Detecting Python Malware in the Software Supply Chain with Program Analysis

Ridwan Shariffdeen
National University of Singapore, Singapore
ridwan@comp.nus.edu.sg

Behnaz Hassanshahi
Oracle Labs, Australia
behnaz.hassanshahi@oracle.com

Martin Mirchev
National University of Singapore, Singapore
mmirchev@comp.nus.edu.sg

Ali El Husseini
National University of Singapore, Singapore
CNRS@CREATE, Singapore
elhusali@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore, Singapore
abhik@comp.nus.edu.sg

***Abstract*—The frequency of supply-chain attacks has reached unprecedented levels, amounting to a growing concern about the security of open-source software. Existing state-of-the-art techniques often generate a high number of false positives and false negatives. For an effective detection tool, it is crucial to strike a balance between these results. In this paper, we address the problem of software supply chain protection through program analysis. We present HERCULE, an inter-package analysis tool to detect malicious packages in the Python ecosystem. We enhance state-of-the-art approaches with the primary goal of reducing false positives. Key technical contributions include improving the accuracy of pattern-based malware detection and employing program dependency analysis to identify malicious packages in the development environment.**

Extensive evaluation against multiple benchmarks including Backstabber's Knife Collection and MalOSS demonstrates that HERCULE outperforms existing state-of-the-art techniques with 0.949 f1-score. Additionally, HERCULE detected new malicious packages which the PyPI security team removed, showing its practical value.

***Index Terms*—software supply chain security, malware detection, program analysis**

I. INTRODUCTION

Recent reports indicate a remarkable surge in the number of OSS supply chain attacks over the past three years, reflecting an astonishing growth rate [1]. Popular package repositories such as PyPI have been flooded with malicious packages, where the PyPI team has removed 12,000 packages in 2022 alone [2]. This alarming increase in malicious packages is further fueled by the recent advancements in large language models (LLMs), which are extensively utilized in the development of artificial intelligence (AI) software systems.

Attackers have been successful in distributing malware by tricking developers into installing malicious packages with the use of Typosquatting [3], Combosquatting [3], Dependency Confusion [4] and Repo Confusion [5]. Typosquatting is a technique that leverages the mistakes a user makes when installing a well-known package with a typo in the package name. Similarly, combosquatting is a technique that rearranges package names in a different order.

A Dependency Confusion attack exploits a vulnerability in the current Python dependency resolution workflow. Malicious users can hijack the namespace of a private dependency by creating a duplicate entry in the public PyPI registry. Since Python's dependency resolution prioritizes public registries over private ones, this allows adversaries to override private dependencies. For instance, PyTorch [6] fell victim to a dependency confusion attack [7], where the malicious code masquerading as the official package received over 2,000 downloads, compromising sensitive information.

Previous efforts to detect such malicious packages have focused on analyzing individual packages, utilizing techniques such as static analysis [8], dynamic analysis [9], and machine learning [10]. Existing static analysis methods [11], [9] often suffer from a high rate of false positives and can be easily circumvented using obfuscation techniques. While dynamic analysis methods provide greater precision and resilience against simple obfuscation, they tend to produce a high number of false negatives. These techniques are limited in their ability to trigger complex malicious behaviors due to the constraints of their testing strategies, resulting in a high number of false negatives. Dynamic analysis necessitates sandbox environments, thus difficult to use in development environments. Furthermore, existing dynamic and static methods are hindered by advanced obfuscation techniques such as splitting malicious code across dependencies [12]. An example attack would be the divide-and-hide attack [13], which employs modular components to formulate the attack by separating malicious code within multiple packages. Existing techniques [14], [11], [15], analyze packages as a single entity rather than analyzing the complete program which uses modular components from other packages. In addition, such methods lead to a large number of false positives [9] since they do not provide sufficient information apart from matching previously known attack pattern signatures [16]. For example, the previous PyPI vetting pipeline only inspected `setup.py` for malicious code that would execute during package installation [8]. Although `setup.py` is a common target, malicious code can also be injected in other files and separated in multiple packages.

In this paper, we propose an *inter-package analysis* approach to detect malicious Python packages for supply chain protection. Our method comprises three key analysis steps. First, we assess the integrity of a distributed package in relation to its source code repository. We perform a differential analysis of the changes between the distributed artifact and the source code to identify malicious updates that were not included in the repository. This approach is based on the observation that malicious actors often conceal harmful code within the final artifact, without disclosing it in publicly accessible source code repositories. Second, we perform data-flow analysis to identify suspicious data flow between different information sources and sinks that can reside in different packages. The core idea is to incorporate data-flow analysis to detect sensitive information flow to malicious targets across different packages. We call such data flow that cross package boundaries **inter-package** flows. Finally, we perform a transitive dependency analysis to detect packages that import previously detected malicious packages as dependencies. Any package (even a legitimate one) can be compromised to distribute and install other malicious packages [7].

In summary, the contributions are as follows.

- We present an *inter-package analysis* based approach for supply chain protection. We implement our analysis in a tool named HERCULE, which outperforms state-of-the-art tools on known malicious datasets and can be extended to detect new types of malicious behaviors.
- We show that by using differential analysis on the abstract syntax tree representation of the source code, we can significantly reduce the number of false positives of existing pattern-based malware detection tools.
- We encode malicious behavior in the CodeQL query language and show static analysis can be extended beyond pattern-based matching for malware detection in Python ecosystem. We also propose a transitive dependency analysis to detect packages that are using other malicious packages.
- Our experimental results show that static analysis can effectively detect malware spread through software supply chain attacks, achieving high recall while maintaining a low false positive rate for benign packages. Furthermore, our approach identifies previously unreported malware.

II. MOTIVATIONAL EXAMPLE

In this section, we demonstrate how a developer environment can be compromised through a supply chain attack, using a real-world malicious package identified and reported by HERCULE, which has since been removed from PyPI. Consider a developer attempting to create a custom dataset of Wikipedia entries. To accomplish this, the developer needs a data scraper, which is a program designed to automatically retrieve web pages and convert HTML data into JSON format for processing by other tools in the scraping workflow. A common practice among developers is to reuse existing programs that offer these capabilities, rather than building them from scratch. The PyPI registry includes a package called `wikipedia-scraper` that provides the desired functionality. However, this package is actually a facade created by an attacker to facilitate a supply chain attack.

