtering out non-Python and functionless files.

2.1.2 Vulnerability Dataset. We classified any package with one or more vulnerabilities reported in PYSEC (Python Security Database) and GHSA (GitHub Security Advisory) from Google's Open Source Vulnerabilities (OSV) [12] as a vulnerable package. Consequently, we collected all versions of the 1,072 identified vulnerable packages, resulting in 70,280 package versions with 4,628,177,354 accumulated lines of code for analysis. This set includes not only vulnerabilities related to input validation but also various types, such as path traversal and cross-site scripting.

2.1.3 Malware Dataset. The malicious dataset was collected from publicly available sources: Guo *et al.* [13] and Ohm *et al.* [30]. They include various types of malicious Python packages (*e.g.*, download-driven, typosquatting, and dependency confusion). As a result, we obtained 2,656 malicious packages comprising 3,883 versions (with 7,291,220 accumulated lines of code), which are used to examine the prevalence and characteristics of duplicated malicious code.

2.1.4 Recent Dataset. This dataset is constructed to capture the trends of the most recent PyPI packages. From May to August 2025, we collected newly registered packages by monitoring the PyPI RSS feed. For each package, we downloaded all available versions at the time of discovery, resulting in 25,407 packages and 36,008 versions with 196,760,716 accumulated lines of code.

2.1.5 Candidate Dataset. Finally, the candidate dataset consists of general packages from PyPI and is regarded as the pool for replication detection. However, collecting all packages from PyPI is resource-inefficient and imposes excessive load during analysis. Therefore, we randomly sampled a subset of packages, representing approximately one-third of the 670,000 packages available on PyPI (as of August 2025), and collected the codebases of their latest versions to construct the candidate dataset. In total, the dataset comprises 200,737 packages and more than 1B lines of Python code.

2.2 Package Embedding

To address our research questions, we perform comparisons across the constructed datasets. For example, to address RQ1 (*i.e.*, popularity analysis), we need to compare the packages in the popularity dataset with those in the candidate dataset. However, direct comparison of the raw codebase is inefficient due to its inherent code diversity, as code can be modified during package replication.

To identify replicated packages, therefore, we embed the packages belonging to each dataset and then apply clustering across embeddings from different datasets (see Section 2.3). Specifically, we embed all the packages collected in the dataset into 256-dimensional vectors.

Before embedding packages, we preprocess the source code by extracting functions and removing all comments to focus solely on functionality and semantics. For function extraction, `ctags` [6] was adopted due to its lightweight and efficient parsing, which scales well to large codebases.

For package embedding, we employ `CodeT5+` [41], a transformer-based large language model designed for code understanding and generation. Its strong performance on clone detection makes it particularly suitable for identifying semantic similarities in source code [29]. To maintain code semantics and ensure consistent analysis, we first apply `CodeT5+` at the function level. Let P be a Python package represented as a set of files, and let each file F_i be represented as a set of functions.

$$P = \{F_1, F_2, \ldots, F_m\}, \quad F_i = \{f_{i1}, f_{i2}, \ldots, f_{in_i}\}.$$

Let ϕ be the embedding function implemented using `CodeT5+`, which maps a function f_{ij} into a 256-dimensional vector.

$$v_{ij} = \phi(f_{ij}) \in \mathbb{R}^{256}$$

Here, we applied a sliding window approach with a stride of 128 tokens to handle long functions, since the model accepts a maximum input length of 512 tokens per function and many real-world functions exceed this limit. Each function is then encoded into a 256-dimensional vector.

Next, we obtain the file vectors by applying a mean operation over the function vectors contained in each file. Let v_{F_i} be the vector representation of file F_i, defined as the mean of its function vectors.

$$v_{F_i} = \frac{1}{n_i} \sum_{j=1}^{n_i} v_{ij}$$

Finally, we obtain the package vector by applying a mean operation over its file vectors. Let v_P be the vector representation of package P, defined as the mean of its file vectors.

$$v_P = \frac{1}{m} \sum_{i=1}^{m} v_{F_i}$$

Through this process, each package version is ultimately represented as a single 256-dimensional vector, which enables efficient similarity computation across packages.

2.3 Clustering

In our experiments, clustering algorithms are required to assess the similarity of embedded packages. Clustering is performed between two different datasets depending on the research question. However, because the generated package vectors are 256-dimensional, performing effective clustering directly on them is challenging. This is because high-dimensional spaces suffer from the curse of dimensionality, where distances become less meaningful and data sparsity increases, substantially degrading clustering performance.

To address this issue, we first perform dimensionality reduction. Specifically, we adopt `UMAP` [25], which effectively maps high-dimensional code embeddings into a lower-dimensional space while preserving the underlying data structure. This property makes `UMAP` well-suited for clustering tasks on code embeddings.

For clustering, we adopt `HDBSCAN` [24], which does not require a predefined number of clusters. In practice, the number of similar package groups is unknown, and specifying the number of clusters in advance can lead to significant detection errors. The algorithm can identify clusters of varying densities, making it appropriate for package detection where different types of packages may exhibit different clustering characteristics.

Formally, given the set of package vectors $\mathcal{V} = \{v_{P_1}, \ldots, v_{P_N}\}$ and the dimensionality reduction function ψ (`UMAP`), we obtain lower-dimensional embeddings $\tilde{v}_{P_i} = \psi(v_{P_i})$. The clustering function C (`HDBSCAN`) is then applied as follows.

$$\{C_1, C_2, \ldots, C_K\} = C\big(\psi(\mathcal{V})\big),$$

where each C_k denotes a cluster of similar packages.

To capture small but meaningful clusters effectively while reducing outliers, we set the minimum cluster size (`min_cluster_size`) to 2, the minimum number of samples (`min_samples`) to 1, and the option `cluster_selection_epsilon` to 0.15. These values were determined empirically through

parameter tuning. In addition, clustering too many package vectors at once may reduce accuracy, thus we divided the candidate dataset into batches and clustered each batch separately.

2.4 Identifying Replicated Packages

Finally, we examine the clustering results and provide insights into replicated packages. However, simply assuming that all packages within the same cluster constitute replications may undermine accuracy. To address this, within each cluster, we computed detailed code similarity scores for all possible package pairs.

2.4.1 Code Similarity Measurement. Let the two packages under comparison be denoted as X and Y. We compare both the file paths and the file contents of X and Y. Each pair of files is classified into one of the following four categories: *identical*, *modified*, *added*, or *deleted*.

- *Identical.* The file in X and the file in Y share the same path (including the immediate parent directory name) and have exactly the same file contents (*i.e.*, code syntax).
- *Modified.* The file paths match, but the file contents differ.
- *Added.* A file exists in Y such that no file in X has both the same path and identical contents.
- *Deleted.* A file exists in X such that no file in Y has both the same path and identical contents.

Based on these classifications, we computed a code similarity score between the two packages.

$$\texttt{sim}(X, Y) = \frac{\#\text{identical} + (0.5 \times \#\text{modified})}{\#\text{identical} + \#\text{modified} + \#\text{added} + \#\text{deleted}}$$

The weight of 0.5 was empirically determined from our candidate dataset. We first defined a preliminary similarity score based on file identity: the number of identical files divided by the total number of files in the union of the original and replicated packages. Among the 2,350 packages with a similarity score of at least 0.1 and at least one modified file, the average line-level modification ratio was 0.44. Based on this observation, we assign a weight of 0.5 to modified files.

2.4.2 Definition of Package Replication. Given two packages X (reference package) and Y that belong to the same cluster, we define the replication level of Y relative to X based on their similarity score $\texttt{sim}(X, Y)$ as follows.

