Typosquatting and Combosquatting Attacks on the Python Ecosystem

Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci
University of Trento, Italy
{ducly.vu, ivan.pashchenko, fabio.massacci}@unitn.it

Henrik Plate, Antonino Sabetta
SAP Security Research, France
{henrik.plate, antonino.sabetta}@sap.com

***Abstract*—Limited automated controls integrated into the Python Package Index (PyPI) package uploading process make PyPI an attractive target for attackers to trick developers into using malicious packages. Several times this goal has been achieved via the combosquatting and typosquatting attacks when attackers give malicious packages similar names to already existing legitimate ones. In this paper, we study the attacks, identify potential attack targets, and propose an approach to identify combosquatting and typosquatting package names automatically. The approach might serve as a basis for an automated system that ensures the security of the packages uploaded and distributed via PyPI.**

***Index Terms*—FOSS, Malicious Software, Supply Chain Attacks, Combosquatting, Typosquatting, Python, PyPI**

I. Introduction

Python Package Index (PyPI) provides a comfortable and widely used way to distribute Python projects users. However, this ease of use comes at a cost: PyPI has been leveraged to spread malware [1]. For example, the Slovak National Security Office[1] reported 10 cases where malicious code was embedded into the installation script to steal users' data. Perica [2] showed that many packages in PyPI contain executables that may include malicious payload triggered by users.

Limited automated controls integrated into the PyPI package publishing system and a small number of administrators prevents the security verification of every package. Hence, attackers can repackage the others package code into a new package with a malicious payload, and trick users into installing it. Several studies demonstrated PyPI vulnerability to the squatting attacks.

To demonstrate the ability to register typosquatting packages Tschacher [1] uploaded artificial packages with the names nearly identical to the legitimate packages, to the three different software repositories (including PyPI), and received 45K downloads over several months. Stagg [3] crafted and uploaded 12 packages that have names of the modules of Python standard library (e.g., `os`, `csv`) and observed a massive number of downloads of these packages (>490K downloads per year). Hence, squatting package names could be an attractive way to introduce malicious packages in PyPI.

Considering the ever-growing popularity of PyPI, there is a significant need for controls capable to automatically find malicious packages hosted in PyPI and prevent attackers from uploading new malicious packages. Hence, in this paper, we provide the following contributions:

- a study of the common attacks to craft malicious packages and trick users into downloading them,
- an approach for automatic identification of packages likely used in combosquatting and typosquatting attacks.

Following the motivating study of Stagg [3], we checked whether any PyPI packages have the same name as any of the 297 module names of the Python standard library.[2] We identified 62 such packages. Our manual analysis of these packages suggested that they are kept in PyPI mostly for backporting reasons.[3]

Table I shows that attackers apply different modifications to package names, however, these names remain similar to the original packages. Therefore, we use the Levenshtein distance [4] as the simplest and widely used technique to calculate the edit distance between package name strings. Our empirical results suggest that 79 933 packages are safe to use, and 67 005 packages require to be further investigated.

II. How PyPI works

PyPI is a popular repository of Python applications or packages: as of February 2020, it contains more than 216K packages with the total number of downloads exceeding four billion times. PyPI is maintained by a group of developers called Python Packaging Authority (PyPA for short). Figure 1 provides an overview of different roles envisioned by PyPI: End Users, Package Owners, PyPI Moderators (PyPA), and PyPI Administrators (PyPA).

End Users provide the name of a package to a package manager tool, like *pip*[5] to install the package from PyPI. Although *pip* does everything automatically for installing a package, it neither requires user authentication nor performs any validation of the package. Instead, *pip* merely looks for the package in PyPI by its name, identifies and resolves its dependencies, downloads all the required components, and installs them on the End User's computer.

[1]https://www.nbu.gov.sk/skcsirt-sa-20170909-pypi/

[2]We checked against all modules appeared in at least one of the existing Python standard libraries: Python 2.6, 2.7, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, and 3.8.