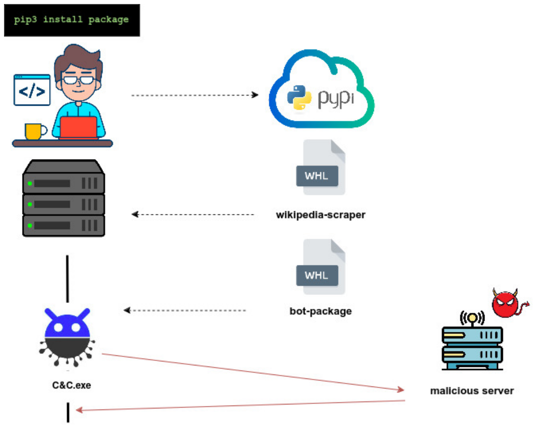

Fig. 1: An illustration from a real-world example how a developer environment can be compromised via supply chain attack

Figure 1 illustrates the supply chain attack carried out by installing package `wikipedia-scraper`. The package itself does not contain any malicious code, however it enlists a dependent package `bot-package` which will get installed as part of the dependency resolution. The attack is not carried out at the installation stage, rather when the program runs. Invoking the API in package `wikipedia-scraper` triggers a chain of API calls that eventually executes a malicious bot. Once deployed, the bot executes in the background and listens on a port, waiting for commands from a command and control server via GET and POST requests.

Inspecting the code of the package `wikipedia-scraper` in isolation is not enough to identify the malicious behavior since part of the malicious code resides in a dependent package i.e. `bot-package` using the *divide-and-hide* [13], [12] obfuscation technique. This highlights one of the limitations in existing techniques, which analyze packages individually one at a time. In order to detect such attacks the detection technique needs to analyze all dependent packages as well.

To effectively protect the developer environment against such attacks, it is essential to conduct an analysis that identifies all dependencies of a downloaded package, including both explicitly declared and implicitly used ones. Once these dependencies are established, it is critical to perform an *inter-package* analysis across all transitively dependent packages to detect and flag any malicious ones. This analysis should capture data and control flow between different packages to identify suspicious behaviors, such as executing unauthorized commands. Additionally, it is vital for the analysis to minimize false positives to be effective in practice.

III. APPROACH

We propose HERCULE, which employs a multi-staged *inter-package* static analysis approach. Our method comprises three

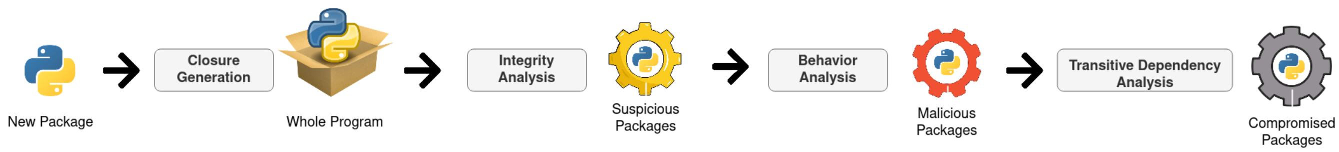

Fig. 2: High-level overview of our approach HERCULE

analyses, each offering insights to help users understand why a package is flagged as malware. First, we analyse the integrity of the package to verify if the distributed artifact adheres to the corresponding source repository modulo benign changes due to the build process. Second, we perform a whole-package behavior analysis to detect suspicious data-flow or control-flow involving protected data sources. Finally, through a transitive dependency analysis, we detect packages that install known malicious packages.

A. High-level Overview

Figure 2 depicts the high-level overview of HERCULE. Given a package named *Package-Z*, our approach first generates the closure of the package $\mathcal{S}$. The closure of the package is the complete collection of dependent packages that are both explicitly and implicitly defined as dependent packages. Note that this collection can be different from the set of necessary packages required to execute *Package-Z*, which is a subset of the closure of the package $\mathcal{S}$. A package may be installed as a dependency of *Package-Z* without ever being used within the package itself. However, because this package is explicitly declared in the metadata files, such as `pyproject.toml`, it will still be installed on the system. Therefore, we include all such packages as part of the closure. Similarly, packages that are not explicitly declared but implicitly used in *Package-Z* are also included in the closure $\mathcal{S}$. Malicious code injected by adversaries may import packages without explicitly declaring them as dependencies to avoid detection. Therefore, we recursively enumerate all implicitly and explicitly declared dependent packages to generate the closure $\mathcal{S}$. We rely on existing features in the package manager *pip* [1] to resolve dependency versions when an explicit version is not specified.

Second, we analyse the integrity of the downloaded package with respect to the corresponding source repository. This analysis provides a signal if the downloaded artifact was modified either during the build process or at the publishing stage. Detecting artifacts that do not adhere to the source repository helps to identify suspicious files that are not aligned with the source repository. Note that misaligned artifacts are flagged as suspicious but not malicious since new files could be introduced or existing files can be modified as part of the build process [8].

Third, HERCULE iterates over the dependency graph of the package to identify known malicious packages. If a malicious package is downloaded as a dependency of the package, we flag the scanning package as malicious. In our motivational example, `wikipedia-scraper` is flagged malicious since it downloads the known malicious package `bot-package` as part of its dependency resolution.

Fourth, HERCULE performs **inter-package** static analysis to track data and control flow across packages. The inter-package analysis includes all definitions and declarations used in the package and its dependencies. HERCULE scans the entire codebase, including all dependencies, for any code that matches abstractions captured from previously known malware. The semantics of malicious behavior is encoded as CODEQL[2] queries, which is a code analysis engine developed by GitHub. We develop queries in CODEQL to specifically analyze suspicious source-sink data flows.

Finally, HERCULE classifies the package as malicious if there is at least one detected malicious behavior or there is one malicious package found in the transitive dependencies.

B. Integrity Analysis

Algorithm 1 lists the steps for our integrity analysis. The first step is to accurately determine the source code repository R for the distributed package P. Following best practices to construct a Software Bill of Materials (SBOM), package distributors are encouraged to advertise the corresponding source-code repository which can be used by third-party analysis such as HERCULE to verify the integrity of the build artifact. FETCHSOURCEREPO method in Algorithm 1 enumerates the meta-data for the package to identify the source repository and the release version. If a repository is detected, HERCULE fetches the source code matching the release version for comparative analysis. Most of the source repositories are hosted with version control systems and follow best practices in software engineering to *tag* each release. HERCULE uses the *Levenshtein distance*[3] to identify the corresponding commit tag in chronological order of the tag creation time. If a corresponding tag cannot be found, the latest version of the code is fetched.