- If $\mathbf{\texttt{sim}(X, Y) \geq 0.9}$, Y is considered a **highly replicated package** of X.
- If $\mathbf{0.5 \leq \texttt{sim}(X, Y) < 0.9}$, Y is considered a **partially replicated package** of X.

In our experiments, all replicated packages underwent additional inspection. Although no strict guidelines exist, we adopted 0.9 as a practical boundary for near-identical packages and 0.5 as the cutoff for partial similarity. Section 7.1 presents the sensitivity analysis of this threshold.

2.4.3 Name and Metadata Similarity. Package names and metadata fields are the primary sources of information users rely on to identify and install Python packages. To measure name similarity, we computed the `Levenshtein` ratio between two package names. This ratio normalizes edit operations by the combined length of the two names, where a score of 1 indicates identical names.

For metadata, we considered five fields (`author`, `author_email`, `summary`, `description`, and `home_page`), and performed exact string matching for each field.

2.4.4 Human Resources. To answer the research questions, the identified replicated packages were analyzed by three experts. One has more than ten years of experience in software engineering and security, while the other two have three and five years of experience, respectively. The manual analysis mainly involves directly reviewing source code or comparing metadata, and, when necessary, includes triggering vulnerabilities or testing malicious code in a sandbox environment.

3 Evaluation

We first evaluate our replication detection approach to validate its effectiveness and to enhance the credibility of our findings in addressing the research questions.

3.1 Replicated Package Detection Accuracy

3.1.1 Methodology. To assess accuracy, we construct a validation set and adopt a comparative validation strategy using a code clone detection tool as a reference. This is necessary because no ground-truth dataset exists for package-level replication, and defining replication at this level is inherently challenging: metadata-based signals are insufficient, and code-level similarity often requires subjective judgment. Therefore, we selected SourcererCC [34], a scalable, language-agnostic code clone detection tool based on token similarity. Because SourcererCC operates at the file level (*i.e.*, detecting similar files using token-level similarity), we extend its output to compute package-level similarity. For a package pair X and Y, we measure the proportion of files involved in clone relationships across both packages using an 80% token similarity threshold (default):

$$\mathtt{sim}_{SCC}(X, Y) = \frac{|F_X^{clone}| + |F_Y^{clone}|}{|F_X| + |F_Y|}$$

where F_X and F_Y denote the sets of files in packages X and Y, while F_X^{clone} and F_Y^{clone} represent the files that participate in at least one clone relationship detected by SourcererCC.

We use the popularity dataset (2,767 packages) as the baseline and compare it against the candidate dataset (200,737 packages). Because SourcererCC is not designed to operate at this scale, we adopt a filtered evaluation strategy. Specifically, we first apply our replication detection approach and retain package pairs with a similarity score above 0.1, excluding clearly unrelated pairs while preserving borderline cases. This results in 3,086 candidate pairs, which are analyzed using SourcererCC.

3.1.2 Criteria for Result Analysis. The 3,086 candidate pairs were manually analyzed and classified into replicated and non-replicated packages. To reduce ambiguity, we first filtered out clear-cut cases: pairs with similarity scores ≥ 0.7 in both tools were labeled as replicated, whereas those with similarity scores < 0.3 in both tools were labeled as non-replicated.

For the remaining ambiguous cases, we prioritized labeling according to the following criteria. First, we classified a pair as replication when it exhibited strong replication signals, *i.e.*, high name similarity (≥ 0.7) together with substantial structural similarity (≥ 0.5 in at least one tool or ≥ 0.3 in both tools). Next, for cases involving vendoring, we labeled a pair as replication only when one package was predominantly derived from the other and further extended or modified it. If the similarity arose because one package included the other as just one component among multiple incorporated libraries, we treated the relationship as ordinary code reuse (*i.e.*, non-replication). Finally, the detected package pair was not considered replicated when the observed similarity was limited to small fragments or boilerplate code.

Any cases that could not be conclusively determined under these criteria were finally resolved based on agreement among the participating analysts. As a result, from the 3,086 candidate pairs, we identified 1,767 replicated pairs and 1,319 non-replicated package relationships.

Table 2. Results of replicated package detection accuracy evaluation (TP: True Positive, FP: False Positive, FN: False Negative, TN: True Negative).

Metric	#TP	#FP	#FN	#TN	Precision	Recall	F1-score	Accuracy
Our approach	1,361	125	406	1,194	**91.58%**	77.02%	**83.68%**	**82.79%**
SourcererCC [34]	1,389	304	378	1,015	82.04%	**78.61%**	80.29%	77.90%

3.1.3 Result Analysis. Table 2 presents the comparison between our approach and SourcererCC under a replication threshold of 0.5 for both methods. Although the scoring methods differ, this threshold indicates cases where at least half of the package-level content is considered replicated.

Overall, because SourcererCC performs finer-grained clone analysis, it achieved a slightly higher recall (78.61% vs. 77.02%). However, our approach yielded substantially higher precision (91.58% vs. 82.04%), resulting in a superior F1-score and overall accuracy.

Notably, SourcererCC generated numerous FPs for small packages. Incidental code overlap cases were frequently assigned high similarity scores and incorrectly labeled as replication. Moreover, similarity driven by commonly included boilerplate files, such as `setup.py`, further contributed to FP classifications. In addition, when code modifications were introduced during the package replication process and sufficient file-level clones were not detected, SourcererCC reported FNs.

Our approach incorporates file names as well as added and deleted files into the similarity computation, which makes it more robust to incidental code overlap cases and results in fewer FPs. However, our approach produced several FNs. Because modified files are assigned uniform weight, near-identical but slightly altered files fall below the replication threshold (*i.e.*, FNs). Furthermore, replication involving extensive file additions or deletions reduces similarity and escapes detection.

Despite the identified FPs and FNs, the achieved accuracy provides a reliable foundation for analyzing the validated true positive replication cases in the subsequent RQs.

3.1.4 Threats to Validity. To mitigate subjectivity, we labeled ambiguous cases using similarity thresholds and structural criteria. Nonetheless, labeling bias and human error may occur. Moreover, our extension of SourcererCC to package-level similarity has not been independently validated; thus, the comparison should be interpreted as relative rather than absolute. Finally, because SourcererCC frequently maps a single file to multiple files across packages, it tends to inflate similarity scores, potentially yielding higher FPs and lower FNs than its intrinsic performance would indicate.

3.1.5 Performance. In the experiment comparing the popularity dataset (2,767 packages) with the candidate dataset (200,737 packages), we measured the execution time of the three stages in our pipeline: embedding, clustering, and similarity computation. All experiments were conducted on a server equipped with two NVIDIA RTX A6000 GPUs (48GB GDDR6 each), an AMD Ryzen Threadripper Pro 7965WX processor (24 cores, 48 threads), 384GB ECC RAM, and 4TB SSD.

For embedding, each package version required less than five seconds on average. Clustering across 200 batches required a total of 1,860 s. Finally, within each cluster, we first identified candidate similar package pairs and then computed their similarity scores. Processing approximately 2.74 million such pairs required 36 hours in total, corresponding to about one second per pair on average. Our choice of file-level granularity, a relatively coarse abstraction, combined with efficient and scalable techniques and encoding strategies, results in a replication detection algorithm that is inherently scalable. In contrast, although SourcererCC was applied only to the specific software version pairs identified after our clustering stage and performed similarity computation, it still required five seconds per package pair on average. In practice, the process must also consider identifying the most similar version pairs within each cluster. Therefore, this difference further underscores that our approach is designed with scalability as a primary consideration.