[3]The manual analysis of 62 packages available at https://github.com/vuduclyunitn/wacco_2020

[4]Data collected from pypistats.com on Feb 15, 2020

[5]https://pip.pypa.io/en/stable/

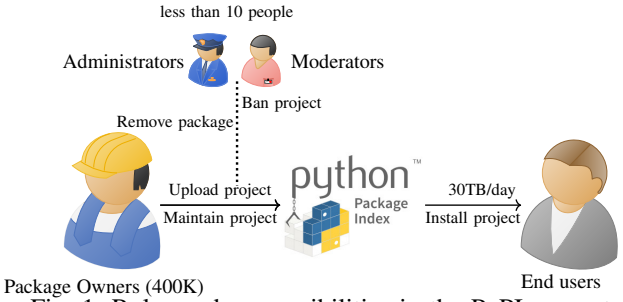

Fig. 1: Roles and responsibilities in the PyPI ecosystem.[4]

Package Owners can distribute code on PyPI using the tool called *setuptools*[6] that packs the original source code and generates a local distribution that is either in a source[7] or built[8] format. Package Owners register a package name on PyPI and publish the distribution artifacts of the package. If Package Owners have provided both distribution types, *pip* prefers to install the built distribution first. Publishing a package on PyPI is restricted to Package Owers, who can later modify (e.g., update a new version) existing packages that they have access.

PyPI Administrators and Moderators have exclusive rights to ban or revoke packages of Package Owners. For example, if a particular package is reported as malware, the Administrators will delete the package from PyPI and block the malicious developer. The Administrators can delete the corrupted package and support the Package Owners in recovering their access if a package owner credentials are compromised or damaged.

Although this scheme proved to support high latency for both Package Owners and End Users, limited resources and automated controls integrated into the package uploading, and distribution process leave the room for attackers to use PyPI for spreading malicious software. PyPI especially becomes an attractive target for attackers, considering the certain unbalance concerning the number of Package Owners and PyPA developers (40K to 1) and continuously growing popularity of PyPI. In the next sections, we will give an overview of the common strategies that attackers use to craft malicious packages and exploit the PyPI package distribution procedure to deliver malicious packages to End Users (Section III).

III. Crafting and Shipping Malicious Packages

In the wild, attackers use mostly two ways to spread malicious code within the PyPI ecosystem:

- steal the legitimate package owner's credentials of an existing package, inject a malicious payload into it so that users in their normal activities can unintentionally download it (e.g., install or update a package);
- create a new package with built-in malicious payload and trick users into downloading it (e.g., by squatting the name of a popular package).

Stealing PyPI credentials. In the first approach, attackers exploit End Users' trust in an already existing package. After the attacker obtained the Package Owner's rights to perform specific operations in the publishing process, they can upload their crafted package as a new (malicious) version of a compromised package. The `ssh-decorate` package (Table I) was affected by such an attack when attackers injected a malicious functionality for sending users' SSH credentials to a remote server.[9] Although this attack requires additional effort (e.g., social engineering) to obtain the credentials of package owners, a certain number of infections are still possible.

Tricking users into downloading malicious packages. In the second approach, attackers create a new package that, by design, features some malicious behavior. Attackers can craft the package from scratch or fork an existing PyPI package. In the latter case, the attackers injected a malicious dependency by modifying the *setup.py* installation script of the package so that the malicious dependency will be downloaded and silently installed along with the benign package (e.g., `acqusition` (#11), `urlib3` (#8) attacks in Table I). Then, they follow a regular procedure to create an account in PyPI, register a new package name, and upload the malicious package.

To increase the chance of getting more infections, attackers register package names that are similar to existing (usually popular) packages by *package typosquatting* or *package combosquatting* ([5],[6]) in which they split the package name into elements based on the "hyphen" character, and rearrange the elements, e.g., "python-nmap" into "nmap-python". Users who mistype or confuse the package name will install the malicious package instead of the legitimate one.