For each python file in the distributed package P, we find the corresponding source file from the code repository R using code clone detection i.e. FINDCLONE(). Each source file f is first normalized by applying standard refactoring techniques to prevent mismatches due to formatting changes that may occur during the build process. In addition, all source files are upgraded to the latest version of Python using the `pyupgrade`[4] tool, which allows HERCULE to only support the latest version of Python. For each identified pair of source

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

[2] https://codeql.github.com/

[3] https://en.wikipedia.org/wiki/Levenshtein_distance

[4] https://pypi.org/project/pyupgrade

Algorithm 1: Integrity Analysis

```
Input: Python Package $P$
Output: suspicious set of modified locations $\mathcal{L}_{susp}$,
        suspicious set of new files $\mathcal{F}_{susp}$
$\mathcal{L}_{susp} \leftarrow \emptyset$, $\mathcal{F}_{susp} \leftarrow \emptyset$
$\mathcal{F}_{pkg} \leftarrow$ ExtractSourceFiles($P$)
$R \leftarrow$ FetchSourceRepo($P$)
if $R \neq \emptyset$ then
    $\mathcal{F}_{src} \leftarrow$ ExtractSourceFiles($R$)
    for file $f \in \mathcal{F}_{src}$ do
        $f_N \leftarrow$ Normalize($f$)
    end
    for file $f \in \mathcal{F}_{pkg}$ do
        $f_N \leftarrow$ Normalize($f$)
        $AST_{pkg} \leftarrow$ GenAST($f_N$)
        $f_{clone} \leftarrow$ FindClone($f_N$, $\mathcal{F}_{src}$)
        if $f_{clone} \neq \emptyset$ then
            $AST_{src} \leftarrow$ GenAST($f_{clone}$)
            $\mathcal{L}_{changes} \leftarrow$ ASTDiff($AST_{src}$, $AST_{pkg}$)
            if $\mathcal{L}_{changes} \neq \emptyset$ then
                for $loc \in \mathcal{L}_{changes}$ do
                    $\mathcal{L}_{susp} \leftarrow \mathcal{L}_{susp} \cup loc$
                end
            end
        end
        else
            $\mathcal{F}_{susp} \leftarrow \mathcal{F}_{susp} \cup f_N$
        end
    end
end
else
    $\mathcal{F}_{susp} \leftarrow \mathcal{F}_{pkg}$
end
return $\mathcal{L}_{susp}$, $\mathcal{F}_{susp}$
```

files from the distributed package P and the repository R, we analyze the changes at the Abstract Syntax Tree (AST) level. Operating at the AST level allows to disregard subtle textual changes (i.e. carriage return token) and reduces the discrepancies between the distributed package P and the code repository R. For example, if source code is developed in a Unix environment but the build happens in a Windows environment, textual changes due to differences in encoding might be introduced.

C. Behavior Analysis

Code differences in the distributed package compared to the source repository alone, do not provide sufficient evidence to flag the package as malicious. The analysis must provide additional substantial evidence of malicious behavior to flag a package as malicious. Thus, a behavioral analysis is needed to detect packages that exhibit malicious behaviors.

We employ *inter-package* analysis, more specifically dataflow analysis to detect potentially malicious behavior. For this purpose, we use CodeQL, which is a semantic code analysis engine primarily used to detect software vulnerabilities in a range of programming languages (i.e. C, C++, Go, Java, Python etc). CodeQL generates a relational database capturing program semantics using the abstract syntax trees, control flow graphs, and data flow graphs. Users can analyze the program by running queries against the generated database with its unique object-oriented query language. For our purposes, we reuse the same infrastructure but develop the queries to capture patterns specific to previously studied supply chain attacks.

To improve recall and avoid missing malicious behavior, we first construct the closure of the package by recursively retrieving all necessary dependencies. Querying this closure enables us to capture code that may be fragmented or concealed within the dependencies [12].

CodeQL Queries Hercule comprises CodeQL queries designed to identify data flows from sensitive sources to sinks, specifically targeting flows previously associated with supply chain attacks [17], [9], [18]. We have written a total of 22 queries, which can be categorized into the following groups.

- **Encoding/Obfuscations**: detect data flows where the data source is encoded or obfuscated. Common behavior observed in malicious packages to thwart rule-based detection techniques.
- **File Operations**: detect data flows that access and manipulate the file system. This includes identifying flows from network sources to the file system.
- **Network Connections**: detect using IP addresses or domain names to create network connections. Hard coded IP addresses and domain names are commonly found in malicious packages.
- **Processes**: detect remote connections influencing operating system commands or new processes, to identify potential remote code executions.
- **Exfiltration**: detect information flow from the local environment or file system to remote end points.

First, we detect data flows that use encoding such as base64 to detect obfuscation in the code. However, the detection is not merely the existence of a base64 encoding but a behavior that exhibits malicious intent (i.e. compile code). We also analyze file modifications to protect files, which can be used to corrupt the integrity of the environment. The protected class of files includes files that encompass the running environment (i.e. Python library files) and configuration files used by the system (i.e. `/etc/resolv.conf`). Behavior that exhibits remote network connection establishment will also be detected, typically used to load the payload of the malware. We also curate queries to capture data flows from sensitive sources to untrusted sinks. Sensitive sources refer to information specific to the running system, while untrusted sinks are those with remote connections that transmit data outside the system.

D. Transitive Dependency Analysis

Using behavior analysis we can detect packages that download other malicious packages. Even though these packages themselves do not directly contribute to the attack, indirectly they install the malicious packages as their dependencies. As discussed earlier, adversaries use various tactics such as *divide-and-hide* and *dependency-confusion* [7] to install malicious dependencies. A user who installs a package installs the entire dependency chain. Thus, it is imperative to analyze the entire dependency graph to protect against supply-chain attacks.

Hercule generates the dependency graph of the *whole-program* based on the dependencies resolved by the package manager *pip* and finds the relation between each of the packages with respect to malicious packages. The dependency graph consists of nodes representing each package in the *whole-program*, with directed edges representing the dependency relation.

Hercule transitively scans each dependency chain in the package closure to detect the existence of a known malicious package. If any of the transitive dependencies include a link to a known malicious package, the package will be flagged as a malicious package. The chain of dependencies that installs a known malicious package will be reported to the user.

IV. Implementation and Evaluation

We implemented Hercule in Python. We use the Python implementation of GumTree to identify AST differences. For behavior analysis we develop CodeQL queries with malicious behavior specifications accustomed for supply chain attacks. We use the default Python package manager *pip* to resolve dependencies and construct the closure of the program.

We evaluate Hercule's effectiveness in identifying malicious packages by answering the following research questions:

RQ1 How effective is Hercule in detecting malicious packages compared to existing state-of-the-art techniques?

RQ2 What is the contribution of each analysis step in Hercule in detecting malicious packages?

RQ3 How effective is Hercule in reducing false positives for benign packages?