Table 3. Sensitivity analysis of clustering parameters.

min_cluster_size	min_samples	epsilon	Avg. #Clusters	Avg. #Outliers	#Pairs (≥ 0.5)
5	20	0	207	5588	54
5	5	0	908	2789	76
5	5	0.15	728	1907	73
3	2	0.15	1000	746	84
2	2	0	1960	2439	71
2	**1**	**0.15**	**1232**	**393**	**94**

3.2 Clustering Threshold Sensitivity

We then examined how the parameters of HDBSCAN affect cluster structure. In this study, clustering is employed as a preprocessing step to generate candidate pairs across the entire package space, rather than the final stage for determining replication packages. Therefore, we adjusted the parameters to maximize the range of candidates searched for.

To strengthen the observed trends without evaluating the full dataset, we conducted sensitivity analysis on randomly selected 20 (out of 200) batches. Table 3 presents the results of the clustering threshold sensitivity analysis. Increasing `min_cluster_size` filtered out small clusters, while increasing `min_samples` classified more boundary points as noise. The `epsilon` parameter controlled the merging of nearby dense regions. Overall, relaxing density constraints reduced outliers and increased candidate clusters, thereby expanding the pool of potentially replicated packages.

Therefore, to minimize outliers and reduce excessive splitting, we set `min_cluster_size` to 2, `min_samples` to 1, and `epsilon` to 0.15. Using this configuration, clustering produced an average of 1,232 clusters and 393 outliers per batch (across 200 batches in total).

4 Popularity Analysis (RQ1)

We then aim to answer RQ1 by analyzing popular packages on PyPI to determine how extensively they have been replicated, as well as the causes and characteristics. To this end, we provide a detailed analysis of the results obtained from accuracy evaluation (Section 3.1). This study does not aim to analyze the complete set of ground-truth replications. Instead, we focus on the characteristics of replications identified by our similarity metric and categorized as high or partial similarity. Replications with similarity scores below 0.5 were excluded, as including them would undermine analytical consistency and reflect fragment-level code cloning rather than package-level replication.

4.1 Status of Popular Package Replications

4.1.1 Status. We examine how extensively 2,767 popular packages are replicated among 200,737 PyPI packages. Consequently, we identified **1,361** replicated packages: *334 (24.54%) highly* replicated packages and *1,027 (75.45%) partially* replicated packages. This observation suggests that replication manifests in different forms, from near-complete copies to partial overlaps. Considering the current size of PyPI (≈ 670,000 packages as of August 2025), we estimate that more than 4,000 packages (≈ 1,361 × 3) in the entire ecosystem may exhibit duplication behavior.

Although the proportion of replicated packages appears relatively small at 0.6%, this result is still meaningful. Given the vast scale of the PyPI ecosystem, even a fraction of a percent amounts to more than a thousand replicated packages. This scale indicates that replication is not a negligible phenomenon and should be considered when analyzing ecosystem quality.

Finding 1. *We identified 1,361 replicated packages, many of which reproduce the majority of the original package's code. This highlights that replication often involves extensive reuse of the original implementation, making it a non-negligible phenomenon affecting ecosystem quality.*

4.1.2 Causes of Replication. Through manual inspection, we inferred the reasons for package replication. The most common reason is *customization*, where developers copy packages to extend or modify their functionality. Among these cases, many involved only minor changes, such as minimal edits to the `setup.py` file. In fact, some packages reproduce the original codebase exactly without any modifications, which appears to be intended for mirroring or rehosting.

Another frequent purpose is vendoring (or internal forking), where external dependencies are cloned to operate within isolated environments, reduce dependency risks, or apply controlled updates. These replications often preserve the exact code of the original package but are redistributed under a different name. Some replications were created for backward compatibility, such as maintaining support for Python 2. In these cases, developers re-released the same package with only minimal changes to ensure compatibility with legacy environments. Finally, we observed some replications with malicious intent. In such cases, attackers insert malicious code into the replicated package and redistribute it, thereby exploiting replication as a way for malware propagation. This will be examined in detail in the experiments for RQ2 and RQ3.

Finding 2. *In many cases, package replication was driven by specific customization needs. In some cases, it was exploited for malicious purposes. These observations suggest that large-scale replication warrants ecosystem analysis to better understand its maintenance and security implications.*

4.2 Characteristics of Replicated Packages

4.2.1 Maintainer. We first analyzed the ownership of packages. Because the *author* field can be arbitrarily set, we focus on the *maintainer* displayed on the PyPI website. Of the 1,361 detected replicated packages, 817 (60.03%) were distributed by accounts different from those of the original packages, indicating that the vast majority of replications were published by unrelated maintainers.

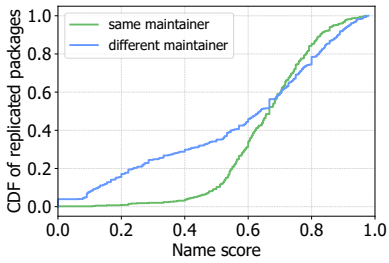
When replicated packages are maintained by the same maintainer, they may reflect legitimate scenarios such as renaming or restructuring. Replication by different maintainers, however, introduces a governance boundary between the original and the replicated package. Although code reuse and modification are fundamental principles of open-source ecosystems, near-full replication under separate maintainership can hinder patch propagation, obscure accountability, and fragment maintenance efforts. In such cases, security fixes applied upstream may not be consistently adopted downstream, potentially leaving replicated variants exposed.

Finding 3. *39.97% of replicated packages were created by the same maintainer, whereas 60.03% were maintained by different maintainers. Although replication is not inherently problematic, the high proportion of cross-maintainer replication raises concerns such as patch consistency.*

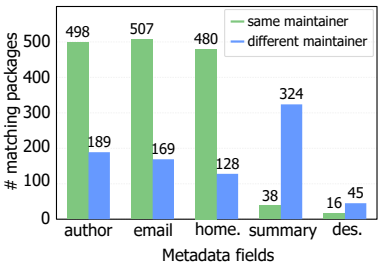
4.2.2 Name and Metadata Similarity. We classify replicated packages by whether they share the same maintainer (544) or not (817) and compare their names and metadata with the originals to distinguish legitimate republishing from potentially malicious or unauthorized replication.

Replication by the Same Maintainer (544 Packages). Figure 2a presents the cumulative distribution function (CDF) of name similarity. For replicated packages maintained by the same maintainer, the average name similarity was 0.65 (median 0.72). Notably, 222 of 544 packages exhibited high similarity (≥ 0.7), indicating that maintainers often reused the original name with minor modifications, such as adding a prefix or suffix (*e.g.*, `Keras` (original) and `Keras_nightly` (replicated)).

For metadata similarity, 491 packages (89.4%) retained at least three identical metadata fields. As shown in Figure 2b, the fields that were most frequently matched were `author_email` (507), followed by `author` (498) and `home_page` (480). When metadata differed, packages often shared similar code but served different functional purposes (*e.g.*, `safetycli` and `safety`).



(a) Distribution of **name similarities** between original and replicated packages.



(b) Illustration of **metadata** matches (original vs. replicated).

Fig. 2. Graphs on name and metadata similarity. Packages maintained by the same maintainer generally have more similar names and a higher possibility of metadata matching.

