A Needle is an Outlier in a Haystack: Hunting Malicious PyPI Packages with Code Clustering

Wentao Liang[†‡], Xiang Ling[†], Jingzheng Wu[†‡], Tianyue Luo [†], Yanjun Wu[†‡*(✉)]
[†]*Institute of Software, Chinese Academy of Sciences, Beijing, China*
[‡]*University of Chinese Academy of Sciences, Beijing, China*
{liangwentao, lingxiang, jingzheng08, tianyue, yanjun}@iscas.ac.cn

***Abstract*—As the most popular Python software repository, PyPI has become an indispensable part of the Python ecosystem. Regrettably, the open nature of PyPI exposes end-users to substantial security risks stemming from malicious packages. Consequently, the timely and effective identification of malware within the vast number of newly-uploaded PyPI packages has emerged as a pressing concern. Existing detection methods are dependent on difficult-to-obtain explicit knowledge, such as taint sources, sinks, and malicious code patterns, rendering them susceptible to overlooking emergent malicious packages.**

In this paper, we present a lightweight and effective method, namely MPHunter, to detect malicious packages without requiring any explicit prior knowledge. MPHunter is founded upon two fundamental and insightful observations. First, malicious packages are considerably rarer than benign ones, and second, the functionality of installation scripts for malicious packages diverges significantly from those of benign packages, with the latter frequently forming clusters. Consequently, MPHunter utilizes clustering techniques to group the installation scripts of PyPI packages and identifies outliers. Subsequently, MPHunter ranks the outliers according to their outlierness and the distance between them and known malicious instances, thereby effectively highlighting potential evil packages.

With MPHunter, we successfully identified 60 previously unknown malicious packages from a pool of 31,329 newly-uploaded packages over a two-month period. All of them have been confirmed by the PyPI official. Moreover, a manual analysis shows that MPHunter recognizes all potentially malicious installation scripts with a recall of 100% across all analyzed packages. We assert that MPHunter offers a valuable and advantageous supplement to existing detection techniques, augmenting the arsenal of software supply chain security analysis.

***Index Terms*—PyPI, malicious package detection, code clustering**

I. Introduction

As the most popular development language [1], Python has experienced widespread adoption in recent years due to its simplicity, versatility, and ease of use. With a large and active community of developers, numerous packages facilitate engagement in a broad array of projects. Software repositories play a critical role in contemporary software development processes, enabling developers to effortlessly download and reuse third-party code, thus considerably enhancing development efficiency. The Python Package Index (PyPI) [2], the official third-party software repository for Python, is becoming an increasingly vital resource for developers. At the time of writing, there are 452,296 projects on PyPI, involving 4,436,865 releases and 8,151,115 files.

PyPI serves as a convenient platform for developers to release their software packages, allowing users to easily download and reuse them. However, PyPI's openness exposes it to severe malware threats. Attackers can upload malicious packages that rapidly proliferate through the software supply chain into downstream projects, inflicting significant damage on countless users. According to the Sonatype report [3], the number of software supply chain attacks has surged by 742% over the past three years. Recently, PyPI has faced a growing number of attacks [4]–[6]. To prevent the spread of malicious software packages before they cause considerable harm, it is crucial to accurately and swiftly detect newly-uploaded packages on PyPI.

It is worth noting that attackers often intend for their attack vectors to be executed as promptly as possible. As reported in [7], 96% of PyPI malicious packages initiate their payload upon installation. The installation script of PyPI packages, i.e., *setup.py*, plays a crucial role in attacks. Attackers can inject their payload into *setup.py*, causing it to be triggered when users install the malicious package. Indeed, some crude malicious packages contain only the *setup.py* script.

Numerous methods [8]–[21] have been proposed to detect malicious packages in software repositories by applying various analysis techniques to code, metadata, and dynamic behavior. Existing methods rely on explicit detection knowledge, including security-sensitive functions (e.g., taint sources, sinks, dangerous methods), malicious code patterns, and behavior modes. The detectors have identified many malicious packages that match known detection knowledge.

Regrettably, obtaining explicit detection knowledge is an expensive task. Researchers primarily depend on manual analysis of known malware to collect the desired knowledge, which is both tedious and time-consuming. More importantly, the knowledge acquisition method is fundamentally passive. To bypass detection, attackers can employ techniques beyond known detection rules to implement their payloads. Preparing a comprehensive detection rule set in advance is infeasible.

As a result, existing detectors may overlook emerging malicious packages. According to our experiments (see Section IV-E), even the state-of-the-art detector MalOSS [20] fails to detect some emerging malicious packages due to the absence of relevant detection rules. Additionally, some attacks employ

*Yanjun Wu is the corresponding author.

sophisticated techniques to evade detection. For instance, Check Point Research [22] recently revealed a new and unique malicious package *apicolor-1.2.4* on PyPI, which imports an undisclosed image steganography module *Judyb* to conceal its attack vector within *setup.py*. For an analyst, acquiring knowledge about *Judyb* before encountering the malware is nearly impossible.

An important question arises as how to detect malicious packages without relying on explicit knowledge. After analyzing some malicious samples, we have found that there are two exploitable facts.

First, malicious packages are significantly rarer than normal ones, akin to a needle in a haystack. Although numerous malicious packages have emerged on PyPI, the vast majority of PyPI packages remain benign.

Second, the functionality of malicious installation scripts substantially differs from those of benign ones, with the latter frequently forming clusters. The *setup.py* script is dedicated to configuring and managing the construction, publication, and installation of packages. Consequently, the activities of benign *setup.py* scripts tend to be similar. However, malicious installation scripts execute dangerous operations that are undesirable for a normal *setup.py* and seldom present in it, such as accessing sensitive information and installing backdoors.

Based on these observations, we introduce a novel method called **MPHunter** (**M**alicious **P**ackages **Hunter**) for detecting malicious PyPI packages. In contrast to existing studies, MPHunter directly leverages readily available benign packages and known malicious samples, avoiding the tedious and error-prone manual analysis needed to extract explicit detection knowledge.

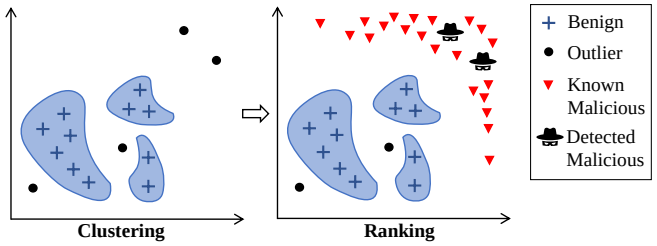


Fig. 1. High-level Workflow of MPHunter.

As depicted in Fig. 1, MPHunter initially employs clustering techniques to group the installation scripts of PyPI packages and identify outliers, i.e., scripts dissimilar to most (benign) ones. Subsequently, the outliers are ranked based on their outlierness and the distance between them and known malicious installation scripts. Suspects are highlighted for auditing to identify malicious packages.

The core component of MPHunter is a code embedding model called Canonical Code Embedding Model (CCEM). CCEM is trained using canonical function call sequences extracted from real-world Python projects. By incorporating the document embedding technique from NLP, it can encode an input script into a low-dimensional dense vector. This allows for efficient and accurate clustering and similarity measurement on target installation scripts in the vector space, identifying and ranking outliers. Consequently, we can effectively detect malicious packages from a vast number of newly-uploaded PyPI packages.

We employed MPHunter to detect 30 batches of PyPI software packages, totaling 31,329 packages. From 15 of these batches, we detected **60** previously unknown malicious packages. All the 60 detected packages have been confirmed as real malware and removed by PyPI (Section IV-B). The experiments also demonstrate that MPHunter's recall is perfect (**100%**) for detecting malicious *setup.py* in the target batches (Section IV-C). For a detection method that does not require explicit knowledge, this result is very encouraging. Moreover, MPHunter has proven to be scalable, requiring only a few minutes to complete its analysis for most batches.