Given a large number of package owners w.r.t. the number of administrators, this likely to be (and so far has been) undetected because several legitimate packages whose names very close to other legitimate packages. For example, there is a PyPI package called `cpython` that has the same name as the Github project "cpython".[10] The package, however, has a very vague description and uses a different source code repository.

Table I shows several past combosquatting and typosquatting attacks in PyPI. Lutoma[11] detected two malicious packages; one substituted the 'l' character with the capital 'I' so that it is quite tricky to distinguish between `jeIlyfish` and `jellyfish`. At the time of its detection, the package had been downloaded 119 times. Another malicious package exploited the difference between the package naming practices established in two python versions Python 2 and Python 3 (`python3-dateutil` vs. `python-dateutil`) to confuse users when selecting the package of a particular Python version. These packages were used to steal users' information and send them to a remote server. Attackers prefer to delete characters from the legitimate packages to generate squatting names. For example, the Slovak National Security identified ten PyPI packages[12] (e.g., `acqusition`, `urllib`) that sent

[6]https://github.com/pypa/setuptools
[7]https://packaging.python.org/glossary/#term-source-distribution-or-sdist
[8]https://packaging.python.org/glossary/#term-built-distribution

[9]https://medium.com/@bertusk/cryptocurrency-clipboard-hijacker-discovered-in-pypi-repository-b66b8a534a8
[10]"cpython" is a compiler or an interpreter, not a third-party package
[11]https://github.com/dateutil/dateutil/issues/984
[12]https://www.nbu.gov.sk/skcsirt-sa-20170909-pypi/

TABLE I: Malicious packages in our sample.

#	Time Appear	Malicious Package	Legitimate package	Names change	Levenshtein distance d d=1	d=2
1	2016-03-02	virtualnv	virtualenv	Delete ‘e’	✓	
2	2016-03-03	mumpy	numpy	Substitute ‘n’ by ‘m’	✓	
3	2017-05-01	crypt	crypto	Delete ‘o’	✓	
4	2017-06-02	django-server	django-server-guardian-api	Delete “-guardian-api”		
5	2017-06-02	pwd	pwdhash.py	Delete “hash.py”		
6	2017-06-02	setuptool	setuptools	Delete ‘s’	✓	
7	2017-06-02	setup-tools	setuptools	Insert ‘-’	✓	
8	2017-06-02	telnet	telnetsrvlib	Delete “srvlib”		
9	2017-06-02	urlib3	urllib3	Delete ‘l’	✓	
10	2017-06-02	urllib	urllib3	Delete ‘3’	✓	
11	2017-06-03	acqusition	acquisition	Delete ‘i’	✓	
12	2017-06-03	apidev-coop	apidev-coop_cms	Delete “_cms”		
13	2017-06-04	bzip	bz2file	Substitute “2file” by “ip”		
14	2017-11-23	djanga	django	Substitute ‘a’ by ‘o’	✓	
15	2017-11-24	easyinstall	easy_install	Delete ‘_’	✓	
16	2017-12-05	colourama	colorama	Delete ‘u’	✓	
17	2018-04-25	openvc	opencv-python	Swap ‘c’ and ‘v’ & Delete “-python”		
18	2018-05-02	mateplotlib	matplotlib	Insert ‘e’	✓	
19	2018-05-02	numipy	numpy	Insert ‘i’	✓	
20	2018-05-02	python-mysql	MySQL-python	Swap “python” and “mysql”		✓
21	2018-05-03	libcurl	pycurl	Substitute “py” by “lib”		
22	2018-05-03	libhtml5	html5lib	Swap “html5” and “lib”		
23	2018-05-03	pysprak	pyspark	Swap ‘a’ and ‘r’		✓
24	2018-05-03	PyYMAL	pyyaml	Swap ‘a’ and ‘m’		✓
25	2018-05-10	nmap-python	python-nmap	Swap “nmap” and “python”		✓
26	2018-05-10	python-mongo	pymongodb	Delete “db” & Substitute “py” by “python-”		
27	2018-05-10	python-openssl	openssl-python	Swap “openssl” and “python”		✓
28	2018-09-17	pytz3-dev	pytz	Insert “3-dev”		
29	2018-10-29	python-sqlite	pysqlite	Substitute “py” by “python-”		
30	2018-10-30	python-ftp	pyftpdlib	Delete “dlib” & Substitute “py” by “python-”		
31	2018-10-30	python-mysqldb	MySQL-python	Swap “python” and “mysql” & Insert “db”		
32	2018-10-30	smb	pysmb	Delete “py”		✓
33	2018-10-31	pythonkafka	kafka-python	Swap “kafka” and “python” & Delete ‘-’		
34	2019-12-01	jeIlyfish	jellyfish	Substitute ‘l’ by ‘I’	✓	
35	2019-12-01	python3-dateutil	python-dateutil	Insert ‘3’	✓	
36	2018-04-25	ssh-decorate	ssh-decorate	Hijacked Package		