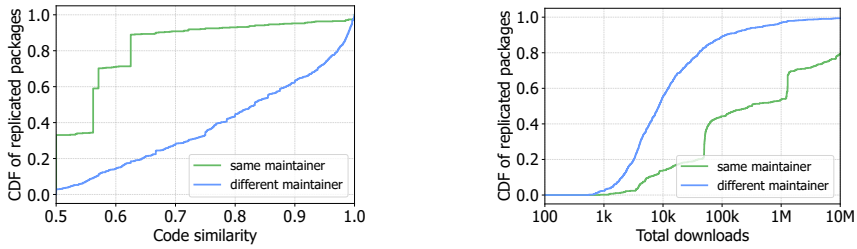
Finding 4. *For replicated packages with the same maintainer, 222 out of 544 packages had highly similar names to their original counterparts. However, the summary and description fields often differed, indicating that the metadata was selectively modified to reflect the intended purpose.*

Replication by Different Maintainers (817 Packages). For replicated packages maintained by different accounts, the average name similarity with their original counterparts was 0.56 (median 0.636; see Figure 2a). Among the 308 highly replicated packages, 149 (48.37%) showed a name similarity score of 0.7 or higher, indicating that their names were often very similar. However, among the 509 partially replicated packages, 189 (37.13%) reached a name similarity score of 0.7 or higher, suggesting a tendency to upload packages to PyPI with more than half of the code replicated but with significantly different names. When a different maintainer republishes most of the code under a highly similar name, it may create user confusion, underscoring the need for careful monitoring to preserve ecosystem clarity and trust.

For metadata similarity, 12 packages had identical values across all five fields even though they were maintained by different accounts. The number of replicated packages that directly reused elements of the original package's metadata is shown in Figure 2b. Overall, 198 (24.23%) highly and 205 (25.09%) partially replicated packages shared at least one identical field.

An interesting observation is that packages replicated by entirely different maintainers frequently retained the original summary. This may be due to convenience or an intentional choice to reduce the effort required while still benefiting from the credibility of the original package. Note that metadata duplication was particularly prevalent among packages with high code similarity. For example, `PyYAMLp-5.4.1` is nearly identical to `PyYAML-5.4.1`, sharing the same source code and all metadata fields, with only the package name and maintainer altered. Such cases are difficult to distinguish from the original and can mislead users. Unlike the original projects, these replicated packages offer no guarantee of consistent maintenance and may retain unresolved vulnerabilities or introduce malicious changes. These findings emphasize the importance of scrutinizing replicated packages to maintain ecosystem trustworthiness.

Finding 5. *Despite being maintained by different accounts, 338 (out of 817; 39.58%) packages exhibited name similarity scores above 0.7 with the original. We further observed that 403 replicated packages (49.32%) shared at least one identical metadata field, which increases the risk of user confusion and potential security threats.*



(a) Distribution of **code similarities** between original and replicated packages.

(b) Distribution of **total downloads** in replicated packages.

Fig. 3. CDF graphs of code similarity and total downloads. In general, same-maintainer replications exhibit higher code similarity and download counts.

Table 4. Number of the files by type.

Type	Total	Identical	Modified	Added	Deleted
#Files	227,380	165,887	24,313	32,310	4,870

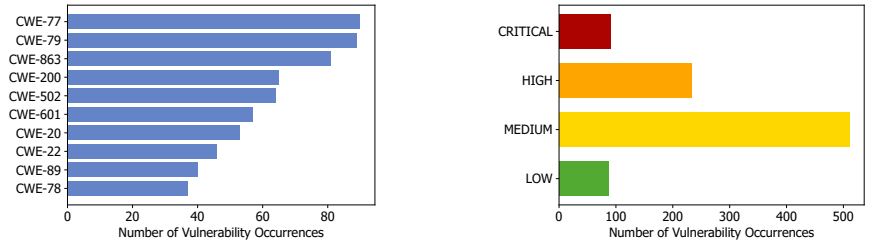
4.2.3 Code Similarity. Next, we examine the code similarity between replicated packages and their original counterparts. Figure 3a shows the CDF of code similarity. Replicated packages with the same maintainer achieved an average code similarity score of 0.58 (median 0.56), whereas those with different maintainers showed a higher average of 0.80 (median 0.83). Table 4 presents the types of file changes identified during replication. We observed that most files remained identical, while a smaller portion were modified, added, or deleted. On average, 85.04% of the original package's files were directly reused, 12.46% were replicated with modifications, and 2.50% were deleted. Although the proportion of modified files was modest, the average line-level code modification ratio was approximately 39%, which implies that code changes were non-trivial once modifications occurred.

In addition, we observed that 32,310 files were newly added during replications. Although many appear to support functional extension, the possibility of security-relevant functionality calls for close scrutiny. Moreover, replications by the same maintainer involved an average modification ratio of 0.53, whereas those by different maintainers showed lower changes (0.26), suggesting substantially less modification among different maintainers.

Finding 6. *Replicated packages tend to reuse most of the original code: on average, 85.04% of the original files are carried over unchanged. When edits are made, changes are concentrated in a small subset of original files, with line-level modification ratios averaging 39%.*

4.2.4 Popularity of Replicated Packages. Finally, we investigated the popularity of replicated packages by analyzing download counts. Figure 3b shows the CDF for download counts. Out of the 544 replicated packages maintained by the same maintainer, 303 (55.70%) had more than 100,000 downloads, indicating high popularity. Notably, packages with more than 10,000 downloads accounted for 469 (86.21%). Packages replicated by different maintainers generally exhibited lower download counts; however, several achieved substantial popularity. Out of 817 such packages, 89 (10.89%) recorded more than 100,000 downloads, 365 (44.67%) surpassed 10,000 downloads, and 26 packages (3.18%) exceeded 1 million downloads (the most downloaded reached over 21 million).

Finding 7. *Replicated packages have also gained notable popularity in the Python ecosystem, with many recording over 10,000 downloads. This suggests that users may inadvertently install these packages, sometimes confusing them with the original ones, which in turn can introduce unnecessary maintenance challenges or even potential security risks.*



(a) Top 10 **CWE**s by vulnerability count. (b) **CVSS** distribution of vulnerabilities.

Fig. 4. Graphs showing the vulnerability types and severities of the identified replicated vulnerable packages.

5 Vulnerability Analysis (RQ2)

To understand the security implications of replication, we then examined how replicated packages overlap with known vulnerabilities. For broad identification of vulnerable replicated packages, we used the vulnerability dataset as a reference and compared it against the candidate dataset pool.

5.1 Status of Vulnerable Replications

By comparing 1,072 vulnerable packages (as a reference) with the candidate pool, we initially identified 543 replicated packages. However, some of the identified replications originated from patched versions rather than vulnerable ones. To refine the initial results, we applied a two-step refinement process. First, through version-based inspection, we verified whether the duplicated package versions fell within the vulnerable version range defined by OSV. Next, through code-level analysis, we manually examined whether the security patches had been applied by comparing the patch code with the corresponding reused code. Finally, we identified **256 packages** in which the vulnerable code had been directly replicated (93 highly and 163 partially replicated packages).

Ethical Disclosure. We prioritized verification starting from the most-downloaded packages and reported vulnerable packages in which the vulnerabilities could be triggered. Unlike malicious packages (see Section 6.2), however, our vulnerability reports to PyPI (or the corresponding team) often received no response, or in some cases, only an acknowledgment without a subsequent patch. To date, we have reported 10 vulnerabilities and received confirmation for only two cases. We plan to continue reporting vulnerabilities that require immediate attention on an ongoing basis.

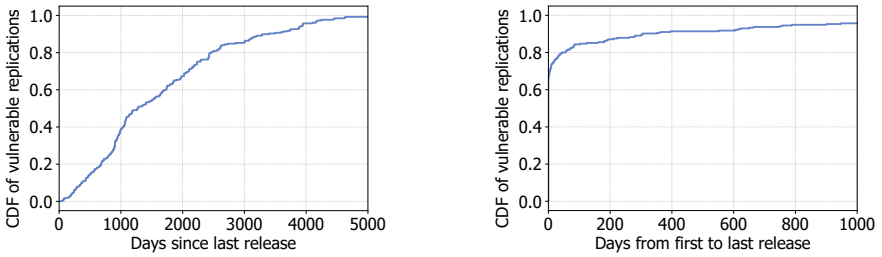
Finding 8. *From 1,072 vulnerable packages, we identified 256 replications that directly preserved the vulnerability, underscoring the security risks posed by vulnerable replications in the ecosystem.*

5.2 Characteristics of Replicated Vulnerable Packages

5.2.1 Vulnerability Types and Distribution. Among the 256 identified replications, we observed a total of 1,025 vulnerability occurrences and 383 unique vulnerabilities (comprising 370 CVE-listed vulnerabilities and 13 additional issues without CVE identifiers). On average, each package contained four vulnerabilities, indicating that replicated packages often inherited multiple flaws.