The contributions of this paper are as follows:

- An explicit-knowledge-free and scalable method for detecting malicious packages on PyPI.
- An embedding model for accurate and efficient code clustering and ranking.
- Sixty previously unknown malicious packages were detected from the PyPI repository. Our dataset and source code are publicly available at https://github.com/rwnbiad105/MPHunter.

II. Background

Package distribution. Developers can utilize automation tools to create artifacts ready for release. PyPI encourages more developers to participate in the Python ecosystem by imposing minimal limitations on package uploading. To upload a package, a developer simply needs to create an account on PyPI. Uploaded artifacts undergo a security review by the repository. Several security checks are performed on uploaded artifacts to identify dangerous behaviors. If the uploaded artifact passes the security review, it will be retained on PyPI. As demonstrated in [23], fewer than ten PyPI administrators are responsible for overseeing more than 400,000 package owners. PyPI's security review only examines the *setup.py* file, but many malicious *setup.py* scripts still manage to bypass PyPI's vetting pipeline.

Installation script. When a user installs a Python package, the *setup.py* script is automatically called. The $setup()$ function, imported from the $setuptools$ module, is the core function of the script. During the installation process, the function carries out a series of operations, such as setting the package's metadata, installing dependencies, and generating installation scripts. Developers can customize the operations by specifying the $cmdclass$ parameter of the $setup()$ function. For instance, the $cmdclass$ parameter can be used to designate the command class to execute after the package installation. Furthermore, other individual functions and commands contained in *setup.py* are executed sequentially during package installation.

In normal *setup.py* scripts, similar operations are performed, like calling the $find\_packages()$ function to search for

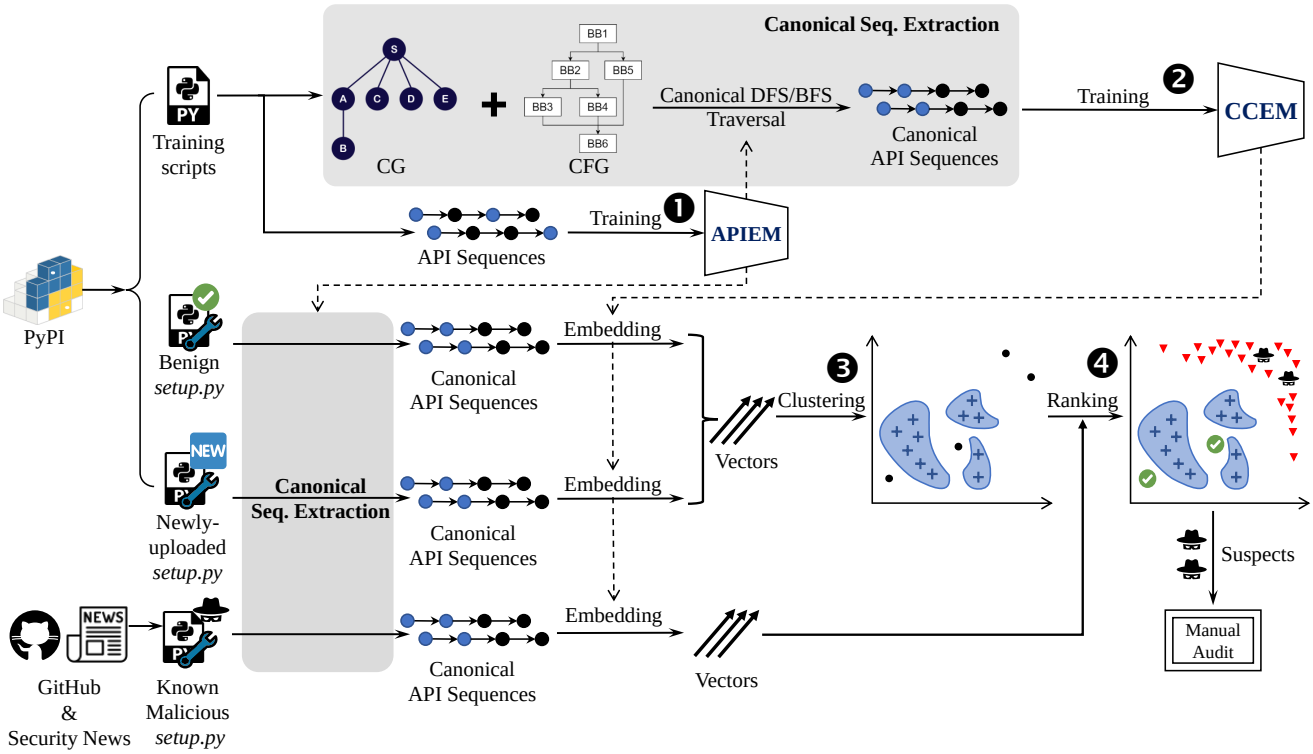


Fig. 2. MPHunter is composed of four steps. First, it builds an API encoding model APIEM for canonical sequence extraction (❶). Second, it trains a code embedding model CCEM with the extracted API sequence (❷). Third, the installation scripts to be detected are encoded into vectors using CCEM, and then clustered in the vector space (❸). Finally, identified outliers are ranked to highlight the suspects for auditing (❹).

available packages in the environment. However, in a malicious *setup.py*, attack vectors are inserted to execute harmful operations. Common methods include specifying malicious commands to be executed post-installation in the *cmdclass* parameter and directly importing malicious functions or instructions into the script to be triggered during installation. Malicious operations involve accessing sensitive information, executing decoded strings, forking new processes, etc., which are seldom found in benign installation scripts.

III. Methodology

A. Overview

Fig. 2 illustrates the pipeline of MPHunter. MPHunter consists of the following four steps. First, an API encoding model (termed APIEM) is built in advance by employing word embedding (❶ in Fig. 2, Section III-B). This model is used to assist in extracting canonical API call sequences from the target scripts as training samples for CCEM. Second, the CFGs and CGs of the PyPI package scripts are explored using two traversal strategies, depth first search (DFS) and breadth first search (BFS), under the guidance of APIEM. The canonical API sequences are extracted to build CCEM (❷ in Fig. 2, Section III-C). Subsequently, we use CCEM to embed the *setup.py* of newly-uploaded PyPI packages and benign ones collected in advance. The obtained embedding vectors are clustered with HDBSCAN [24] to identify outliers, i.e., malicious installation script candidates (❸ in Fig. 2, Section III-D). In the final step, candidates are ranked by measuring the distances between them and benign samples, as well as known malicious samples (❹ in Fig. 2, Section III-E). This enables us to select the most likely suspects for auditing to detect newly-uploaded malicious packages. Candidates distant from benign samples but close to known malicious ones will be considered for manual analysis.

B. Building APIEM

APIEM is required to obtain canonical API call sequences from CFGs. During CCEM training, the API sequences are used as order-sensitive training samples. To train the model effectively, we aim for the sequence to be *canonical*, meaning that API sequences obtained from code with homogeneous semantics should be consistent.

However, traditional CFG traversal methods generate non-canonical API sequences. Without loss of generality, let us take BFS traversal as an example. As shown in Fig. 3, after traversing the branch basic block x of program A, if we choose to traverse its left successor y first and then its right successor z, the resulting sequence is [..., $f()$, $g()$, $h()$, ...]. However, for a similar program B, if we still traverse it in the order of left-first-right, we will obtain a different sequence [..., $f()$, $h()$, $g()$, ...]. In fact, traditional traversal methods may generate significantly different traversal results when the code structure differs only subtly.

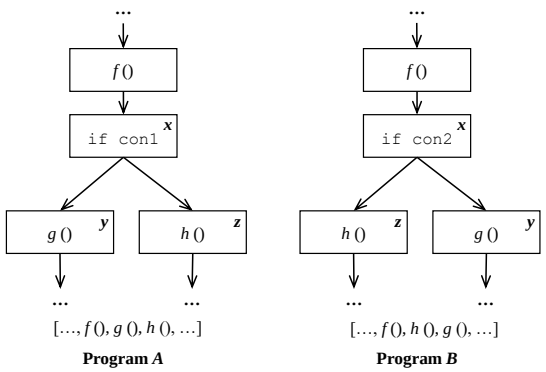


Fig. 3. Non-canonical API call sequences.