sensitive information to a remote server, nine of the packages were created by deleting characters from original packages.

Several combosquatting packages were distributed by exploiting the usage of common prefixes or suffixes within PyPI packages (see the analysis in Section V), e.g., `pytz` into `pytz3-dev` making the malicious package look like the distribution of a Python 3 development version of the original package. Similarly, attackers used a singular form of a package name instead of a plural one (`setuptool` instead of `setuptools`), add special characters (e.g., hyphens or underscores), somewhere in a package name (`setup-tools` instead of `setuptools`, `easy_install` instead of `easyinstall`), or create a similar package name a commonly known tool or a module of the standard library (e.g., `pwd`[13]). Attackers leveraged the difference in spellings of UK and US languages: malicious package `colourama`, resembling the benign package `colorama`, download a crypto miner upon installation by victims.[14]

IV. Telling Malicious Packages Apart

We start by describing our collection of the ground-truth packages. They are the ones whose characteristics of a legitimate package. Then we proceed to identify suspicious packages whose names that are the same or similar to the ground-truth packages.

Assumptions: Our ground-truth packages consist of two trusted sources:

1) Modules of the Python standard library[15] (e.g., `os`, `csv`, `re`) that are bundled into Python distributions.
2) PyPI packages with known source code repositories.

Similar to [3], we assume that a PyPI package should not use the name of a module of the Python standard library. If a package in PyPI has an exact or nearly identical name to one of these modules, we mark such a package as suspicious.

We assume that a PyPI package's legitimacy could be verified by checking its source code repository that provides additional metrics (e.g., number of stars, followers, and forks) often used by developers to reason about the package reputation and community support [7]. Considering the examples of Table I, we observe that developers tend to use the same package name as the repository name in Github. Although there is no strict requirement on the name correspondence, we base on this observation to identify the list of packages with known code repositories as ground-truth:

- PyPI packages whose names are the same as the repository names in Github are not typosquatting packages,

[13]There exist similar Linux commands.

[14]https://medium.com/@bertusk/cryptocurrency-clipboard-hijacker-discovered-in-pypi-repository-b66b8a534a8

[15]https://docs.python.org/3/library/

- PyPI packages whose names are different than the repository names or do not have any reference to a Github repository, require additional verification (Section VI).

Algorithm: To measure the similarity of package names, we calculate the Levenshtein distance [4] between each pair of packages and check if the distance is less than or equal to a threshold heuristically. Based on the list of previous typosquatting package names in Table I, we define the threshold for the package name similarity to be equal to two as it allows us to identify the majority of known attacks (21 out of 36) and reduce the number of false positives.

Figure 2 summarizes the proposed idea for detection of squatting packages. Packages where source appears in a repository (e.g., Github) can be verified either by checking their reputation (e.g., number of stars) or source code. We assume that the packages that do not have source code repositories or share the same repository while having a different name at the same time with another are suspicious. We note that it is not required that a legitimate package name in PyPI is the same as the repository name in Github or other version control systems.