To analyze vulnerability types and severity, we use Common Weakness Enumeration (CWE) and Common Vulnerability Scoring System (CVSS). Several non-CVE vulnerabilities also provide this information through PYSEC and GHSA, thus, this decision enables a consistent analysis.

The distribution of vulnerabilities by type and severity is summarized in Figure 4. Regarding vulnerability types, input validation-related vulnerabilities (*e.g.*, CWE-20: Improper Input Validation and CWE-77: Command Injection) were the most common, as they frequently arise from developers mishandling untrusted user data. This was followed by access control failures (*e.g.*, CWE-863:



(a) Distribution of **days since last release** of vulnerable replication.

(b) Distribution of **days from first to last release** of vulnerable replication.

Fig. 5. CDF graphs of days since last release and days from first to last release.

Improper Authorization), which often result from insufficient restrictions on sensitive data. Redirection and deserialization issues (*e.g.*, CWE-601: Open Redirect and CWE-502: Deserialization of Untrusted Data) were also highly represented, particularly in Python due to the common use of built-in serialization libraries and the need for careful path handling.

In terms of severity, medium-severity vulnerabilities (511 cases) were the most prevalent, followed by high (233), critical (91), and low (87) severities. This finding suggests that replicated packages frequently contain vulnerabilities of substantial risk, underscoring the need for timely mitigation. In particular, high and critical vulnerabilities increase the possibility of exploitation and, if reused as dependencies, may propagate downstream, amplifying their impact.

Finding 9. *The 256 identified replication packages contained 383 unique vulnerabilities (370 CVEs), with input validation flaws being most common, followed by access control and data handling issues. In addition, we found that high- and critical-severity vulnerabilities account for 35.14%, which indicates that replicated vulnerable packages require more careful attention and mitigation.*

5.2.2 Maintenance Aspects of Replicated Vulnerable Packages. To assess the maintenance level of the replicated vulnerable packages, we analyzed their release activity. Figure 5 shows the maintenance status of vulnerable replications. Among the 256 vulnerable packages, 182 (71.09%) exhibited a release interval of less than one week between their initial release and the most recent update. In addition, 231 (90.23%) out of the 256 packages had not been updated for over a year as of September 2025. The fact that the last update of the replicated vulnerable packages dates back approximately 1,630 days on average (median 1,274 days) suggests that they are either poorly maintained or left unattended after release. Notably, 92 of these poorly maintained packages had more than 10,000 downloads, indicating that users continued to install them despite their outdated and insecure state.

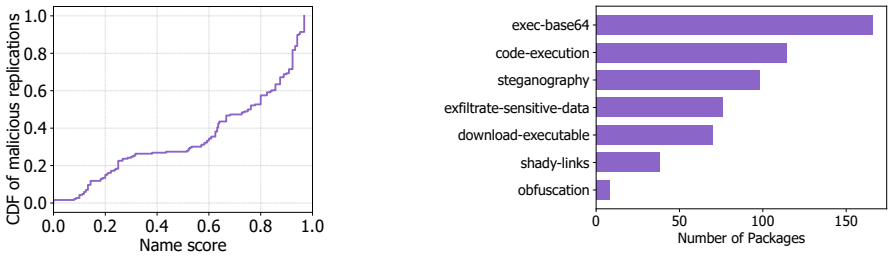
Finding 10. *Of the 256 vulnerable package replications, 231 (90.23%) were unmaintained for over a year, with an average of 1,630 days since their last update. This suggests that most of these packages remain effectively unmaintained.*

5.2.3 Detection Gaps in Existing Tools. Next, we evaluated the detectability of vulnerable replications identified by our replication-based approach. Specifically, we used `DependencyTrack` [8] and `Safety` [33], two widely adopted tools detecting vulnerable packages via dependency metadata.

Out of the 256 vulnerable replications identified, only four and one cases were detected by `Safety` and `DependencyTrack`, respectively. This discrepancy arises because dependency-based scanners rely on known package metadata, such as names and versions. Consequently, repackaged vulnerable code under different identifiers can evade detection. For example, Listing 1 shows a vulnerability propagated through replication (for ethical reasons, we show only the original CVE

Listing 1. Example of propagated vulnerable code (CVE-2025-48889).

```
def _copy_to_dir(self, dir: str) -> FileData:
  pathlib.Path(dir).mkdir(exist_ok=True)
  new_obj = dict(self)
  if not self.path:
    raise ValueError("Source file path is not set")
  new_name = shutil.copy(self.path, dir) # vulnerable sink
  new_obj["path"] = new_name
  return self.__class__(**new_obj)
```



(a) Name similarity between replicated malicious packages and their originals.

(b) Statistics of suspicious APIs added in replicated malicious packages.

Fig. 6. Graphs of name similarity in replicated malicious packages and statistics of suspicious APIs.

code). The flaw, originally reported in `Gradio`, enables arbitrary file copying and may lead to denial-of-service attacks. We found a replicated package containing the same vulnerable code without modification. Because it was published under a different name and versioning scheme, conventional dependency-based techniques failed to detect it. In contrast, our code-based replication analysis successfully uncovered such vulnerable replicas that existing approaches overlook.

Finding 11. *Dependency-based vulnerability scanners fail to detect replicated vulnerable packages because they rely on original package information. Replicated packages distribute the same vulnerable code under different names, creating invisible security risks.*

6 Malware Analysis (RQ3)

Finally, we examine how package replication contributes to malicious code distribution.

6.1 Analysis of Known Malware Exploiting Package Replication

6.1.1 Status of Replicated Malicious Packages. To examine how replication is exploited in real-world attacks, we compared the popularity dataset against the malware dataset (*i.e.*, 2,656 known malicious packages). We confirmed that 186 (4.79%) known malicious packages were replicated from popular packages: 125 highly and 61 partially replicated packages. Replicating code from popular packages increases the possibility of evading code-based malware detection and exacerbates developer confusion, potentially leading to inadvertent malware installation and broader propagation. Notably, most cases replicated over 90% of the original code, indicating minimal modification by attackers and highlighting package replication as a notable vector for malware distribution.

Finding 12. *186 (4.79%) of known malicious packages were replications of popular packages, a majority (67%) being highly replicated (≥90% code reuse), demonstrating that attackers exploit package replication as an observable distribution mechanism by making minimal modifications.*

6.1.2 Name and Metadata Similarity. Figure 6a shows the CDF of name similarities for replicated malicious packages. Notably, 88 (47.31%) out of 186 had a name similarity score of 0.8 or higher, often reflecting typosquatting patterns (*e.g.*, `jeilyfish` mimicking `jellyfish`). However, some malicious packages avoided typosquatting by replicating most of the code under entirely different names. For instance, `youtube-new` replicated `requests`, and `u283udsfru` replicated `certifi`. In such cases, name-based detection is ineffective, highlighting the necessity of code-based identification.

Next, we measured the metadata similarity. Because the description field was unavailable for known malicious packages, similarity was computed using the remaining four fields.