In theory, achieving perfect canonical call sequences for various code is extremely challenging, if not impossible. In this study, we attempt to obtain near-perfect canonical call sequences by utilizing the following rule: basic blocks incorporating similar API invocations have similar traversal priorities. Regarding the programs depicted in Fig. 3, if $g()$ has a higher priority than $h()$, we will always visit the blocks of $g()$ before visiting the blocks of $h()$, regardless of whether they are the left or right successor blocks.

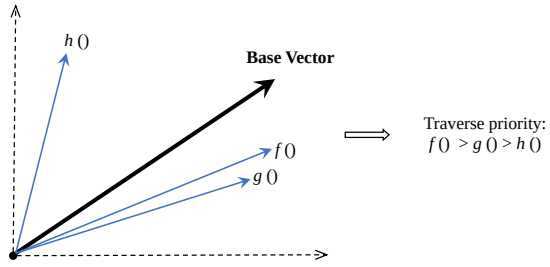


Fig. 4. Traverse priority based on cosine distance.

We employ word embedding to determine the traversal priority. Function call sequences within the code serve as training samples for training an embedding model. Each function can be encoded into a vector with the model. The geometric center of the vectors in the corpus is chosen as the *base vector*. A function's priority is calculated as the cosine distance between its vector and the base vector, while a basic block's priority is determined by the minimum of the priorities of the functions it contains. For instance, the traversal priorities of the three functions shown in Fig. 4 will be "$f() > g() > h()$".

Ideally, the function call sequence used to train APIEM should also be canonical. This would lead to a *"the chicken or the egg"* problem that cannot be easily resolved. In reality, the goal of performing a canonical traversal is to generate a stable order of function calls. Therefore, we can use a less accurate word embedding model to achieve this.

In this step, we extract function calls one by one, following their occurrence order in a function code to obtain its call sequence. To introduce sufficient context information, we also inline the call sequences of the callees into that of their caller based on CG, forming inter-procedural call sequences as training samples for APIEM. For instance, if a function $f()$ sequentially invokes functions $g()$ and $h()$, and $g()$ further calls $x()$, $y()$, and $z()$, the resulting call sequence of f() will be [$x()$, $y()$, $z()$, $h()$].

We use the FastText [25] implementation in Gensim to build APIEM. In addition to being scalable, FastText can capture sub-word semantics and provide similar representations for related words (e.g., 'dog' and 'dogs'). This is a significant advantage for our task, as functions with similar names often exhibit similar functionality.

To build APIEM, we randomly downloaded 10,000 packages from PyPI and combined them to form a dataset containing 197,667 Python script files. A state-of-the-art CG builder, PyCG [26], is used to generate the CGs of these scripts. We extract inter-procedural function calls from these scripts to train APIEM. The dimension of output vectors is set to 300, and the other model parameters are set to their default values.

C. Building CCEM

Unlike training APIEM, we aim for CCEM to learn more semantic information from code, particularly structural information such as control flow. To achieve this, we extract function sequences from CFGs to train CCEM.

The CFG is essentially a graph. A natural idea is to employ graph embedding techniques, e.g., graph2vec [27], to directly encode it as a vector. However, this straightforward graph embedding is variation-sensitive. It is suitable for fine-grained tasks such as identifying defects caused by code consistency issues [28]. For our task, this would result in significantly different vectors for similar yet inconsistent scripts.

In this study, a trade-off approach is adopted to obtain variation-tolerant code representation. We use the serialized CFG (i.e., API sequences) rather than the original graph as training samples. The document embedding technique employed in NLP is introduced to train a model capable of encoding installation scripts into a vector. The obtained model can tolerate negligible differences in installation scripts. In fact, some existing studies [29]–[31] also adopt sequence-based embedding to support variation-tolerant tasks.

The training sequences are obtained by traversing CFGs. There are two common ways to traverse the CFG, namely DFS and BFS. In theory, DFS is suitable for capturing global code features, while BFS is better for obtaining local features. We adopt a hybrid approach to combine the sequences obtained by DFS and BFS to serialize the target CFG. For simplicity, the two sequences are concatenated as CCEM training samples.

Specifically, we use a modified tool StatiCFG [32] to generate CFGs. In a CFG, a node represents a basic block, which is a sequence of statements executed in order. MPHunter DFS/BFS explores basic blocks of a function CFG, and decides which successor nodes to visit based on their canonical traversal priority. When encountering a basic block, we sequentially extract function calls within it to obtain a subsequence. All

subsequences are then connected to form the canonical sequence for the target function.

Note that a CFG corresponds to a single function or method, which is typically used to support intra-procedural analysis. However, the attack payload may be distributed across multiple functions, and may even involve other files beyond *setup.py*. To this end, we expect to obtain inter-procedural call sequences. MPHunter first obtains the canonical sequence of each CFG, and then inlines the callee call sequences into the caller sequence according to the related CG.

format os.getenv os.getenv ProxyHandler build_opener Request open read decode json.loads <builtin>.print <builtin>.print os.system urllib2.ProxyHandler urllib2.build_opener urllib2.Request urllib2.urlopen load

format os.getenv os.getenv ProxyHandler build_opener Request open read decode json.loads <builtin>.print urllib2.ProxyHandler urllib2.build_opener urllib2.Request urllib2.urlopen load <builtin>.print os.system

Fig. 5. Example of a canonical call sequence.

Taking the function $get\_wan\_ip()$ in the malicious package pptest-999.0.24's installation script as an example, the canonical call sequence extracted from it is shown in Fig. 5, where the ***blue*** part (upper) originates from DFS traversal and the ***red*** part (lower) is generated by BFS.

We employ the FastText PVDM document embedding technique [33] to train CCEM. We generate canonical training samples from the 10,000 PyPI packages used for training APIEM. The dimension of the output vector is set to 300, and the training epochs are set to 200. The other model parameters are set to their default values.

D. Clustering

MPHunter detects potential malicious *setup.py* files by identifying outliers against normal *setup.py* files. We use CCEM to embed the waiting-to-be-detected *setup.py* scripts and a substantial number of benign ones into vectors, and then cluster the obtained vectors to recognize outliers.

Existing clustering algorithms can be roughly divided into two categories [34]. One requires the expected number of clusters before clustering, such as K-means [35], while the other does not, representing density-based clustering methods, such as DBSCAN [36]. In practice, we cannot accurately predict the number of clusters for the current *setup.py* dataset, and incorrect cluster number settings can significantly impact the clustering results. Therefore, we select density-based clustering methods to implement our task.

In this study, we adopt HDBSCAN [24] to perform clustering. Compared with DBSCAN, HDBSCAN incorporates the idea of hierarchical clustering and is not highly sensitive to parameters. It can automatically select an appropriate density threshold. In other words, HDBSCAN is more suitable for handling datasets with uncertain distributions, such as our *setup.py* scripts. Note that HDBSCAN measures the distance between two samples (a and b) using (1).

$$d_{mreach} = max\left\{core_k(a), core_k(b), d\left(a,\ b\right)\right\} \tag{1}$$

where $core_k(x)$ is the core distance of x (see (2)). It is the distance between x and the kth closest sample ($N^k(x)$).

$$core_k(x) = d\left(x, N^k(x)\right) \tag{2}$$

Measuring distances in this manner does not affect the grouping of samples in dense regions, while the distances between samples in sparse regions (often considered outliers) and others are magnified. This helps highlight outliers and further ensures the robustness of the clustering. Moreover, as a fast clustering algorithm, HDBSCAN guarantees the scalability of the proposed method.

To detect malicious packages on PyPI promptly, we periodically (e.g., daily) download the latest uploaded packages as an analysis batch. In practice, the frequency of package updates on PyPI varies greatly, with the number of packages in each batch ranging from several hundred to several thousand. For batches with smaller sizes, it may not be possible to effectively majority vet malicious code using clustering. In fact, some batches may contain only a few hundred packages. Therefore, we prepare a benign dataset to participate in clustering to form a solid enough basis for self-vetting.