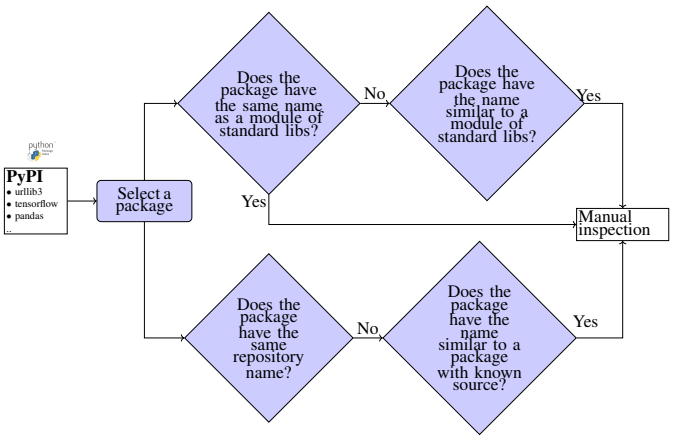

Fig. 2: Detecting Suspicious Squatting Packages.[16]

Step 1: Processing of modules of Python standard library To compose a list of module names of the Python standard library, we base on the `stdlib-list`[17] package to collect Python standard library module names of the Python standard library from nine different Python versions. Then, we scan the whole PyPI and report packages whose names are the same as any module of Python standard library as suspicious.

Step 2: Processing of PyPI packages with known source code repositories For those PyPI packages whose URLs lead to Github repositories, we use the URLs to extract the source cdoe repository names. After comparing the repository and package name, we classify packages that have the same name as not created for a typosquatting attack and all other packages as required to pass through additional verification.

Step 3: Identification of packages possibly created for squatting attacks We look for packages whose names have the Levenshtein distance less or equal than two to the module names of Python standard library and the packages whose the same names as code repositories (Step 2). To capture the common prefix "python" added during the combosquatting attack, we preprocess package names by substituting "python" with '*'. This transformation allows us to capture, e.g., attacks #20, #25-27, #29-31, and #33 in Table I.

[16]To simplify the algorithm, unspecified paths all lead to the legitimate packages which are not required inspection.

[17]https://pypi.org/project/stdlib-list/

TABLE II: Descriptive statistics of package name lengths.

	count	mean	std	min	25%	50%	75%	max
package	216 547	12.4	7.4	1	7	10	16	80

V. Empirical results

Descriptive statistics: In total, we analyzed 216 548 packages from PyPI as of February 20, 2020. On average, a package name has 12 characters, while lengths of names of 50% of packages are shorter or equal to ten characters. Table II shows descriptive statistics of package name lengths in PyPI.

We identified 165 878 packages have homepage URLs, and 197 packages provide code page URLs.[18] The largest share of PyPI packages use Github as a place to store their source code: 141 358 homepage URLs (85%) and 196 codepage URLs (99%). Other packages host their source code on GitLab (2792 homepage URLs and 10 codepage URLs), BitBucket (4606 homepage URLs and three code page URLs), Google code (847 homepage URLs), and SourceForge (618 homepage URLs). 3550 packages (13.3%) either do not provide codepage URLs or use URLs on PyPI as their homepages.

We noticed that 613 packages use https://github.com/pypa/sampleproject as their homepage URLs. This case might happen because package developers had used a template to create PyPI packages, but did not update their homepage URLs to the template. Moreover, 12 266 other packages are sharing several homepage URLs. This might happen for both malicious and benign reasons. The typosquatting package `jeLlyfish` used the homepage URL of the legitimate package `jellyfish` unchanged so it may look legitimate to End Users.