We observed that metadata replication was widespread. Among 186 cases, 159 (85.48%) shared at least one field with their originals. Of the 125 highly replicated packages, 108 (86.4%) duplicated at least one field and 66 (52.8%) duplicated three or more. Partially replicated packages showed a similar but weaker pattern, with 50 (81.96%) sharing at least one field and 32 (52.45%) sharing three or more. The most frequently copied field was `summary` (119), followed by `author_email` (109), `home_page` (99), and `author` (73). These results indicate that many malicious packages replicate not only code but also identifying metadata to enhance credibility or evade detection.

Finding 13. *Among the replicated malicious packages, 88 (47.31%) showed high name similarity (≥0.8) with their originals and 158 (84.94%) shared at least one metadata field, indicating that attackers frequently copy both names or metadata to mislead users and complicate detection.*

6.1.3 Suspicious API Injection. Next, we investigated the additional functionalities in replicated malicious packages by focusing on the modified and newly added files (see Section 2.4.1).

Our analysis revealed that 180 (96.77%) out of 186 replicated malicious packages incorporated new APIs absent from the original repositories. Specifically, we identified 796 unique APIs newly introduced across the 180 replicated malicious packages. These additions frequently appeared in combination, with multiple suspicious APIs embedded within a single line of code.

To conduct a clear analysis of these APIs, we leveraged `GuardDog`'s `Semgrep` ruleset [7]. Among the ruleset, seven categories of suspicious behaviors were detected.

(1) **`Shady links.`** Embedding or redirecting to suspicious or malicious external websites.
(2) **`Obfuscation.`** Hiding code logic through encoding, packing, or complex transformations.
(3) **`Download executable.`** Fetching and saving external binary files that may contain malware.
(4) **`Exfiltrate sensitive data.`** Collecting and transmitting private information.
(5) **`Steganography.`** Concealing malicious code or data within seemingly benign files or media.
(6) **`Code execution.`** Executing arbitrary or attacker-controlled code on the victim system.
(7) **`Exec-base64.`** Executing Base64-encoded payloads to bypass simple inspection.

Following their definitions, we examined additional suspicious APIs introduced in replicated malicious packages. As shown in Figure 6b, the most prevalent type was `exec-base64`, appearing 582 times across 166 packages, as it enables payload decoding and execution in a single step. `Code execution` APIs were added 217 times in 114 packages, allowing direct command or script execution. `Steganography` appeared 173 times in 98 packages, facilitating concealment of malicious content. We also identified APIs related to `exfiltrate sensitive data` (76 packages), `download executable` (70), `shady links` (38), and `obfuscation` (8).

Beyond single API insertion, some replicated malicious packages leveraged combinations of these suspicious behaviors. For example, 37 replicated packages were found to include APIs related to `steganography`, `code execution`, and `exec-base64` simultaneously.

Listing 2. Malicious code injected in the `_resolve_endpoint` function.

```
  resolve_request.endpoint_regional = request.endpoint_regional
+ try:
+   sessions = Session()
+   data = {"ak": self._ak, "secret": self._secret}
+   sessions.close()
+ except:
+   pass
  return self._endpoint_resolver.resolve(resolve_request)
```

Finding 14. *180 replicated malicious packages (96.77%) injected 796 unique suspicious APIs, with exec-base64 and code execution being most common, and these suspicious functionalities often appeared in combination, suggesting sophisticated malicious intent.*

6.1.4 Case Study. We present a case in which package replication was used to introduce malicious code. We identified `python-aliyun-sdk-core-2.13.36` as a highly replicated version of `aliyun-python-sdk-core-2.13.36`, with a code similarity score of 0.993. The package names were highly similar, and all four metadata fields were identical. Of the original package's 146 files, 144 were reused without modification, while two files, `aliyunsdkcore/client.py` and `setup.py`, were altered. In particular, malicious logic was injected into the `_resolve_endpoint` function in `client.py`, as shown in Listing 2. This function determines service endpoints based on region and product information. In the replicated version, the attacker introduced suspicious APIs related to `exfiltrate sensitive data` and `shady links`, attempting to leak access keys such as `ak` and `secret` to an external domain. This example illustrates a credential exfiltration pattern embedded within an otherwise near-identical replica of a legitimate SDK.

6.2 Replicated Malicious Package in the Wild

We applied our replication detection and suspicious API injection analysis to the recent dataset collected between May and August 2025, comparing newly uploaded packages against the popularity dataset. We used the recent dataset rather than the candidate dataset to enable early detection and timely response to potentially malicious code introduced through new package releases. Malware is often removed or modified shortly after upload, so recent data more accurately reflects attacker activity. Accordingly, prior Python malware-detection research has also focused on recently uploaded packages (*e.g.*, [9, 20]), rather than on randomly sampled data.

Malicious package detection was conducted following the approach described in Section 6.1.3. Specifically, we tracked recent packages that replicated popular packages and manually examined whether they contained APIs associated with malicious behavior.

From a recent dataset of 25,407 packages, we identified 227 (0.89%) that replicated popular packages. Among them, 67 contained previously identified suspicious APIs in newly added or modified files. We manually inspected these 67 cases, focusing on code regions that differed from the originals, and confirmed malicious behaviors such as data exfiltration and unauthorized downloads. As a result, we uncovered **seven previously unknown malicious packages**, all of which were removed from PyPI after our disclosure. One representative case, `zoz-requests`, replicated `requests` with minimal changes but injected code into the `request` function to redirect traffic through a hardcoded proxy and disable SSL verification, enabling man-in-the-middle attacks (see Listing 3). The package remained on PyPI for 123 days before removal, demonstrating how malicious replicas can evade detection by blending into trusted codebases.

Finding 15. *Our replication- and suspicious API–based analysis uncovered seven unknown malicious packages, confirming that replication remains an attack vector in recent uploads.*

Listing 3. Malicious code injected in the `request` function.

```
  def request(method, url, **kwargs):
    with sessions.Session() as session:
+     kwargs.setdefault("proxies", "http": "http://192.168.0.128:8080", "https":"http://192.168.0.128:8080")
+     kwargs.setdefault("verify", False) # Bypass SSL verification
      return session.request(method=method, url=url, **kwargs)
```

Table 5. Distribution of replicated package detection results by the similarity score.

Similarity score	0.0-0.1	0.1-0.2	0.2-0.3	0.3-0.4	0.4-0.5	0.5-0.6	0.6-0.7	0.7-0.8	0.8-0.9	0.9-1.0	Total
RQ1 (D1 vs. D5)	3,114	572	536	315	177	575	229	141	166	375	6,200
RQ2 (D2 vs. D5)	1,259	152	79	54	40	39	48	73	68	129	1,977
RQ3 (D1 vs. D4)	13	5	7	8	10	22	20	11	13	126	235

7 Discussion

7.1 Similarity Threshold Sensitivity

Table 5 summarizes the distribution of similarity scores across the RQs. For RQ2, we report the number of replicated packages that copied vulnerable versions. For RQ3, we exclude explicit false positives, such as cases involving only test file replication. As expected, lowering the threshold identifies more replicated packages but also introduces more false positives. Notably, the number of packages in the 0.9-1.0 range exceeds those in the 0.8-0.9 range, supporting 0.9 as a meaningful boundary for highly replicated packages. Similarly, the 0.5 threshold effectively separates partially replicated packages from lower-similarity cases in RQ1 and RQ3. Although both thresholds were empirically chosen, they can be adjusted based on analytical objectives (*e.g.*, a lower threshold may be preferred when recall is prioritized). Overall, the observed distributions suggest that our chosen thresholds provide a reasonable and well-balanced criterion for replication analysis.