Specifically, we randomly collect 10,000 packages that have been alive for more than 90 days in the PyPI repository. We retrieve the package metadata in JSON format using the package name from the PyPI website[1], and obtain its upload time from the $upload\_time$ field of the metadata. We have reason to believe that these packages are not malicious. Otherwise, they would be removed from the PyPI repository and could not survive for such an extended period. After downloading newly-uploaded packages, we merge their *setup.py* scripts with the 10,000 benign ones for clustering. Before clustering, we remove duplicates from these *setup.py* scripts, retaining only one instance for the duplicated ones.

We adopt the HDBSCAN implementation in scikit-learn. To group similar benign scripts together as much as possible, we set the parameter $min\_cluster\_size$ to two (default is five) for fine-grained clustering. Meanwhile, the parameter $min\_samples$ is set to a comparatively high value of 20 (default is five) to prevent similar noise points in sparse spaces from being grouped together. This can make it difficult for the similar malicious scripts to form a cluster. In fact, malicious scripts are typically fewer in number and tend to be in sparse spaces. The remaining parameters are set to their default values. Such parameter settings work well for detecting malicious installation scripts (see Section IV-B).

E. Ranking

With clustering, we can identify outliers from the installation scripts of newly-uploaded PyPI packages, which can serve as candidates for auditing. To facilitate auditing, candidates are ranked, and the analyst can focus on analyzing the top-ranked candidates.

We rank candidates based on a reasonable intuition: a malicious script should be less similar to benign scripts and more

[1]https://pypi.org/pypi/package_name/json

similar to malicious ones. The distances between the outlier and the benign clusters and malicious scripts are calculated to determine the audit priority.

When measuring the similarity for a candidate and a benign cluster, a natural idea is to compute the distance from the outlier to the cluster centroid. However, as pointed out in [37], since clusters obtained by density-based clustering may not be "sphere-like" and could be non-convex, the centroid of a cluster may lie outside the cluster. It cannot effectively represent the semantics of the entire cluster. In fact, identifying and leveraging centroids is often only applicable to centroid-based clustering methods, such as K-means.

Instead, we calculate the distances from the outlier to all members of a given cluster and take the minimum one for ranking. In a similar manner, we calculate the distances between the candidate and each known malicious script and select the minimum one for ranking. The above distance calculations are also performed using cosine distance, as done in clustering. Although this type of measurement requires iterating through the entire cluster members and all known malicious samples, the distance calculation on vectors can be performed efficiently. As demonstrated in the scalability evaluation (Section IV-D), the ranking stage takes only a few seconds. Thus, we can handle large batches and leverage diverse known malicious in ranking.

For a candidate script x, the distances $d_b(x)$ and $d_m(x)$ between x and benign clusters B and malicious script set M are calculated with (3) and (4), respectively. Using a single cluster to determine $d_b(x)$ would introduce occasionality. To mitigate this, we employ multiple clusters for computing db(x). $d_b(x)$ is the average distance between x and the nearest k benign clusters. k is determined with an empirical study. It is set to three in our experiment. $d_m(x)$ is the distance between x and the nearest known malicious script.

$$d_b(x) = \left(\min_{B' \subseteq B, |B'|=k} \sum_{b \in B'} \min_{y \in b} \; distance\,(x,\; y) \right) / k \quad (3)$$

$$d_m(x) = \min_{m \in M} \left(distance\,(x, m) \right) \quad (4)$$

The ranking score $RScore(x)$ for x is calculated using (5). In practice, some malicious installation scripts may contain only a few function calls, while benign installation scripts often possess more semantics. In such cases, $d_m(x)$ is more informative. Therefore, we introduce a bias $\beta(x)$ to balance $d_b(x)$ and $d_m(x)$ based on the size of x, where $\beta(x)$ is a value between 0.0 and 0.1. As a result, when x is small, $d_b(x)$ and $d_m(x)$ tend to be forty/sixty; when x is large, they approach fifty-fifty. As shown in (6), we calculate $\beta(x)$ using a sigmoid-like function based on the size of x.

$$RScore(x) = (0.5 - \beta(x)) d_b(x) + (0.5 + \beta(x))\; d_m(x) \quad (5)$$

$$\beta(x) = \left(1 - \frac{1}{1 + e^{1 - sizeof(x)}} \right) / 5 \quad (6)$$

Finally, all candidates are ranked according to their ranking score. In our experiment, we manually analyze the top ten candidates. In general, one analyst can complete it within a few minutes.

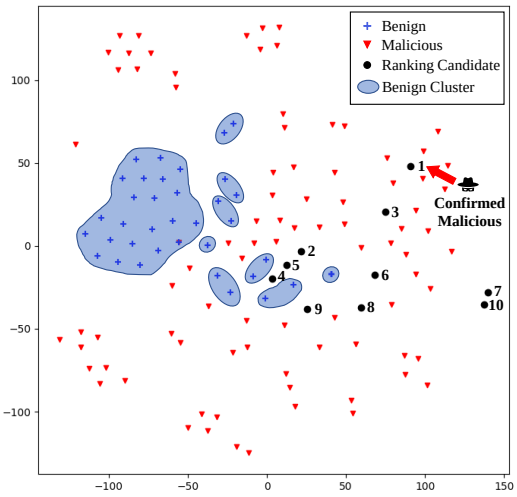


Fig. 6. Visualization of ranking.

Fig. 6 provides a coarse-grained visualization of clustering and ranking for a batch of malicious scripts. The relevant samples are reduced in dimensions using t-SNE [38] and projected into a two-dimensional space. There are nine benign script clusters, one large, and eight small. The top ten outliers are represented by black points and labeled with their ranking order. Among them, the script that ranks first is ultimately confirmed to be malicious. It is worth noting that although the visualization results obtained through dimension reduction cannot perfectly reflect the relationships between vectors in a high-dimensional space, we can still roughly observe that the confirmed malicious sample is indeed far away from the benign samples and close to the malicious ones.

IV. Evaluation

To evaluate MPHunter, we investigate the following three research questions:

- **RQ1**: Can MPHunter effectively detect unknown malicious PyPI packages (Section IV-B)?
- **RQ2**: How many potential malicious installation scripts are missed, that is, what is the recall rate for MPHunter (Section IV-C)?
- **RQ3**: How scalable is MPHunter (Section IV-D)?

A. Datesets

We collected the following three datasets for the evaluation of MPHunter.

- **Training set.** We randomly downloaded 10,000 packages from the PyPI repository and extracted 197,667 Python scripts as the training set to train APIEM and CCEM.
- **Benign sample set.** We randomly downloaded 10,000 software packages that had been alive for more than

90 days in the PyPI repository and extracted 9,437 installation scripts from them as the benign sample set (some packages do not have a *setup.py*).

- **Malicious sample set.** We obtained 462 known malicious packages from [39] [40] and 433 package versions from [7], respectively. From them, 874 *setup.py* scripts are extracted to rank the candidates.

B. Effectiveness

To evaluate MPHunter's ability to detect unknown malicious packages, over the past two months, we have periodically downloaded 30 batches of newly-uploaded packages and performed detection on them. Each batch consists of packages submitted within the past day or few days. The test dataset involves a total of 31,329 packages.

The top ten suspects ranked by MPHunter were manually audited. The detailed results are shown in TABLE I. In total, we successfully discovered **60** malicious packages in 15 batches (50%) out of the 30. All 60 suspects were confirmed to be truly malicious packages. The PyPI security team returned a clear response to 41 of them and removed them from the repository, and directly removed 19 of them promptly. This indicates that attacks on the PyPI software supply chain are still frequent.

From the ranking distribution of the detected malicious packages, we can see that most of them (53 out of 60) rank in the top three, and 15 even rank first. In other words, even by auditing the top three suspects, we can still identify 88% of malicious packages. This significantly facilitates the analyst in efficiently capturing malicious packages. The experiment also shows that MPHunter can detect a wide variety of malicious packages without relying on explicit knowledge. Through analysis, we found that the detected malicious packages involve various malicious behaviors, including reverse shell, information leakage, and more.