RQ4 How efficient is Hercule across benign and malicious packages?

A. Experimentation Setup

Dataset: We evaluate our technique using two types of dataset; existing benchmarks in the literature, and benign packages collected from PYPI registry. From the existing benchmarks in the literature, we extracted 255 unique malicious packages from MalOSS data-set [9], out of which we removed 13 files that are not Python packages (including DLL files and `setup.py` files), resulting in 242 malicious packages. Similarly we extracted 345 malicious packages from Backstabber's Knife Collection [17] and 2935 packages from PyPIMalRegistry data-set [18]. We further supplement our data-set with 1300 benign packages. We curate our benign packages from a set of packages published by Google, Microsoft, Oracle, and Amazon, which are called trusted packages as well-known organizations publish them. The benign packages also include the top-100 popular packages[5] on the PyPI registry. These packages are the most downloaded, most used, and most depended upon in the PyPI ecosystem. Although there is no guarantee that these packages are not compromised or do not contain malicious code, we use this dataset to provide insights on the false-positive rate of Hercule.

TABLE I: Statistics of the packages used in the evaluation

Category	Data-Set	# Pkgs	Size	Files	LoC
Malicious Packages	MalOSS	242	214.87	5.21	491.55
	BackStabber	345	407.09	9.51	1374.18
	MalRegistry	2935	40.67	10.79	487.36
Benign Packages	Popular Packages	100	2147.49	395.80	25806.06
	Trusted Packages	1200	1781.06	116.05	18778.47

Table I lists the details of the python packages used in our evaluation. Columns "#Pkgs" captures the number of packages in each dataset. Columns "Size", "Files" and "LoC" capture the average file size of the distributed package, the average number of files in a package and the average number of lines of code (Python), respectively. There is a significant difference in a malicious package captured in existing benchmarks and the benign packages we captured from PyPI. All three metrics including the package size, number of files and number of lines of code are in different order of magnitude. This also underscores a key challenge for detection tools: scalability.

Comparison with Existing Tools: For comparison with existing state-of-the-art tools we select tools that support Python packages for supply chain malware detection. Bandit4Mal [11] is an extension of Bandit [14]. Bandit is designed to find common security issues in Python code. Bandit4Mal extends Bandit by adding rules specifically targeting patterns derived from previously studied supply chain attacks. MalOSS [9] is another technique that employs a combination of static and dynamic analysis to detect malicious packages in multiple ecosystems, supporting *PyPI*, *npm*, and *RubyGems*. We note that the current version is outdated due to lack of maintenance, limiting us to using only the static analysis component in our experiments. GuardDog is a rule-based detection tool employing a set of predefined heuristic rules and patterns to characterize suspicious behavior.

LastPyMile [8] is a technique specifically designed to detect phantom files, i.e., source files within a package that do not match any source files in the advertised origin repository. LastPyMile uses artifact hashing to identify discrepancies between files in the source repository. We compare LastPyMile with the integration analysis component of Hercule. For comparison purposes, we implemented our own version of LastPyMile since the original repository[6] is customized for packages currently hosted on PyPI, however the malicious packages in existing benchmarks are no longer available on PyPI.

Setup: All experiments are conducted using Docker containers on a 192-core 2.40GHz 512G RAM Intel Xeon machine.

B. Evaluation: Efficacy of Hercule

We evaluate if Hercule can effectively detect malicious packages and can accurately classify benign packages. We also compare the efficacy of Hercule with other state-of-the-art

[5] https://hugovk.github.io/top-pypi-packages/

[6] https://github.com/assuremoss/lastpymile

TABLE II: Comparison with state-of-the-art supply chain malware detection tools

Data-Set	**# Pkgs**	**Hercule**				**GuardDog**				**Bandit4Mal**				**MalOSS**			
		TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN
MalOSS	242	213	0	0	29	142	0	0	100	220	0	0	22	185	0	0	57
BackStabber	345	313	0	0	32	294	0	0	51	330	0	0	15	247	0	0	98
MalRegistry	2935	2761	0	0	174	1985	0	0	950	1971	0	0	964	2505	0	0	430
Popular Packages	100	0	3	97	0	0	18	82	0	0	93	7	0	0	38	62	0
Trusted Packages	1200	0	113	1087	0	0	37	1163	0	0	981	219	0	0	411	789	0
Overall	4822	3287	116	1184	235	2421	55	1245	1101	2521	1074	226	1001	2937	449	851	585

tools in these data-sets. Table II compares the performance of state-of-the-art tools on the studied Python packages. Each column summarizes the efficacy of each tool. Sub columns "TP" and "FP" depict the count for true positives and false positives, respectively. A true positive is a malicious package that is correctly identified as such, while a false positive occurs when a benign package is incorrectly classified as malicious. Columns "TN" and "FN" represent the count for true negatives and false negatives, respectively. A true negative refers to a report that labels a benign package as benign, and a false negative refers to a malicious package that is identified as benign.

Bandit4Mal identifies the most number of malicious packages while Hercule has the second highest count, and MalOSS scored the lowest. In terms of false positives, Bandit4Mal reports the highest number of false positives and GuardDog reports the lowest. Bandit4Mal uses pattern based matching without differentiating intended behavior and malicious behavior, which flags many packages as malicious resulting in the highest number of false positives. GuardDog is more conservative and reports the least number of false positives, but also has the highest number of false negatives. Using a more lightweight integrity analysis to detect intended behavior Hercule strikes a balance between finding the most number of malicious packages while reporting the least number of false positives. Hercule detects 3287 malicious packages from the malicious benchmarks with an 89.6% accuracy, while reporting the second lowest number of false positives from the benign benchmarks with a 91.1% accuracy.

TABLE III: Comparison with state-of-the-art tools using precision, recall, f1-score, fp-rate and accuracy

Technique	Precision	Recall	F1-Score	FP-Rate	Accuracy
Hercule	0.966	**0.933**	**0.949**	0.089	**92.721**
MalOSS	0.867	0.834	0.850	0.345	78.557
GuardDog	**0.978**	0.687	0.807	**0.042**	76.027
Bandit4Mal	0.701	0.716	0.708	0.826	56.968

Table III captures the overall accuracy of each tool. High precision implies that among the flagged packages, the majority are true positives. GuardDog reports the highest precision of 0.978 with Hercule having the second highest of 0.966. Although GuardDog does not detect most of the malicious packages, due to its low number of false positives, the precision is high. Hercule can detect more malicious packages than GuardDog and uses integrity analysis to reduce the number of false positives. High recall implies that among the malicious packages, the technique can identify most of them. Hercule demonstrates a high recall rate of 0.933 compared to MalOSS and GuardDog. Bandit4Mal analysis flags any package matching certain patterns, which is also why it has the lowest precision of 0.701.