7.2 Application

Application to Code Clone Detection. Our methodology for identifying packages that replicate most of the original code can be applied to code clone detection research in package manager ecosystems. In particular, our framework integrates embedding, clustering, code similarity, and name/metadata similarity rather than relying on any single signal, offering useful intuition for analyzing code reuse across various ecosystems, including Python.

Application to Vulnerability Detection. The discovery of vulnerable replications underscores the need for code-level vulnerability detection in PyPI. Prior 1-day vulnerability studies targeting package manager ecosystems relied on package names, versions, and dependency information. By contrast, vulnerability analysis for C/C++, where code-level reuse is more prevalent, has depended on direct code analysis (*e.g.*, [10, 44]). Our findings suggest that code-level analysis is equally necessary for PyPI, and provide a foundation for future work on vulnerable packages.

Application to Malware Detection. Our findings suggest a new research direction for detecting malicious packages that prior approaches have not fully addressed. Existing PyPI malware detection studies have focused on code shared across packages—identifying those with functionalities similar to known malware (*e.g.*, [11, 22]). In contrast, our findings highlight the importance of detecting malicious packages by focusing on subtle *differences* within highly similar packages. Statistical analysis of suspicious APIs can provide a basis for such detection and foster future research.

7.3 Replication in Open-Source Ecosystems

Open-source ecosystems encourage reuse and redistribution, and package replication can arise for legitimate reasons such as customization or compatibility maintenance. Our study does not treat

replication as inherently problematic. Rather, we focus on its security implications: replication can be exploited for malicious purposes, and it can also unintentionally preserve vulnerabilities when patches are not consistently propagated. By examining replication from a security perspective, we aim to understand how this common engineering practice may introduce risks at ecosystem scale.

7.4 Limitations

First, our detection pipeline focused exclusively on Python source files, excluding other components (*e.g.*, C extensions). Although manual inspection confirmed the presence of some vulnerabilities in such files, our automated replication detection was not applied beyond Python. Second, in some cases the accuracy of clone detection decreases due to the small size of packages and environment leakage (*e.g.*, site-packages included in distributions). Third, manual analysis remains essential for confirming both malicious and vulnerable packages. Although our framework aims to identify replication and inherited security risks, false negatives may arise due to limitations in embedding, clustering, or similarity thresholds, potentially leading to missed replications or undetected vulnerable variants. Lastly, the scope of this study is limited to vulnerabilities and malicious activities arising from package cloning. Other issues, such as vulnerabilities introduced through insecure coding practices, are out of the scope of this study.

7.5 Threats to Validity

First, our dataset represents approximately one-third of PyPI packages, which may not fully capture the characteristics of the entire Python ecosystem. Second, the distinction between vulnerable and patched versions can be subtle, as patches often introduce only minor code changes. Although we perform manual inspection to analyze the relevant code regions and assess whether vulnerability-inducing logic persists, the mere presence of vulnerable patterns does not guarantee exploitability in practice. Consequently, some flagged cases may not be practically exploitable under realistic attack conditions, and conversely, certain exploitable cases may remain undetected. Third, as our analysis focuses on replicated packages that reuse a substantial portion of an existing codebase, malicious variants that incorporate only small code fragments fall outside our scope. Nevertheless, our approach is complementary to existing techniques that detect malicious packages based on fine-grained code reuse or small injected snippets. Finally, parts of the study relied on manual inspection by experts. Although multiple reviewers were involved in the investigation process to enhance reliability, this manual component inevitably introduces subjectivity and potential inconsistencies. The interpretation of vulnerability persistence and suspicious behavior may vary between reviewers, potentially affecting the reproducibility and generality of our findings.

8 Related Work

Code Clone Detection. Prior research has mainly focused on detecting code clones, either from a general software engineering perspective or from a security perspective. Early studies developed scalable techniques to detect similar code fragments across large codebases (*e.g.*, [16, 27, 34]), while others aimed to identify vulnerable clones using pattern matching, machine learning, or function-level abstraction (*e.g.*, [15, 18, 21, 43, 44]). More recently, researchers have shown that code cloning is pervasive across ecosystems, with large-scale studies reporting high duplication rates in GitHub and persistent vulnerabilities in shrinkwrapped clones in npm (*e.g.*, [23, 45]). These studies attempted to identify small reused code fragments or to analyze packages from a vulnerability perspective. However, they did not closely investigate the phenomenon of large-scale code replication, nor did they attempt to apply it to vulnerable or malicious packages. Building on these insights, our work provides the first large-scale prevalence and security analysis of such

replication in the Python ecosystem, revealing how it creates security risks through vulnerability preservation and malicious code injection.

Security Issue Detection in Python. Previous research has explored various approaches to detecting malicious and vulnerable packages in PyPI. Early rule-based techniques combined static, dynamic, and metadata analysis (*e.g.*, [5, 9, 20]), demonstrating practical detection capabilities but remaining limited against novel or obfuscated attacks. To overcome these limitations, machine learning has been increasingly adopted, using code embeddings, behavioral modeling, and graph-based analysis to improve detection accuracy (*e.g.*, [11, 22, 35, 46]). Although they are effective, these methods require substantial training data and computational resources, and often focus on specific files (*e.g.*, `setup.py`) or predefined behavioral patterns. [39] compared package registry code against upstream repositories but they do not capture emerging malicious uploads. Typosquatting attacks have been widely studied in supply chain security through name-based similarity analysis (*e.g.*, [17, 28, 36, 40]). These approaches focus on lexical characteristics of package identifiers and do not examine code-level replication across packages, which is our focus. ML-based approaches have also been proposed for detecting vulnerabilities (*e.g.*, [26, 42, 47]), but remaining limited in capturing complex contextual patterns. Moreover, analysis of commit histories shows that vulnerability fixes are often delayed [2]. In contrast to these approaches, our work shifts the focus from detecting suspicious patterns within individual packages to analyzing replication across the ecosystem, revealing how vulnerabilities and malicious code propagate through cloned distributions.

9 Conclusion

As PyPI has become increasingly central in the software ecosystem, ensuring the security of published packages has emerged as a critical concern. To address this, we conducted a large-scale empirical study on package replication, a key factor affecting package security. We identified 1,361 replicated packages and 256 replicated vulnerable packages, and further uncovered seven previously unknown malicious packages that had emerged through package replication. Our findings highlight the characteristics and prevalence of package replication and, from a security perspective, show how it propagates vulnerabilities, enables suspicious modifications, and undermines ecosystem security. Our replication-based approach complements existing dependency- and metadata-driven tools, and in future work, we plan to develop security-oriented clone detection techniques and extend our pipeline beyond Python to other ecosystems.

Data Availability

Our code and results are available at https://github.com/sunha21/pypi-replication-analysis.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2025-00517788, Research on Intelligent SBOM Generation and Automated Vulnerability Analysis through Multi-level Code Analysis),and the Culture, Sports and Tourism R&D Program through the Korea Creative Content Agency grant funded by the Ministry of Culture, Sports and Tourism (International Collaborative Research and Global Talent Development for the Development of Copyright Management and Protection Technologies for Generative AI, RS-2024-00345025).

References

[1] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical Analysis of Security Vulnerabilities in Python Packages. *Empirical Software Engineering* 28, 3 (2023), 59. doi:10.1007/s10664-022-10278-4

[2] Gábor Antal, Márton Keleti, and Péter Hegedűs. 2020. Exploring the Security Awareness of the Python and JavaScript Open Source Communities. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 16–20.