Figure 3 shows the distribution of the Levenshtein distances between all packages names in PyPI. The distance distribution has a shape of a normal distribution with 4 225 244 (0.02%) pairs of packages that have distances between one and two. 54.8% of the pairs have a distance between 9 and 16. The biggest Levenstein distance between package names is 80.[19]

Hence, the identified threshold of the Levenshtein distance (d=2) could be seen as a good trade-off, since it allows us to identify the majority of known attacks and does not generate a significant number of alerts (d=4 increases the 51 times amount of alerts from 4.2 million to 215 million).

Known attacks. We observe that the attacks in Table I targeted popular packages: seven Github repositories (25%) of the attacked packages have more than 2356 stars, 14 repositories (50%) have more than 972 stars, and 21 repositories (75%) of the packages have more than 84 stars. Fifteen out of the 28 squatting attacks are identified by setting the distance

[18]We found seven packages whose URLs are broken and corrected them.

[19]`2to3` and `aaaaaaaaaaaaaaaaaaa-aaaaaaaaa-aaaaaaaasa-aaaaaaasa-aaaaaaasa-aaaaaaasa-bbbbbbbbbbb`

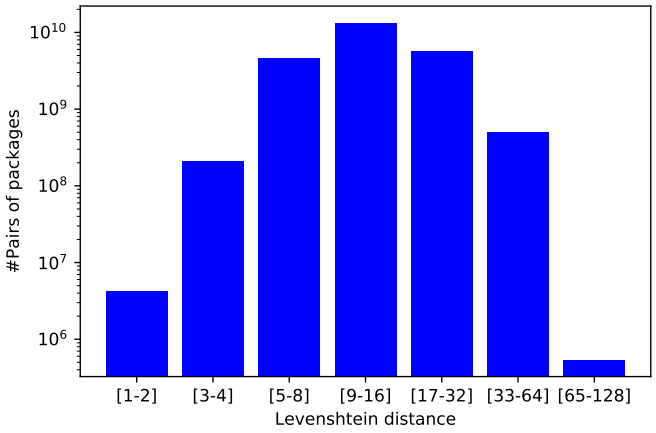

Fig. 3: Levenshtein distance distribution of package names.

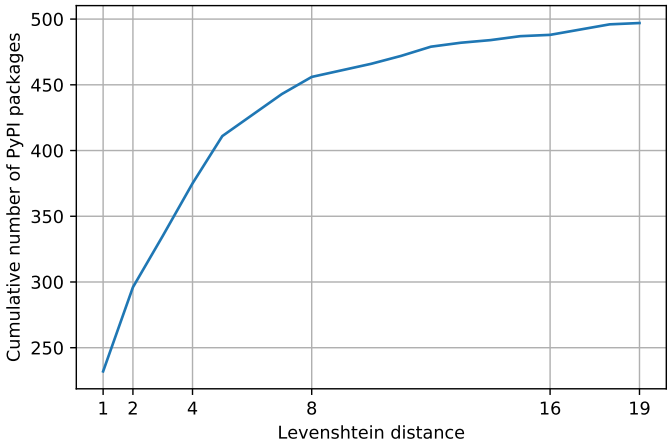

Fig. 4: #Packages whose name differ from standard modules.

threshold of one. Increasing the threshold to two allows us to capture six additional attacks (21 out of 28).

Looking at other packages in PyPI.

Modules of Python standard library: We found 62 packages in PyPI that have the same names as modules of the Python standard library. We further checked whether we found the packages published by Stagg [3]: while Stagg published 'empty' packages that were removed from PyPI, we identified several packages with non-empty sources. Particularly, 16 out of 62 packages do not have any releases, and 12 packages have only one release. Hence, we confirm that these packages are not the ones published by [3] and confirm our manually analysis.

Figure 4 shows the distance distribution between Python standard library module names and other packages' names. There are 296 PyPI packages whose names have the distance less than or equal to two from Python standard library module names, and therefore, suspicious.