For datasets with a high imbalance between classes, known as the *imbalance classification* problem [19], the more accurate metric for effectiveness is the F1-score. F1-score captures the harmonic mean of the precision and recall. A high F1 score indicates the strong overall performance of a binary classification task. Hercule reports the highest f1-score of 0.949, outperforming the rest with a significant margin. This is because of its ability to analyze **inter-package** data flows to detect malicious behavior.

We also report the false positive rate (FP-Rate) in Table III, which indicates the effectiveness in practice. A lower false positive rate is preferred to reduce developer fatigue. GuardDog reports the lowest FP-Rate of 0.042 while Bandit4Mal has the highest. Since Bandit4Mal employs simple pattern-based matching, this observation is not surprising. Hercule reports the second best of 0.089 using an integrity analysis to remove false positives.

> **RQ1:** Comparison with multiple state-of-the-art tools shows that Hercule outperforms existing techniques with an f1-score of **0.949**.

C. Evaluation: Contribution of each Component

We evaluate the contribution of each component in our proposed approach using each data-set we have evaluated on. Table IV summarizes the analysis of Hercule on the evaluated Python packages. Column "#Pkgs" represents the total number of packages in each data-set and column "#S" represents the total number of packages for which Hercule identifies the source repository. Columns "#I", "#M" and "#C" represent the total number of packages which have violated the source integrity, the total number of packages detected with malicious behavior and the total number of packages identified as compromised, respectively.

Hercule correctly identifies the label for each package with an accuracy of 92.721%. The integrity analysis captures packages with discrepancies in the distributed package. Hercule reports 3 and 113 false positives in popular packages and trusted packages, respectively. We investigated the reasons for the false positives. Hercule flagged 3 of the popular packages as integrity violations. Some of these packages (i.e. `.grpcio-status`) do not provide the source repository

TABLE IV: Contribution of each component in HERCULE to accruactely detect malicious packages

Data-Set	# Pkgs	#S	#I	#M	#C
MalOSS	242	101	219	209	6
BackStabber	345	147	291	310	9
MalRegistry	2985	1433	2807	2678	25
Popular Packages	100	89	30	1	1
Trusted Packages	1200	1081	1023	113	0
Overall	4872	2851	4370	3311	41

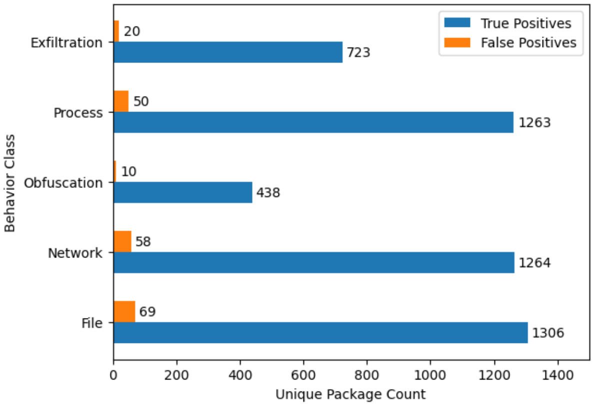

Fig. 3: Accuracy of each Behavior Class

information, hence, all files in these packages are considered to have failed the integrity analysis. However, among the projects for which HERCULE identified the source repository, new files were generated as part of the build process, such as `setup.py` and `version.py`. A few others include cache files from test directories.

The malicious behavior analysis of HERCULE accurately identifies malicious packages among the malicious dataset with an accuracy of 89.6%. For the benign packages HERCULE also reports an accuracy of 91.1%, reflecting the analysis capability to differentiate the benign from malicious behavior. This is because the malicious behavior analysis in HERCULE is combined with the integrity analysis which prunes alerts based on the source locations that do not violate the integrity. For example, the Python package `requests` exhibits the behavior of data communication with external sources, which is the intended behavior of the package. CODEQL reports 2314 alerts on the package, however using the integrity analysis HERCULE filters out all 2314 alerts since they are not reported on source locations that violate integrity analysis.

Lastly, the transitive dependency analysis flags 40 compromised packages in total. Among the packages `learninglib:0.1`, `learninglib:0.2` and `mllearnlib:0.7`, do not exhibit any malicious behavior within the package, however, all three packages explicitly download a malicious package (i.e. `maratlib:0.6`) as one of its dependencies.

We also analyzed the contribution of each behavior class to the high precision and recall of HERCULE. Figure 3 depicts the number of false positives and true positives flagged by each of the Behavior Classes we have encoded in CODEQL. The majority of malicious packages are identified by the File and Network classes, indicating that most packages in our benchmarks establish remote network connections. This is expected, as many malicious packages either send exfiltrated data to a remote server or create a connection to load their payload.

The majority of false positives were reported from the File class, suggesting that file system manipulation behavior is frequently observed in the benign packages within our dataset. Further investigation revealed that many false positives stem from a behavior pattern where test cases create and modify existing ones. These packages also lack repository information. Improving metadata extraction to accurately identify the source repository could help reduce such false positives.

RQ2: All three analyses enhance HERCULE's ability to accurately label packages, achieving manageable false positives while maximizing the detection of malicious packages in our studied dataset.

D. Evaluation: Precision and Reducing False Positives

One core component of HERCULE is the *integrity analysis* which we use to prune false positives. The discrepancies between the distributed package and the source repository have been studied previously in LASTPYMILE [8]. However, in our approach we use an AST level differentiation to precisely and efficiently identify the discrepancies compared to a hash-based system as proposed in LASTPYMILE. We compare the impact of having an integrity analysis, by comparing it with a baseline tool such as BANDIT4MAL which generates a vast number of alerts that match its detection rules. We evaluate the reduction of false positives on benign packages. HERCULE* represents the combination of BANDIT4MAL with our AST based integrity analysis used to filter out false positives.

TABLE V: Impact of integrity analysis captured in terms of number of alerts pruned using hash-based and AST-based techniques

Data-Set	BANDIT4MAL	LASTPYMILE	HERCULE*
Popular Packages	19900	3562 (-82.10%)	2241 (-88.74%)
Trusted Packages	44353	24309 (-45.19%)	8553 (-80.72%)
Overall	64253	27871 (-56.62%)	10794 (-83.20%)

Table V summarizes the number of alerts pruned by different implementations of the integrity analysis. Column BANDIT4MAL shows the total number of alerts generated for all the packages in a given data set. Columns LASTPYMILE and HERCULE* represent the filtered number of alerts in the form of `x(-y\%)`, where `x` is the total number of alerts remaining after pruning and `y` is the reduction percentage.