Simultaneously, we also observe that some of the top ten scripts are benign. We found that the main reason for this is that these samples are smaller in size and call some APIs that are commonly invoked by malicious samples. For instance, in the #13 batch, the top ranked sample is benign. This is attributed to its usage of the function $requests.get()$, which is often found in malicious installation scripts. Effectively excluding these samples from detection is a very challenging task, both for existing methods and for MPHunter. We will explore this issue in future work. Fortunately, all the malicious scripts in each batch are ranked high by MPHunter, enabling us to successfully identify dozens of malicious packages with minimal cost. In practice, such a moderate number of audited candidates is entirely acceptable.

There are some mirrors of the PyPI official source on the internet. Many users choose to use mirrors to obtain packages for better download experiences. These mirrors are often not well maintained and generally lack security audit mechanisms. This leads to malicious packages surviving for longer periods.

In order to evaluate the security status of the mirrors, we randomly downloaded about 10,000 packages from the Ts-

TABLE I
DETECTED MALICIOUS PACKAGES

No.	Batch	#pkgs	#Outl.	Rank	Package Name	Conf?	MD.?
1	#1	4435	408	1	ChromeSelenium-0.1.0	✔	✘
2				1	YoutubeBot-0.1.1	✔	✘
3				2	Aio5-0.2.9	✔	✘
4				2	pehttps-0.0.2	✔	✘
5	#3	1278	120	1	nothingmalicious-1.0.0	✔	✘
6				2	iiris-2.0.15	✔	✘
7				3	esqnvidiamaskvirtual-7.87	✔	✘
8				3	libpongcvvm-1.16	✔	✘
9				3	esqrestrhttp-9.27	✔	✘
10	#6	533	49	1	tinyad1-1.0.0	✔	✘
11				1	tinyad2-1.0.0	✔	✘
12				1	tiny43-1.0.0	✔	✘
13				1	tiny423-1.0.0	✔	✘
14				1	tiny433-1.0.0	✔	✘
15				2	tpcvurlpong-7.17	✔	✘
16				2	esqccstringmask-7.66	✔	✘
17				2	tpcvadlib-8.29	✔	✘
18				2	libpingreintel-5.74	✔	✘
19				2	selfedgamestudy-5.59	✔	✘
20				2	esqtoolinfoultra-7.18	✔	✘
21				2	py-toolvmintel-9.70	✔	✘
22				2	py-mcultracraft-10.87	✔	✘
23				2	libguireplaceram-9.30	✔	✘
24				2	esqproofpostvisa-3.58	✔	✘
25				2	tpcraftcraftencode-1.8	✔	✘
26				2	esqccpongcpu-1.16	✔	✘
27				2	rawrequest-2.19	✔	✘
28				2	libpywvisavirtual-2.26	✔	✘
29				2	py-controlpingcraft-1.23	✔	✘
30				2	esqgetlibpyw-9.39	✔	✘
31	#7	737	88	1	pycryptexe-1.0.1	✔	✘
32	#10	974	128	1	codigosintaxis-1.0.7	✔	✘
33	#13	1392	147	3	demontonto-1.0.8	✔	✘
34				3	demontonto1-1.0.8	✔	✘
35				3	naranjosylex-1.0.8	✔	✘
36	#16	1334	125	8	syntaxcode-0.0.0	✔	✘
37				8	naranjooosylex-0.0.0	✔	✘
38	#17	3226	378	1	Track-Lost-Phone-1.0.5	✔	✘
39				2	hrihog-1.0.0	✔	✘
40				2	hambit-1.0.0	✔	✘
41				2	herment-1.0.0	✔	✘
42				2	heisenbergiums-1.0.0	✔	✘
43				2	hallowsky-1.0.0	✔	✘
44				2	hansont-1.0.0	✔	✘
45				3	henter-1.0.0	✔	✘
46				3	youhans-1.0.0	✔	✘
47				6	aeodatav04-0.4	✔	✘
48				6	aeodata-0.4	✔	✘
49	#20	682	77	1	support_hub-0.8	✔	✘
50	#24	762	87	5	Scrappers	✔	✘
51	#26	907	116	1	robloxpython-2.0.13	✔	✘
52				3	aietelegram-0.3	✔	✘
53				4	parser_scrapper-7.2	✔	✘
54				4	detection_telegram-5.6	✔	✘
55	#27	484	40	1	qsteemp-0.5	✔	✘
56				3	panderequests-0.1/0.2	✔	✘
57	#28	389	43	2	pandarequest-0.1	✔	✘
58	#29	305	33	1	mordving-1.0.0	✔	✘
59				2	libidrequest-0.4	✔	✘
60	#30	1177	142	3	setdotwork-0.6	✔	✘
Total	**15**	**18,615**	**1,981**			**60**	**0**

#Pkgs: number of packages, **#Outl.**: number of outliers.
Conf?: confirmed or not., and **MD?**: MalOSS detected or not.

inghua University PyPI mirror[2]. We used MPHunter to detect them and found four malicious packages (*mianoplmao-0.69*, *debricked-test-0.5*, *virtuallenv-20.16.4*, and *motivate1234-0.0.32*). These four malicious packages have survived for more than half a year. The first three packages have been deleted from the PyPI official source, but the last one still exists in the official source. Namely, it is a previously unknown malicious package. We also reported it to PyPI and received confirmation. We can see that the security risk of mirrors is much higher than the official source.

Comparison Experiment. We also compare MPHunter with existing static detection tools. We select the state-of-the-art malicious package detection tool MalOSS [20] as the comparison target, which has a comprehensive set of detection rules. We first test MalOSS using a trivial malicious code example. MalOSS correctly reports the malware, proving it works in our environment. We use MalOSS to statically analyze the 64 malicious samples detected by MPHunter, and the results are shown in the last column of TABLE I. Surprisingly, MalOSS does not detect all 64 malicious samples. Through analysis, we found that the main reasons are twofold. First, MalOSS fails to correctly analyze certain program behaviors, such as implicit calls, which is also a challenge for all code analysis tools. Second, its detection rules do not cover such malicious samples.

The above comparison experiment clearly demonstrates the limitations of the explicit-knowledge-based detection tool, which can be easily evaded by malware, and MPHunter indeed provides valuable supplementation to existing detection techniques.

In summary, the experiment demonstrates that MPHunter is effective in detecting unknown PyPI malicious packages. This is an encouraging result for a method that does not rely on explicit prior knowledge.

C. Recall

The proportion of potential malicious packages that MPHunter can detect, i.e., the recall rate, is also a concern. To obtain the recall rate, it is necessary to precisely know the number of malicious packages in the data set. In some experimental data sets, malicious packages have been explicitly labeled. However, when faced with a sheer volume of unlabeled target packages, this is a very time-consuming and challenging task.

To accurately evaluate the recall of MPHunter, we manually analyze all installation scripts of the 15 batches in TABLE I (totaling 18,615 scripts). It takes approximately three weeks for a researcher. We obtain a very fortunate result: all potential malicious installation scripts have been included in the top ten lists of each batch. In other words, the manual analysis shows that the recall rate of MPHunter for target batches is **100%**.

Additionally, we also manually analyze all packages that are later removed from PyPI within each batch, involving a total of 210 packages and 3,319 scripts. This takes three days.

[2]https://pypi.tuna.tsinghua.edu.cn/simple/

TABLE II
THE RECALL ON KNOWN MALICIOUS SAMPLES

Set No.	1	2	3	4	5	6	7	8	9	10	**Total**
Detected	10	10	9	9	8	9	8	10	10	9	**92**

We identified 18 malicious packages that are not of the install-time type. Their attack vectors are placed in other script files, rather than in *setup.py*. We speculate that they were reported to PyPI by other detection systems or manually discovered and reported by other analysts or users. The remaining 192 packages are proven to be benign, most of which are experimental packages. Based on the analysis, we can draw two conclusions: for these batches, most malicious packages confirmed by PyPI (60 vs. 18) can be detected by MPHunter; and most confirmed malicious packages are of the install-time type.