PyPI packages with known sources: From those packages that use the Github to host their source code, 79 933 packages (36.9%) have the same name for both Github repository and package name in PyPI (i.e., safe packages). Names of 61 522 packages differ from the names of their source code

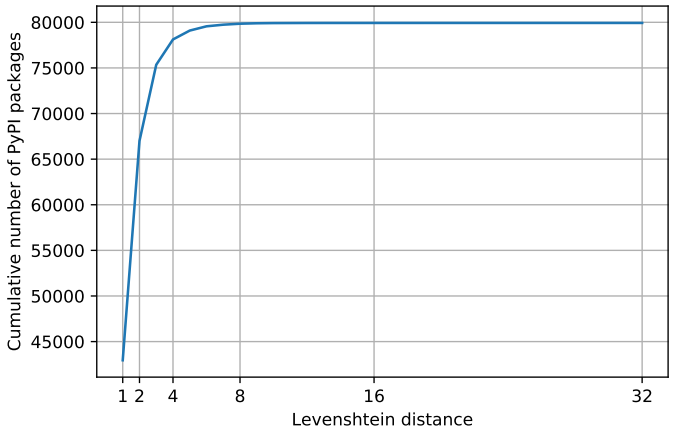

Fig. 5: #Packages whose names differ from repository names.

repositories, and therefore, these packages require additional analysis. In Figure 5, we identified approximately 65 000 PyPI packages have the names similar to the packages with known sources (they have a distance less than or equal two).

Interesting naming patterns. We noticed some patterns between the package names and repository names:

- 16 070 (7.4%) package names include names of their code repositories or vice versa (e.g., the package `0-core-client` has its repository name `0-core`),
- several common prefixes and suffixes are added or removed from the repository names to create PyPI package names. Several common prefixes are 'python-' (2287), '-python' (1343), '.git' (1324),

Understanding these naming patterns might potentially support predicting future combosquatting attacks of legitimate packages as adding/deleting a suffix or prefix is one of the combosquatting strategies (See Section III).

VI. Threats to Validity

Missing potential typosquatting packages. Our approach can be applied for detecting typosquatting candidates of packages whose names are longer than 2 (the selected threshold). However, as shown in Table II, 75% of packages in PyPI have more than seven characters in their names. Hence, our proposed approach relies on the Levenshtein distance applicable to most of the PyPI ecosystem packages. To capture packages with short names, we plan to use common name patterns (e.g., repeated or swapped characters)([5],[8]).

False positives The proposed approach has generated false-positive findings. For example, a package name might differ from its source code repository name for a good reason, e.g., the developers could not register the repository name because the name has been reserved. We manually verified 62 packages (the ones look the same to standard libraries) and identified the following reasons behind existing of these packages in PyPI: backport to old Python versions (17), empty packages (17), toy packages (2), legitimate deprecated packages with different functionality (26).

The manual analysis allowed us to identify the following ideas to reduce the number of false positives by analyzing:

package info (e.g., author reputation, package popularity) and code features (e.g., suspicious API calls).

We use source code repository as a trusted source. The legitimacy of a package depends on the equality of package and repository name. However, a repository name is not necessarily unique across Github (e.g., stub42/pytz and newvem/pytz). An organization and a repository name identify a project in Github. Hence, attackers may create a repository with the same name but different organization identifiers as an existing repository in Github (or even a new repository) and publish a corresponding package in PyPI. Our approach is not capable of identifying such an attack. However, this attack requires additional effort to trick the users into downloading packages that come from an unknown source. To overcome this limitation, we plan to extend the proposed approach to considering other reputation metrics (e.g., number of stars).

On the other hand, one Github repository might be used to store code of several different PyPI packages. For example, a Github repository https://github.com/Azure/azure-sdk-for-python organization corresponds to 140 packages in PyPI (e.g., `azure`, `azure-ai-nspkg`). In such a case, our approach would generate a false alert.

Also, the referred repository does not need to contain code that bears any relation to the package. We plan to employ a code analysis to identify such a discrepancy (whether a particular code fragment in a package originates from its source code repository). This might be a good signal for detection of injected code added by attackers or backporting changes added by developers directly to the packages.