From the results of the analysis, it is evident that blindly using static analysis to match patterns can create a large number of false positives. Additional analysis, such as differential analysis can help to prune such false positives. LASTPYMILE [8] generates a database of hashes for every commit and compares the hashes in the distributed package to find files that do not match any of the hashes. This process is expensive as it has to iterate through all the commits and is sensitive to even the slightest space change in the file. Hercule* employs an AST differentiation that is robust to formatting changes such as additional spaces, tabs, and special character changes.

AST-based differentiation prunes 83.20% of false positives in both popular and trusted packages, allowing to achieve a manageable number of false positives. Hercule* implements a lightweight differentiation based on a single commit identified as the commit matching the version number of the package, while LastPyMile differentiates with all commits in the repository. On average LastPyMile needs more than 60 minutes to differentiate modified files, mainly because of the complexity of identifying phantom (i.e. modified or new) files which require comparison with all commits in the repository. For the same dataset Hercule* differentiate modified files and prune alerts on average within 5 minutes. Using a lightweight differential analysis Hercule* was able to achieve the highest overall prune ratio.

RQ3: The lightweight AST differentiation-based integrity analysis achieves a pruning ratio of 83.20% over Bandit4Mal.

E. Evaluation: Efficiency of Hercule

We also evaluate the efficiency of Hercule in terms of the time duration taken to classify a package as malicious or benign. Figure 4 depicts the violin plot for the time duration taken for Hercule to complete, across each dataset in our evaluation. The violin plot depicts the distribution of the time duration, capturing both the statistical values (i.e percentiles) and the density as well. Hercule on average is able to detect malicious packages in 3 minutes across all datasets as shown in Figure 4. The average time taken by Hercule to detect a malicious package is 1.84 minutes for the BackStabber dataset, 1.9 minutes for MalOSS, and 1.56 for the MalRegistry dataset. The average time to identify a benign package is 3.07 minutes for the top 100 popular packages and 6.32 minutes for packages from trusted organizations.

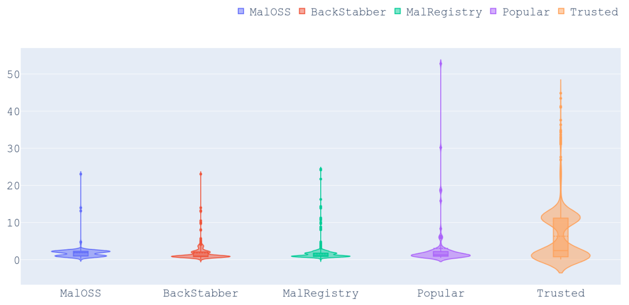

Fig. 4: Performance of Hercule across different dataset

The analysis time for benign packages requires a significantly longer time compared to the packages in the malicious benchmarks. This is mainly due to the differences in the average file size of packages in the benign dataset compared to the malicious benchmarks (Ref Table I). Hercule requires more than 10 minutes for 10 of the packages in our top-100 dataset, including *scipy, virtualenv, numpy and pandas*. These packages represent the largest packages in the top-100 dataset in terms of size and number of files. For instance, the *scipy* package has 6983 files amounting to 230701 lines of code.

RQ4: Hercule completes the analysis of malicious packages in 1.61 minutes and benign packages in 6.08 minutes.

V. Case Study: New Malware Detected on PyPI

In this section we briefly discuss our experience in finding new malware using Hercule by randomly scanning packages from the PyPI package registry. All of these malware packages have been confirmed by the PyPI security team and removed from the registry.

`wikipedia-scraper-in` is a product that allows the user to generate data by crawling web pages. Similar packages for popular websites such as YouTube, Facebook, and LinkedIn are published in PyPI registry. A manual inspection of `wikipedia-scraper-in` shows that the package only imports a library called `btstudio` and invokes an API call. When analyzed by a state-of-the-art tool, such as Bandit4Mal or MalOSS, this would not raise any flags. Hercule first runs an integrity analysis to detect code not part of the source repository. However, the package does not include information to establish a mapping with a source repository, hence integrity analysis flags all files as suspicious. Using our malicious behavior analysis, Hercule finds a suspicious control flow starting from `wikipedia-scraper-in`, leading to a method invocation in `btstudio` for the `subprocess.run` method, which starts a separate process (specifically, invoking a call to execute a Windows executable which resides within the dependent package `btstudio`). We note that Hercule detects a flow from `wikipedia-scraper-in` to `btstudio`, instead of simply matching for code patterns that include `subprocess.run`.

Upon detecting the malicious behavior, we further verified it by uploading the binary file to `VirusTotal` [7], which flagged the file as a Trojan virus, confirming its malicious nature. We then collected all packages from the same user, who had uploaded 136 packages to the PyPI registry that followed a similar pattern. Among these 136 packages, four served as payload distributors, while the remaining packages directly or indirectly invoked APIs to execute the binary file.

We extracted the binaries and analyzed each one separately. We analyzed two binaries, `datakund.exe` and `datakund-btstudio.exe`, which are both Windows executables. They are essentially the same, however, parts of the code in `datakund-btstudio.exe` are more advanced, indicating an evolution from `datakund.exe` to

[7] https://www.virustotal.com

`datakund-btstudio.exe`. For example, some functions have included exception handling likely an indication of code iteration. The two executable files are bots used by the adversarial user. Once deployed, the bots execute in the background and establish a connection to listen on a port for commands sent by the command server via GET and POST requests.

VI. Practical Impact

We evaluate the effectiveness of Hercule in reducing false positives reported by the industry-grade supply chain detection tool Macaron [20][8], developed by Oracle Labs. Our analysis focused on the 1000 Python packages recently released in the PyPI registry as of October 10, 2024. The malicious metadata check in Macaron identified five packages as potentially malicious, as shown in Table VI. One package identified as a true positive was flagged by both tools, while the remaining packages that were false positives were classified as benign by Hercule. This demonstrates that Hercule can effectively complement the metadata analysis in a tool like Macaron, enhancing its practical effectiveness by reducing its false positives.

TABLE VI: Reducing False Alerts of Macaron

Package Name	Version	Macaron	Hercule	Duration (min)
cmdb-worker-pckg	1.0.0	✗	✗	1.672
hacking-shield	0.2.3	✗	✓	0.788
fruitspace.py	1.0.2	✗	✓	1.8
one-chat-api	0.2.8	✗	✓	0.727
httpsy	0.3.0	✗	✓	0.775

VII. Limitations and Threats to Validity

A. Limitations

Hercule mainly relies on the integrity of the source repository to precisely differentiate modifications. If the source repository itself is compromised by means of social engineering attacks, the analysis would fail to flag the package. Even if the malicious behavior identifies the injected code, Hercule would filter it out since it does not violate the integrity of the source repository. In addition, the current prototype of Hercule only analyses data-flow within Python files and does not capture malicious code written in C/C++ code that is interacting with the Python code.