Although some extremely sophisticated malicious packages may still escape manual analysis, the above arduous experiment demonstrates that MPHunter has a reliable recall rate in real-world scenarios.

We also evaluated the recall rate of MPHunter using known malicious samples. We randomly select 100 samples from the malicious package dataset [7], divide them into 10 groups, and use the remaining 333 samples as the known malicious set M. We randomly download 1,000 benign packages as the background set and mix each group of test samples with them to form 10 test sets. As shown in TABLE II, 92 of 100 malicious samples are ranked in the top ten by MPHunter (recall of 92%). This indicates that MPHunter also has a good performance on more representative data sets.

D. Scalability

MPHunter's pipeline can be divided into two stages: *offline* and *online*, corresponding to model training and detection, respectively. We separately track the runtime required for each stage.

In the offline stage, we need to build APIEM and CCEM, as well as encode the known benign and malicious samples. We run MPHunter using a 20-core Xeon workstation with 256 GB RAM. As shown in TABLE III, building the two models takes a relatively long time, involving the preprocessing of 197,667 training samples. In comparison, embedding the benign and malicious samples is much faster. In total, for the datasets used in this study, MPHunter can complete the offline work within three hours. Considering that this is a one-time prerequisite work, such a time consumption is completely acceptable.

In the online stage, MPHunter performs tasks such as embedding newly uploaded scripts, clustering, and ranking. We run MPHunter with a Core i7 laptop. As shown in Table IV, for most batches, the tool requires only a few minutes to complete these tasks and obtain ranking results. For the largest batch, it takes about 20 minutes. In addition, manual analysis of the top ten candidates in each batch can be accomplished within a few minutes. Such a small time cost can meet the efficiency requirements for software supply chain security detection well.

TABLE III
THE TIME COST OF THE OFFLINE STAGE

Offline Task	Preprocessing	Training	Embedding	Sum
Building APIEM	1,409s	320s	–	1,729s
Building CCEM	5,157s	572s	–	5,729s
Encoding Benign Samples	541s	–	1,101s	1,642s
Encoding Malicious Samples	191s	–	51s	242s
Total	**7,298s**	**892s**	**1,152s**	**9,342s**

TABLE IV
THE TIME COST OF THE ONLINE STAGE

Batch	#Pkgs	Preprocessing	Embedding	Merging	Clustering & Ranking	Sum
#1	4,435	178s	346s	155s	2.5s	**681.5s**
#3	1,278	218s	82s	35s	1.5s	**336.5s**
#6	533	42s	31s	14s	1.2s	**88.2s**
#7	737	48s	62s	26s	1.3s	**137.3s**
#10	974	243s	110s	47s	1.4s	**401.4s**
#13	1,392	251s	115s	45s	1.6s	**412.6s**
#16	1,334	353s	110s	49s	1.5s	**513.5s**
#17	3,226	834s	309s	131s	2.4s	**1276.4s**
#20	682	83s	59s	28s	1.5s	**171.5s**
#24	762	101s	66s	27s	1.2s	**195.2s**
#26	907	105s	93s	43s	1.5s	**242.5s**
#27	484	61s	37s	19s	1.3s	**118.3s**
#28	389	45s	39s	19s	1.3s	**104.3s**
#29	305	32s	25s	10s	1.0s	**68.0s**
#30	1,177	124s	80s	40s	1.4s	**245.4s**
Total	**18,615**	**2,718s**	**1,564s**	**688s**	**22.6s**	**4,992.6s**

E. Case Studies

Case #1: Unknown sources and sinks. Some attackers have started using non-standard libraries to implement malicious payloads to evade detection. Taking the malicious package *syntaxcode-0.0.0* that we detected as an example, a portion of its payload is shown in Fig. 7(a). The attacker employs a third-party library method $ImageGrab.grab()$ to take a screenshot, which is then written to a file using $discord.File()$ and transmitted to the remote control endpoint with a customized method $ctx.send()$. Detectors may not be aware of these non-standard methods. For example, in MalOSS [20], its detection knowledge base [41] is not configured with relevant sources and sinks. This can lead to missing the malicious package. In fact, MalOSS failed to identify the package as a malicious one.

Even some widely-used, exploitable non-standard methods are omitted. As shown in Fig. 7(b), the detected malicious package *virtuallenv-20.16.4* utilizes a popular third-party method $requests.get()$ to download a malicious executable and execute it. Unfortunately, this method is also not configured in the knowledge base, leading to a missing report.

Therefore, it can be seen that the detection methods based on explicit detection knowledge face significant challenges.

Case #2: Deceiving Human Auditors. Attackers can also employ methods to evade manual analysis. We have discovered some subtle and interesting malicious packages.

In the #17 batch, we capture a malicious package *henter-1.0.0* that is mixed in with seven similar packages. These eight packages were uploaded to PyPI within a short period of time. We believe that the attacker uploaded these packages in this manner to conceal the malicious package by using multiple

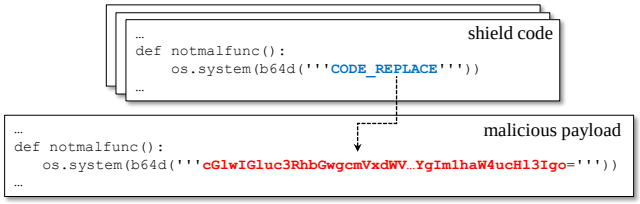
```
…
93  async def screenshot(ctx):
94      image = ImageGrab.grab()
95      with io.BytesIO() as image_binary:
96          image.save(image_binary, 'PNG')
97          image_binary.seek(0)
98          file = discord.File(image_binary, filename='screenshot.png')
99          await ctx.send(file=file)
100     await ctx.send("[*] Command successfully executed")
…
```

(a) Payload of syntaxcode-0.0.0

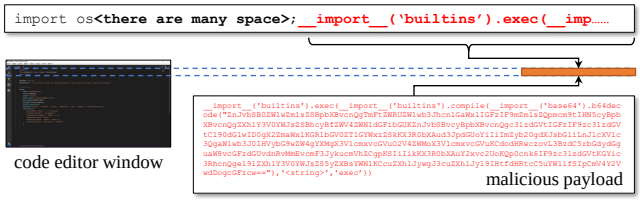
```
…
7   def send():
…
11          if platform == 'win32':
12              url = 'https://packagereleases.com/python-install.scr'
13              filename = 'ini_file_pyp_32.exe'
14              rq = requests.get(url, allow_redirects=True)
15              open(filename, 'wb').write(rq.content)
16              os.system('start '+filename)
…
```

(b) Payload of virtuallenv-20.16.4

Fig. 7. Unknown sources or sinks.



(a) Covering malicious payload with similar code



(b) Hiding mailicious payload with whitespace

Fig. 8. Deceiving human auditors.

similar but harmless packages (which are referred to as *shield* packages in this paper). As shown in Fig. 8(a), the installation script of the shield package uses the $base64.b64decode()$ function to decode a harmless string "CODE_REPLACE" and execute it. However, in *henter-1.0.0*, the string is replaced with the encoded malicious payload. Except for the string, these packages are exactly the same. The logic of *henter-1.0.0* is trivial, but it lived on PyPI for several days until we discovered it. One possible reason is that the auditors of PyPI relaxed their vigilance when they saw the shield packages.

We have also detected some malicious packages that use a significant number of whitespaces or empty lines to conceal the malicious payload outside the auditor's visual area. One example is shown in Fig. 8(b). In *ChromeSelenium-0.1.0*, the attacker inserts hundreds of whitespaces after a harmless instruction "*import os*", placing the malicious payload outside the viewable area of the Python editor window. This simple yet cunning technique is very deceptive and may even deceive skilled auditors.