Some known attacks are not caught. Although the proposed approach allows us to catch most of the known typosquatting attacks, some attacks in Table I still remained unidentified. Additional ways of checking modifications of package names might allow the detection of such attacks. For example, attacks #20, #22, #23, #24, #25, #27 are based on the permutation of the legitimate package names, while typosquatting package names in attacks #4, #5, #12, #8 were created as a result of deletions of a part of legitimate package names. However, including such checks to enlarge the search space for the possible typosquatting candidates might significantly increase the number of false positive alerts, and therefore, could not be used alone. Hence, we are planning to investigate the common patterns of packaging names in future work and embed them into our approach.

VII. Related work

Duan et al. [9] extract various features of a package to identify its maliciousness. They rely on the predefined list of popular packages to find suspicious packages. We propose a method to automatically find legitimate packages that can be used for finding potential typosquatting attacks.

Tschacher [1] presented a comprehensive analysis of typosquatting attacks, including the systematic generation of typosquatting package names, the publication of forks of the original packages in several open-source ecosystems. By doing this, they can measure the severity of such an attack by counting the number of successful installations. Our study complements this work by providing an approach for obtaining a list of legitimate package names that can be used for automatic identification of typosquatting packages in PyPI.

Taylor et al. [8] proposed an approach to identify typosquatting candidates based on package name patterns, like repeated, omitted, or swapped characters, common typos, swapped words, and Python version numbers. While the authors considered the most downloaded packages, we have analyzed all the packages with source code repository links.

VIII. Conclusion and Future Work

In this paper, we have studied attackers strategies in crafting malicious packages and trick Python developers into downloading malicious packages whose names are similar to the legitimate packages, i.e., the combosquatting and typosquatting attacks in the Python Package Index ecosystem. We have also proposed an automatic approach for detecting packages affected by the typosquatting attack. The empirical evaluation of the proposed approach on the list of known squatting attacks suggests that the approach is promising to be used in future research for automatic identification of malicious packages in PyPI and has the potential for creating an automatic system that prevents squatting attacks in the PyPI ecosystem.

For the future work, we plan to extend the approach to employ the code level checks of the identified packages so that it will be capable of automatically identifying injected malicious code snippets into Python packages. Additionally, we plan to explore the applicability of the approach to other software ecosystems, like NPM or Maven.

Acknowledgments

This research has been partly funded by the EU under the H2020 Programs H2020-EU.2.1.1-CyberSec4Europe (Grant No. 830929), the NeCS: European Network for Cyber Security (Grant No. 675320) and the SPARTA project (Grant No. 830892).

References

[1] N. P. Tschacher, "Typosquatting in programming language package managers," Bachelor's Thesis, Universität Hamburg, Fachbereich Informatik.
[2] A. Z. Robert Perica, "Suppy chain malware - detecting malware in package manager repositories," https://blog.reversinglabs.com/blog/suppy-chain-malware-detecting-malware-in-package-manager-repositories.
[3] S. Stagg, "Building a botnet on pypi," https://hackernoon.com/building-a-botnet-on-pypi-be1ad280b8d6, 2017.
[4] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, 1966.
[5] Y. Hu, H. Wang, R. He, L. Li, G. Tyson, I. Castro, Y. Guo, L. Wu, and G. Xu, "Mobile app squatting," in *Proc. of WWW'2020*, 2020.
[6] P. Kintis, N. Miramirkhani, C. Lever, Y. Chen, R. Romero-Gómez, N. Pitropakis, N. Nikiforakis, and M. Antonakakis, "Hiding in plain sight: A longitudinal study of combosquatting abuse," in *Proc. of CCS'17*.
[7] I. Pashchenko, D. L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proc. of CCS'20*.
[8] M. Taylor, R. K. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, "Spellbound: Defending against package typosquatting," *arXiv preprint*.
[9] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Measuring and preventing supply chain attacks on package managers," *arXiv preprint*.