B. Threats to Validity

While we consider the most downloaded and trusted packages to be benign, they may still contain undetected malicious code. Additionally, these packages do not encompass the entirety of benign packages. Therefore, further evaluation using a larger dataset of benign packages is necessary to generalize our findings. To address the potential limitations in the generality of benign packages, we collected over 1000 packages from two distinct sources: the most downloaded packages and those from reputable organizations.

The behavior specifications are based on previously identified malicious packages. While we have generalized these specifications in our behavior analysis, they remain dependent on the packages used for our evaluation. To mitigate this issue, collecting additional malicious packages to create an independent dataset would be beneficial. However, there is a scarcity of publicly available packages with verified ground-truth labels. To address this validity concern, we evaluated our approach using three widely studied benchmarks. These datasets were curated at different times and include a diverse set of packages.

Lastly, our implementation of LastPyMile may differ from the original implementation provided by the authors, which could yield different results. However, the authors' implementation only works for packages currently hosted in the PyPI registry, limiting our ability to evaluate malicious packages that have already been removed. To address this issue, we will release our implementation of LastPyMile alongside our artifact release, allowing the research community to replicate our results.

VIII. Related Work

A. Software Supply Chain Attacks

As reliance on open-source software grows, attackers find more opportunities to contaminate software artifacts by targeting software supply chains [21]. A recent study shows various ways that adversaries spread malware by compromising the open-source software supply chain [17]. According to this study, adversaries either compromise existing packages or create and publish malicious packages that mimic popular benign packages. This study also provides information about the attacker's goals, methods to trigger the malicious behavior, and common obfuscation techniques. Bagmar et al. [22] performed an extensive analysis of the PyPI ecosystem, and showed issues that allow an attacker to run malicious code at package installation time. Hercule supports these scenarios identified in this study: (1) it verifies the integrity of the package w.r.t. the source repository to detect existing compromised packages, and (2) it detects fake packages that pretend to be benign using data-flow analysis techniques. Moreover, Hercule is the first tool that detects more advanced attacks, such as Divide and Hide [13], which have not been studied in the previous work [12].

B. Integrity Violation

To detect potentially malicious packages that are published on public registries, one approach is to find discrepancies between the source code of the package and its corresponding repository [23], [24], [8]. The assumption in these works is that if the source code of a project is tampered with in the build process by a malicious actor, the injected malicious code that is present in the resulting artifact will not be present in the corresponding code repository. Identifying the source code repository for an artifact is not always straightforward and several techniques are proposed to improve the accuracy of such mappings [25], [20]. The supply chain security community has come up with frameworks, such as SLSA [26] to establish a trusted link from an artifact to a commit

[8] https://github.com/oracle/macaron

in its corresponding source-code repository and publishing CI pipeline using verifiable provenances. Unfortunately, the current adoption of such provenances in open-source projects is low [20]. Moreover, such provenances are not enough for detecting malicious packages. A malicious package can compromise the integrity of the environment after it is installed by modifying other packages that are installed on the system. More recently, cryptographic solutions [27], [28] are proposed to ensure integrity of the software. These solutions aim to provide users the ability to verify the entire supply chain from development to deployment. While the proposed solutions provides assurance, it requires the package developers to adapt their proposed solutions. Different to this approach our integrity analysis is lightweight and can be utilized by the end user irrespective of the project development process.

C. Supply Chain Malware Detection and Prevention

Duan et al. [9] combine metadata analysis with static and dynamic analysis techniques to shortlist suspicious packages and manually determine if they are malicious. Similarly, we use static analysis to find security-sensitive data flows. However, our static analysis detects data flow across packages, enabling us to detect advanced non-trivial attacks, such as Divide and Hide [13]. Sejfia and Schäfer [10] use machine learning to find potentially malicious npm packages using features such as fingerprinting personal information, presence of minified code, binary files and usage of specific APIs, excluding packages that have reproducible builds. To reduce false negatives, this work also uses a clone-detection technique to find discrepancies to detect injected malicious code. More recently Froh et al. [29] proposed an approach to detect malicious updates to the packages. While Hercule differentiates between the distributed package and the source repository, this work differentiates between previous versions of the same package. The two techniques are complementary to each other since they focus on two different dimensions of how the supply chain can be compromised. Our work is also complementary to Macaron [20], which provides an extensible framework to detect supply chain attacks. Hercule can be added as a check to Macaron to detect and prevent the described attacks during the development and deployment of artifacts.

Gonzalez et al [30] proposed to detect malicious commits that are injected into open source repositories, using a rule based decision model. The solution aims to prevent the integration of malicious code commits, thus improving the supply chain during development. Our work is complementary with the solution, which detects malicious intent at the development stage of the project while Hercule focus on detecting malicious behavior during deployment. Vasilakis et al [31] proposed an active learning to eliminate the possibility of integrating vulnerable library components in to the environment. The proposed solution aims to learn the correct functional behavior of the desired third party library, and regenerate a newer safer version. This line of work is orthogonal to Hercule which detects malicious behavior to prevent supply chain attacks, while this work aims to eliminate supply chain attacks by regenerating the code.

D. Program Analysis

Program Analysis have been widely studied for detecting logical errors [32], [33], identifying security issues [34], [35], program repair [36], malware detection [37], sand boxing [38] and supply chain protection [29]. Atkinson et al [33] discussed the need for whole program analysis to improve automated support for program understanding. They introduce the challenges to reason about a program's behavior by simply examining individual modules alone, and the need to have analyses that work on the whole program. Balzarotti et al [39] proposed a vulnerability analysis approach that characterizes both the extended state and the intended workflow of a web application. The proposed analysis also take into account inter-module relationships and the interaction with external sources. Ferreira et al [38] proposed lightweight permission system that protects Node.js applications by enforcing package permissions at runtime. They proposed lightweight sand boxing strategy that combines dynamic checks with static analysis. Mantovani et al [40] proposed to use program analysis to detect security vulnerabilities in decompiled binaries. Similar work has also been proposed for JavaScript code [41] and C code [42].

Different to existing work using program analysis, Hercule use program analysis to detect software supply chain attacks. Specifically, we use inter-package data-flow analysis to identify malicious behavior patterns.