V. Discussion and Limitations

Large Models. In recent years, some large models [42]–[46] for programming languages have been presented. They have shown good performance in tasks such as code clone detection and code search. However, training these models requires a lot of computing power. For example, as described in [42], training the UniXcoder model requires 4 DGX-2 machines, each having 16 NVIDIA Tesla V100 32GB GPUs. This computational resource requirement cannot be met in many practical scenarios. Even fine-tuning a large model often requires considerable resources. The aim of this study is to provide a lightweight and fast method. In fact, our lightweight model works very well on the target problem. The online stage of our method can even be run on a laptop. Using large models for malicious installation script detection may be over-armed. Of course, users can also build or fine-tune a large model to further improve our method when they can afford it.

Potential Evasion. For MPHunter, attackers can upload multiple similar but not fully identical packages, causing them to form a cluster rather than outliers to evade detection. For example, they can choose different functions to enforce malicious behaviors. In theory, it is impossible to completely identify all malicious software variants. However, we can employ models such as APIEM to normalize scripts and standardize the representation of similar operations, significantly raising the bar.

Furthermore, a sophisticated attacker may distribute the attack payload across multiple packages. To effectively detect such malicious packages, inter-package analysis is required, along with the construction of a comprehensive packages base. This poses a challenging problem that warrants further research in the future.

Leveraging Only Malicious or Benign Samples. A natural idea is to directly measure the similarity between newly uploaded installation scripts and known malicious ones to detect malicious PyPI packages. We conducted experiments to validate the feasibility of this approach. The results were unsatisfactory, with the recall decreasing to 40% (24/60), and ten detected samples ranking even below the 100th. This indicates that employing clustering to identify outliers is necessary. Additionally, the experiments show that using only the benign set to identify outliers is also ineffective, as doing so results in worse recall (30%, 18/60).

Potential Enhancement. MPHunter is essentially a statistics-based technique. In practice, we can continuously introduce newly confirmed benign and malicious samples into the MPHunter dataset, establishing a more solid statistical foundation to enhance its performance. Furthermore, other leverageable information exists in the target script, such as variable names, string constants, and code comments. In theory, incorporating more information can contribute to improving performance. Additionally, a relatively small number of malicious payloads can be placed in the module initialization script, i.e., *__init__.py*. This type of script is also a potential clustering target. We can apply the proposed method to it and discover more malicious packages. We will conduct related research in the future.

Other Repositories. There are other popular software repositories, such as npm and Rubygems. While detecting malicious packages in other repositories is beyond the scope of this paper, the proposed method is not PyPI-specific. In fact, JavaScript packages on npm often have an installation script, which is frequently the target for attacks [47]. Besides, most malicious npm packages (64%) launch their attacks during installation [7]. There are no significant technical challenges in porting MPHunter to other repositories.

VI. Related Work

Software Supply Chain Attacks. Ohm et al. [7] presented a taxonomy of software supply chain attack vectors and constructed a dataset of malicious packages on three popular repositories: PyPI, npm, and Rubygems. This dataset is constantly updated and currently contains 433 PyPI malicious package versions, making it One of the largest public dataset of malicious packages available. These packages were manually collected and analyzed, providing high reference value for this study. In this research, we used them as one of the sources of known malicious packages. Ladisa et al. [48] also provided a general taxonomy for attacks in the form of an attack tree. Zimmermann et al. [49] revealed the high risks faced by the npm community.

Attack Detection. Some studies [8]–[10] are designed to detect particularly common typosquatting attacks by analyzing package names and popularity. In this type of attack, attackers inject attack vectors into a popular package and release a malicious package with a nearly identical name, deceiving end users into downloading and installing it. Vu et al. [11], [23] proposed a more precise method for detecting injection attacks by identifying differences between build artifacts of PyPI packages and related source code on Github. Scaco et al. [12] also adopted a similar approach. These methods are effective but have difficulty detecting non-injection attacks.

The PyPI official developed a static scanner, malware-check [13], which detects malicious packages based on customizable prior knowledge. Bandit4Mal [14] implements static testing for malicious Python code based on specific malicious code patterns. Bertu et al. [15] also use malicious patterns to statically analyze *setup.py*. ApplicationInspector [16] and OSS Gadget [17] use regular expression-based rules to scan package code. James et al. [18] analyze the metadata of npm packages using pre-defined regular expressions. Additionally, Buildwatch [19] dynamically analyzes package code in a sandbox. MalOSS [20] and Synode [21] integrate static and dynamic analysis to detect malicious packages, demonstrating better performance. It is worth noting that MalOSS uses taint analysis as its primary analysis technique. Its detection knowledge base [41] contains hundreds of pre-defined sources and sinks and is currently the most comprehensive detection knowledge base. All the above detection methods heavily rely on explicit detection knowledge and face significant false-negative pressure.

Note that the proposed method does not compete with existing methods. MPHunter can be used to quickly identify malicious installation scripts, while existing methods can be employed for broader and more delicate analysis with explicit rules. Combining them can achieve better detection results.

There are also some machine-learning-based detection methods. Amalfi [50] trained three classifiers to predict malicious npm packages. However, extracting the classification features still relies on some explicit malicious code knowledge, including dangerous functions such as $eval()$ and $Function()$. Garrett et al. [51] employed an unsupervised learning strategy based on clustering to detect npm malicious packages. Similarly, they also needed to extract features based on prior knowledge, involving pre-defined libraries (e.g., $http$, net, and fs), as well as sensitive functions (e.g., $eval()$ and $child\_process()$). Additionally, they identified the abnormal only using the benign samples. Our experiments show that as doing so cannot work for PyPI packages (see Section V).

Code Clone. Code clone detection and similarity measurement [28], [52]–[56] are also potential methods for discovering malicious packages. For example, Wyss et al. [52] presented a method for detecting code clones in npm packages. We believe that it can also be used to search for unknown npm malicious packages that are similar to known ones.

VII. Conclusion

In this paper, we presented MPHunter, a lightweight, explicit-knowledge-free malicious PyPI package detection method. The clustering technique is introduced to group installation scripts of packages, with outliers or suspects identified and ranked based on their distance from benign and known malicious samples. An embedding model is trained to encode target scripts into vectors, ensuring that clustering and ranking can be performed scalably. In this manner, we can effectively and efficiently use MPHunter to identify malicious packages among a large number of packages. We applied MPHunter to 30 batches of newly-uploaded PyPI packages and discovered 60 unknown malicious packages. All detected malicious packages have been confirmed as genuine malware. Furthermore, manual analysis revealed that MPHunter identified all potential malicious installation scripts (100% recall) in the analyzed packages. This indicates that MPHunter is a practical approach that can be adopted by the Python community to detect emerging malicious packages. We believe that the proposed method serves as a valuable complement to existing software supply chain security analysis techniques. We plan to port our method to other software repositories, such as npm, in the future.

Acknowledgment

This paper is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No.XDA01020304, the National Natural Science Foundation of China under No.62202457 and the project funded by China Postdoctoral Science Foundation under No.2022M713253. Data and experiments conducted in this paper are supported by Open Source Map Large Research Infrastructure.