[3] Ethan Bommarito and Michael Bommarito. 2019. An Empirical Analysis of the Python Package Index (PyPI). *arXiv preprint arXiv:1907.11073* (2019). doi:10.48550/arXiv.1907.11073

[4] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie Van Deursen. 2015. Tracking Known Security Vulnerabilities in Proprietary Software Systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 516–519. doi:10.1109/SANER.2015.7081868

[5] Seogyeong Cho, Seungeun Yu, and Seunghoon Woo. 2025. Cryptbara: Dependency-Guided Detection of Python Cryptographic API Misuses. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1578–1590. doi:10.1109/ASE63991.2025.00133

[6] Ctags. 2026. Universal Ctags. https://github.com/universal-ctags/ctags.

[7] Datadog. 2026. GuardDog: A CLI Tool to Identify Malicious Packages. https://github.com/DataDog/guarddog.

[8] DependencyTrack. 2026. DependencyTrack. https://github.com/DependencyTrack/dependency-track.

[9] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium, NDSS*. doi:10.14722/ndss.2021.23055

[10] Siyue Feng, Yueming Wu, Wenjie Xue, Sikui Pan, Deqing Zou, Yang Liu, and Hai Jin. 2024. FIRE: Combining Multi-Stage Filtering with Taint Analysis for Scalable Recurring Vulnerability Detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1867–1884. doi:10.5555/3698900.3699005

[11] Xingan Gao, Xiaobing Sun, Sicong Cao, Kaifeng Huang, Di Wu, Xiaolei Liu, Xingwei Lin, and Yang Xiang. 2025. MALGUARD: Towards Real-Time, Accurate, and Actionable Detection of Malicious Packages in PyPI Ecosystem. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security '25)*. doi:10.5555/3766078.3766322

[12] Google. 2025. OSV: A Distributed Vulnerability Database for Open Source. https://osv.dev/.

[13] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An Empirical Study of Malicious Code In PyPI Ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 166–177. doi:10.1109/ASE56229.2023.00135

[14] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. 2008. Code Reuse in Open Source Software. *Management science* 54, 1 (2008), 180–193. doi:10.1287/mnsc.1070.0748

[15] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 48–62. doi:10.1109/SP.2012.13

[16] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.

[17] Berkay Kaplan and Jingyu Qian. 2021. A Survey on Common Threats in npm and PyPi Registries. In *International Workshop on Deployable Machine Learning for Security Defense*. Springer, 132–156.

[18] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 595–614. doi:10.1109/SP.2017.62

[19] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2022. Taxonomy of Attacks on Open-Source Software Supply Chains. *arXiv preprint arXiv:2204.04008* (2022). doi:10.1109/SP46215.2023.10179304

[20] Ningke Li, Shenao Wang, Mingxi Feng, Kailong Wang, Meizhen Wang, and Haoyu Wang. 2023. MalWuKong: Towards Fast, Accurate, and Multilingual Detection of Malicious Code Poisoning in OSS Supply Chains. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1993–2005. doi:10.1109/ASE56229.2023.00073

[21] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32nd annual conference on computer security applications*. 201–213. doi:10.1145/2991079.2991102

[22] Wentao Liang, Xiang Ling, Jingzheng Wu, Tianyue Luo, and Yanjun Wu. 2023. A Needle is an Outlier in a Haystack: Hunting Malicious PyPI Packages with Code Clustering. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 307–318. doi:10.1109/ASE56229.2023.00085

[23] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: A Map of Code Duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28. doi:10.1145/3133908

[24] Leland McInnes, John Healy, and Steve Astels. 2017. hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software* 2, 11 (2017), 205. doi:10.21105/joss.00205

[25] Leland McInnes, John Healy, and James Melville. 2018. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv preprint arXiv:1802.03426* (2018). doi:10.21105/joss.00861

[26] Abdechakour Mechri, Mohamed Amine Ferrag, and Merouane Debbah. 2025. SecureQwen: Leveraging LLMs for Vulnerability Detection in Python Codebases. *Computers & Security* 148 (2025), 104151. doi:10.1016/j.cose.2024.104151
[27] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. 2021. NIL: Large-Scale Detection of Large-Variance Clones. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–841. doi:10.1145/3468264.3468564
[28] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. 2023. Beyond Typosquatting: An In-depth Look at Package Confusion. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3439–3456.
[29] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2136–2148. doi:10.1109/ICSE48619.2023.00180
[30] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. doi:10.1007/978-3-030-52683-2_2
[31] David Reid, Mahmoud Jahanshahi, and Audris Mockus. 2022. The Extent of Orphan Vulnerabilities from Code Reuse in Open Source Software. In *Proceedings of the 44th international conference on software engineering*. 2104–2115.
[32] David Reid, Kristiina Rahkema, and James Walden. 2023. Large Scale Study of Orphan Vulnerabilities in the Software Supply Chain. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*. 22–32. doi:10.1145/3617555.3617872
[33] Safety. 2025. Safety: Python Dependency Vulnerability Scanner. https://pypi.org/project/safety/.
[34] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th international conference on software engineering*. 1157–1168.
[35] Xiaobing Sun, Xingan Gao, Sicong Cao, Lili Bo, Xiaoxue Wu, and Kaifeng Huang. 2024. 1+1>2: Integrating Deep Code Behaviors with Metadata Features for Malicious PyPI Package Detection. In *Proceedings of the 39th IEEE/ACM international conference on automated software engineering*. 1159–1170. doi:10.1145/3691620.3695493
[36] Matthew Taylor, Ruturaj K. Vaidya, Drew Davidson, Lorenzo D Carli, and Vaibhav Rastogi. 2020. SpellBound: Defending Against Package Typosquatting. arXiv:2003.03471 [cs.SE] doi:10.48550/arXiv.2003.03471
[37] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 644–655.
[38] Hugo van Kemenade, Cal Paterson, Martin Thoma, Mike Fiedler, Richard Si, and Zsolt Dollenstein. 2025. hugovk/top-pypi-packages: Release 2025.08. Zenodo. doi:10.5281/zenodo.16672093
[39] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. LASTPYMILE: Identifying the Discrepancy between Sources and Packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 780–792.
[40] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and Combosquatting Attacks on the Python Ecosystem. In *2020 ieee european symposium on security and privacy workshops (euros&pw)*. IEEE, 509–514. doi:10.1109/EuroSPW51379.2020.00074
[41] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv preprint arXiv:2305.07922* (2023).
[42] Laura Wartschinski, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunske. 2022. VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python. In *Information and Software Technology*. Elsevier.
[43] Seunghoon Woo, Eunjin Choi, Heejo Lee, and Hakjoo Oh. 2023. V1SCAN: Discovering 1-day Vulnerabilities in Reused C/C++ Open-source Software Components Using Code Classification Techniques. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6541–6556. doi:10.5555/3620237.3620603
[44] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. 2022. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In *31st USENIX Security Symposium (USENIX Security 22)*. 3037–3053.
[45] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2022. What the Fork? Finding Hidden Code Clones in npm. In *Proceedings of the 44th international conference on software engineering*. 2415–2426. doi:10.1145/3510003.3510168
[46] Junan Zhang, Kaifeng Huang, Yiheng Huang, Bihuan Chen, Ruisi Wang, Chong Wang, and Xin Peng. 2025. Killing Two Birds with One Stone: Malicious Package Detection in NPM and PyPI using a Single Model of Malicious Behavior Sequence. *ACM Transactions on Software Engineering and Methodology* 34, 4 (2025), 1–28. doi:10.1145/3705304
[47] Kunpeng Zhao, Shuya Duan, Ge Qiu, Jinyuan Zhai, Mingze Li, and Long Liu. 2024. Python source code vulnerability detection based on CodeBERT language model. In *2024 7th International Conference on Algorithms, Computing and Artificial Intelligence (ACAI)*. IEEE, 1–6. doi:10.1109/ACAI63924.2024.10899694

Received 2026-02-25; accepted 2026-03-24