IX. Conclusion

We presented a *inter-package analysis* approach that combines three analyses to identify malicious packages with high precision and high recall. Our approach incorporates an integrity check based on AST differentiation analysis, that can identify discrepancies between the distributed artifacts and the source repository. We then utilize CodeQL to detect malicious behavior using data-flow analysis. Lastly, we implement a transitive dependency analysis to identify malicious packages installed as part of the dependency resolution. Our *inter-package* analysis was able to outperform existing state-of-the-art techniques to detect malicious packages with high precision and fewer false positives.

Our tool and experiment setup is available via:
https://github.com/rshariffdeen/Hercule

Acknowledgments

This research was supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity RD Programme (Fuzz Testing NRF-NCR25-Fuzz-0001). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

References

[1] "9th Annual Report of SonaType: State of the Software Supply Chain," https://www.sonatype.com/state-of-the-software-supply-chain/open-source-supply-and-demand, accessed: 2024-03-19.

[2] "PyPI Inbound Malware Volume Repor," https://blog.pypi.org/posts/2023-09-18-inbound-malware-reporting/, accessed: 2024-03-19.

[3] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020, pp. 509–514.

[4] "Dependency confusion: How i hacked into apple, microsoft and dozens of other companies," https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610, accessed: 2023-10-31.

[5] "Blog article of Github repository confusion attack," https://apiiro.com/blog/malicious-code-campaign-github-repo-confusion-attack/, accessed: 2024-03-01.

[6] "Official website of PyTorch," https://pytorch.org, accessed: 2023-10-27.

[7] "Namespace Confusion Attack in PyTorch," https://www.bleepingcomputer.com/news/security/pytorch-discloses-malicious-dependency-chain-compromise-over-holidays/, accessed: 2023-10-27.

[8] D.-L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. 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*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 780–792. [Online]. Available: https://doi.org/10.1145/3468264.3468592

[9] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/towards-measuring-supply-chain-attacks-on-package-managers-for-interpreted-languages/

[10] A. Sejfia and M. Schäfer, "Practical Automated Detection of Malicious Npm Packages," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1681–1692. [Online]. Available: https://doi.org/10.1145/3510003.3510104

[11] "Github repository for Bandit4Mal scanner," https://github.com/lyvd/bandit4mal, accessed: 2023-10-31.

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

[13] "Blog article explaining divide and hide," https://www.endorlabs.com/blog/divide-and-hide-how-malicious-code-lived-on-pypi-for-3-months, accessed: 2023-11-06.

[14] "Github repository for Bandit scanner," https://github.com/PyCQA/bandit, accessed: 2023-10-31.

[15] "GuardDog Tool," https://github.com/DataDog/guarddog, accessed: 2024-09-25.

[16] M. Ohm, L. Kempf, F. Boes, and M. Meier, "Supporting the detection of software supply chain attacks through unsupervised signature generation," *arXiv preprint arXiv:2011.02235*, 2020.

[17] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds. Cham: Springer International Publishing, 2020, pp. 23–43.

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

[19] X. Guo, Y. Yin, C. Dong, G. Yang, and G. Zhou, "On the class imbalance problem," in *2008 Fourth International Conference on Natural Computation*, vol. 4, 2008, pp. 192–201.

[20] B. Hassanshahi, T. Nhan Mai, A. Michael, B. Selwyn-Smith, S. Bates, and P. Krishnan, "Macaron: A Logic-based Framework for Software Supply Chain Security Assurance," in *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED, 2023. [Online]. Available: https://doi.org/10.1145/3605770.3625213

[21] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1509–1526.

[22] A. Bagmar, J. Wedgwood, D. Levin, and J. Purtilo, "I Know What You Imported Last Summer: A study of security threats in thePython ecosystem," *CoRR*, vol. abs/2102.06301, 2021. [Online]. Available: https://arxiv.org/abs/2102.06301

[23] P. Goswami, S. Gupta, Z. Li, N. Meng, and D. Yao, "Investigating The Reproducibility of NPM Packages," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 677–681.

[24] S. Scalco, R. Paramitha, D.-L. Vu, and F. Massacci, "On the feasibility of detecting injections in malicious npm packages," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ser. ARES '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3538969.3543815

[25] D.-L. Vu, "py2src: Towards the automatic (and reliable) identification of sources for pypi package," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2021, pp. 1394–1396.

[26] "Supply-chain Levels for Software Artifacts (SLSA)," https://slsa.dev, accessed: 2024-3-19.

[27] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software signing for everybody," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2353–2367. [Online]. Available: https://doi.org/10.1145/3548606.3560596

[28] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1393–1410. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias

[29] F. N. Froh, M. F. Gobbi, and J. Kinder, "Differential static analysis for detecting malicious updates to open source packages," in *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3605770.3625211

[30] D. Gonzalez, T. Zimmermann, P. Godefroid, and M. Schäfer, "Anomalicious: automated detection of anomalous and potentially malicious commits on github," in *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '21. IEEE Press, 2021, p. 258–267. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP52600.2021.00035

[31] N. Vasilakis, A. Benetopoulos, S. Handa, A. Schoen, J. Shen, and M. C. Rinard, "Supply-chain vulnerability elimination via active learning and regeneration," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1755–1770. [Online]. Available: https://doi.org/10.1145/3460120.3484736

[32] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[33] D. Atkinson and W. Griswold, "The design of whole-program analysis tools," in *Proceedings of IEEE 18th International Conference on Software Engineering*, 1996, pp. 16–27.

[34] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 459–465.

[35] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. USA: USENIX Association, 2005, p. 18.

[36] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 691–701. [Online]. Available: https://doi.org/10.1145/2884781.2884807

[37] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 116–127. [Online]. Available: https://doi.org/10.1145/1315245.1315261

[38] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Containing malicious package updates in npm with a lightweight permission system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1334–1346.

[39] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna, "Multi-module vulnerability analysis of web-based applications," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 25–35. [Online]. Available: https://doi.org/10.1145/1315245.1315250

[40] A. Mantovani, L. Compagna, Y. Shoshitaishvili, and D. Balzarotti, "The convergence of source code and binary vulnerability discovery – a case study," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 602–615. [Online]. Available: https://doi.org/10.1145/3488932.3497764

[41] T. Brito, M. Ferreira, M. Monteiro, P. Lopes, M. Barros, J. F. Santos, and N. Santos, "Study of javascript static analysis tools for vulnerability detection in node.js packages," *IEEE Transactions on Reliability*, vol. 72, no. 4, pp. 1324–1339, 2023.

[42] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 544–555. [Online]. Available: https://doi.org/10.1145/3533767.3534380