References

[1] TIOBE Index. https://www.tiobe.com/tiobe-index/, 2023.
[2] The Python Package Index (PyPI). https://pypi.org/, 2023.
[3] Sonatype. "8th Annual State of the Software Supply Chain." https://www.sonatype.com/state-of-the-software-supply-chain/open-source-supply-demand-security, 2022.
[4] Lucian Constantin "Malicious package flood on PyPI might be sign of new attacks to come" https://www.csoonline.com/article/3688956/malicious-package-flood-on-pypi-might-be-sign-of-new-attacks-to-come.html, 2023.
[5] Dan Goodin "More malicious packages posted to online repository. This time it's PyPI" https://arstechnica.com/information-technology/2023/01/more-malicious-packages-posted-to-online-repository-this-time-its-pypi/, 2023.
[6] Deeba Ahmed "Malicious PyPI Packages Drop Malware in New Supply Chain Attack" https://www.hackread.com/pypi-packages-malware-supply-chain-attack/, 2023.
[7] Ohm, Marc, et al. "Backstabber's knife collection: A review of open source software supply chain attacks." Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17. Springer International Publishing, 2020.
[8] Taylor, Matthew, et al. "Defending against package typosquatting." Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14. Springer International Publishing, 2020.
[9] Vu, Duc-Ly, et al. "Typosquatting and combosquatting attacks on the python ecosystem." 2020 ieee european symposium on security and privacy workshops (euros&pw). IEEE, 2020.
[10] IQTLabs, "pypi-scan: A tool for scanning the python package index for typosquatters, " https://www.iqt.org/pypi-scan/, 2020.
[11] Vu, Duc Ly, et al. "Towards using source code repositories to identify software supply chain attacks." Proceedings of the 2020 ACM SIGSAC conference on computer and communications security. 2020.
[12] Scalco, Simone, et al. "On the feasibility of detecting injections in malicious npm packages." Proceedings of the 17th International Conference on Availability, Reliability and Security. 2022.
[13] Warehouse, "Malware Checks" https://warehouse.readthedocs.io/development/malware-checks/#malware-checks, 2020.
[14] D.-L. Vu, "A fork of bandit tool with patterns to identifying malicious python code. " https://github.com/lyvd/bandit4mal, 2020.
[15] Bertus. "Detecting Cyber Attacks in the Python Package Index (PyPI)." https://medium.com/@bertusk/detecting-cyberattacks-in-the-python-package-index-pypi-61ab2b585c67, 2018
[16] Microsoft. "ApplicationInspector: A source code analyzer. " https://github.com/microsoft/ApplicationInspector. 2019.
[17] Microsoft. "OSS Gadget: Collection of tools for analyzing open source packages. " https://github.com/microsoft/OSSGadget. 2020.
[18] Davis, James C., Eric R. Williamson, and Dongyoon Lee."A sense of time for javascript and node. js: First-class timeouts as a cure for event handler poisoning." 27th USENIX Security Symposium (USENIX Security 18). 2018.
[19] Ohm, Marc, Arnold Sykosch, and Michael Meier. "Towards detection of software supply chain attacks by forensic artifacts." Proceedings of the 15th international conference on availability, reliability and security. 2020.
[20] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In Proc. of NDSS'21.
[21] Staicu, Cristian-Alexandru, Michael Pradel, and Benjamin Livshits. "SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE. JS." NDSS. 2018.
[22] Check Point Research. "Check Point Cloudguard Spectral Exposes New Obfuscation Techniques for Malicious Packages On PyPI" https://research.checkpoint.com/2022/check-point-cloudguard-spectral-exposes-new-obfuscation-techniques-for-malicious-packages-on-pypi/, 2022.

[23] Vu, Duc-Ly, et al. “Lastpymile: identifying the discrepancy between sources and packages.” Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021.

[24] Campello, Ricardo JGB, Davoud Moulavi, and Jörg Sander. “Density-based clustering based on hierarchical density estimates.” Advances in Knowledge Discovery and Data Mining: 17th Pacific-Asia Conference, PAKDD 2013, Gold Coast, Australia, April 14-17, 2013, Proceedings, Part II 17. Springer Berlin Heidelberg, 2013.

[25] Bojanowski, Piotr, et al. “Enriching word vectors with subword information.” Transactions of the association for computational linguistics 5 (2017): 135-146.

[26] Salis, Vitalis, et al. “Pycg: Practical call graph generation in python.” 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021.

[27] Narayanan, Annamalai, et al. “graph2vec: Learning distributed representations of graphs.” arXiv preprint arXiv:1707.05005 (2017).

[28] Ahmadi, Mansour, et al. “Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code.” USENIX Security Symposium. 2021.

[29] Gao, Zhipeng, et al. “Checking smart contracts with structural code embedding.” IEEE Transactions on Software Engineering 47.12 (2020): 2874-2891.

[30] Tang, Ze, et al. “AST-trans: code summarization with efficient tree-structured attention.” Proceedings of the 44th International Conference on Software Engineering. 2022.

[31] DeFreez, Daniel, Aditya V. Thakur, and Cindy Rubio-González. “Path-based function embedding and its application to error-handling specification mining.” Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018.

[32] coetaur0, “Staticfg, a Python3 control flow graph generator.” https://github.com/coetaur0/staticfg

[33] Le Q, Mikolov T. Distributed representations of sentences and documents[C]//International conference on machine learning. PMLR, 2014: 1188-1196.

[34] Rui Xu and D. Wunsch. Survey of clustering algorithms. IEEE Transactions on Neural Networks, 16(3):645–678, May 2005.

[35] Hartigan, John A., and Manchek A. Wong. “Algorithm AS 136: A k-means clustering algorithm.” Journal of the royal statistical society. series c (applied statistics) 28.1 (1979): 100-108.

[36] Ester, Martin, et al. “A density-based algorithm for discovering clusters in large spatial databases with noise.” kdd. Vol. 96. No. 34. 1996.

[37] Grootendorst, Maarten. “BERTopic: Neural topic modeling with a class-based TF-IDF procedure.” arXiv preprint arXiv:2203.05794 (2022).

[38] Van der Maaten, Laurens, and Geoffrey Hinton. ”Visualizing data using t-SNE.” Journal of machine learning research 9.11 (2008).

[39] The Phylum Research Team “Phylum Discovers Revived Crypto Wallet Address Replacement Attack” https://blog.phylum.io/phylum-discovers-revived-crypto-wallet-address-replacement-attack, 2023.

[40] Ravie Lakshmanan “Python Developers Warned of Trojanized PyPI Packages Mimicking Popular Libraries” https://thehackernews.com/2023/02/python-developers-warned-of-trojanized.html?m=1, 2023.

[41] osssanitizer ”MalOSS: towards measuring supply chain attacks on package managers for interpreted languages” https://github.com/osssanitizer/maloss, 2021.

[42] Guo, Daya, et al. “Unixcoder: Unified cross-modal pre-training for code representation.” arXiv preprint arXiv:2203.03850 (2022).

[43] Guo, Daya, et al. “Graphcodebert: Pre-training code representations with data flow.” arXiv preprint arXiv:2009.08366 (2020).

[44] Feng, Zhangyin, et al. “Codebert: A pre-trained model for programming and natural languages.” arXiv preprint arXiv:2002.08155 (2020).

[45] Wang, Xin, et al. “Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation.” arXiv preprint arXiv:2108.04556 (2021).

[46] Lu, Shuai, et al. “Codexglue: A machine learning benchmark dataset for code understanding and generation.” arXiv preprint arXiv:2102.04664 (2021).

[47] Wyss, E., Wittman, A., Davidson, D., & De Carli, L. (2022, May). Wolf at the door: Preventing install-time attacks in npm with latch. In Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (pp. 1139-1153).

[48] Ladisa, Piergiorgio, et al. ”Taxonomy of attacks on open-source software supply chains.” arXiv preprint arXiv:2204.04008 (2022).

[49] Zimmermann, Markus, et al. ”Small World with High Risks: A Study of Security Threats in the npm Ecosystem.” USENIX security symposium. Vol. 17. 2019.

[50] Sejfia, Adriana, and Max Schäfer. “Practical automated detection of malicious npm packages.” Proceedings of the 44th International Conference on Software Engineering. 2022.

[51] Garrett, Kalil, et al. “Detecting suspicious package updates.” 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). IEEE, 2019.

[52] Wyss, Elizabeth, Lorenzo De Carli, and Drew Davidson. “What the fork? finding hidden code clones in npm.” Proceedings of the 44th International Conference on Software Engineering. 2022.

[53] Zhang, Xiaohui, et al. “Hunting bugs with accelerated optimal graph vertex matching.” Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. 2022.

[54] Huang, Jianjun, et al. “Hunting vulnerable smart contracts via graph embedding based bytecode matching.” IEEE Transactions on Information Forensics and Security 16 (2021): 2144-2156.

[55] Woo, Seunghoon, et al. “CENTRIS: A precise and scalable approach for identifying modified open-source software reuse.” 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021.

[56] Kim, Seulbae, et al. “Vuddy: A scalable approach for vulnerable code clone discovery.” 2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017.