Bad Snakes: Understanding and Improving Python Package Index Malware Scanning

Duc-Ly Vu
Chainguard and FPT University
lyvd@fe.edu.vn

Zachary Newman
Chainguard
zjn@chainguard.dev

John Speed Meyers
Chainguard
jsmeyers@chainguard.dev

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00052

***Abstract*—Open-source, community-driven package repositories see thousands of malware packages each year, but do not currently run automated malware detection systems. In this work, we explore the security goals of the repository administrators and the requirements for deploying such malware scanners via a case study of the Python ecosystem and PyPI repository, including interviews with administrators and maintainers. Further, we evaluate existing malware detection techniques for deployment in this setting by creating a benchmark dataset and comparing several existing tools: the malware checks implemented in PyPI, Bandit4Mal, and OSSGadget's OSS Detect Backdoor.**

We find that repository administrators have exacting requirements for such malware detection tools. Specifically, they consider a false positive rate of even 0.1% to be unacceptably high, given the large number of package releases that might trigger false alerts. Measured tools have false positive rates between 15% and 97%; increasing thresholds for detection rules to reduce this rate renders the true positive rate useless.

While automated tools are far from reaching these demands, we find that a socio-technical malware detection system has emerged to meet these needs: external security researchers perform repository malware scans, filter for useful results, and report the results to repository administrators. These parties face different incentives and constraints on their time and tooling. We conclude with recommendations for improving detection capabilities and strengthening the collaboration between security researchers and software repository administrators.

***Index Terms*—Open-source software (OSS) Supply Chain, Malware Detection, PyPI, Qualitative Study, Quantitative Study**

I. Introduction

The first computer worm immediately prompted the corresponding malware detection software [1]. Since then, entire textbooks [2], [3] and conferences [4] have been devoted to the automated detection of unwanted software, to remove it from computer systems or preventing it from arriving at all.

Malware detection may have a role to play in overcoming "software supply chain attacks." Such attacks do not exploit the target software itself, but rather vulnerabilities or backdoors introduced during the development, building, and deployment of software [5]. These attacks can take many forms, from an update introducing malicious behavior in a formerly-benign software dependency to "trusting trust" attacks [6] where a malicious compiler inserts a backdoor into the software it compiles (including itself). An effective malware detection system could halt the propagation of such attacks up the OSS supply chain.

Programming language-specific package repositories (such as npm, PyPI, or RubyGems) are both a target for supply chain attacks and a site from which to mount defenses. Developers use such repositories to manage their software's dependencies but, in doing so, incur supply chain risk: recent academic work [7]–[9] quantifies and enumerates attacks using these repositories. Other academic works [10], [11] propose defenses for specific classes of attacks, and repositories [12] and nonprofit organization [13] are deploying techniques like software signing and multifactor authentication.

However, deployments of malware-detection solutions in these settings have met with limited success in integrating into a package repository. None of these malware detection techniques have been successfully adopted by PyPI. These repositories differ from the traditional settings for such malware scanners: antivirus software and intrusion detection systems. Consequently, techniques designed for those settings may not be suitable for a package repository. For instance, in 2020, the Python Software Foundation oversaw the creation of a malware scanning pilot (called PyPI malware checks) for Warehouse, the application which serves PyPI [14]. However, two years later, administrators disabled its malware checks, as reported by one of our interviewees.

A. Objectives and methods

In this work, we seek to understand requirements for malware detection tools in the package repository setting as well as the fitness of current malware detection techniques for this application. To do so, we study the Python ecosystem and the PyPI repository, focusing on Python-specific malware detection techniques at scale. We conduct interviews (see Section III) with contributors to and administrators of PyPI, along with a software repository security researcher. In Section IV, we create a Python malware benchmark dataset and evaluate existing automated malware detection techniques. We analyze the results of the experiments in Section V.

B. Findings and recommendations

In our user research (Section III), we find that the motivation and even the setting for repository-side malware scanning differ in practice from those assumed in the literature. Administrators want to mitigate low-effort, high-volume attacks with minimal effort of their own; not remove all malware. Additionally, repository administrators possess demanding criteria for a practical deployment of automated checks: false positive rates must be virtually zero.

Benchmarks show that the studied automated detection methods unambiguously fail to meet these criteria, with false positive rates of 15 % or higher. While authors of these tools often note a high false positive rate, the exact rates are either not given, or given for different datasets. Our analysis provides a uniform dataset, allowing comparison across tools. Further, it quantifies these rates and compares them to the requirements of administrators unearthed in interviews.

Instead, the interviews revealed that the true "malware checks" in the PyPI ecosystem involve external security researchers with their own incentives, who find and report malware to the administrators [15]. This socio-technical malware detection system prevents PyPI from needing its own automated malware detection system.

We propose recommendations for researchers and repository administrators (Section VI) in order to improve the security of these ecosystems. In short, most researchers should primarily focus on other efforts to secure package repositories in collaboration with the administrators, who are in the best position to understand their security needs. In automated malware detection, researchers should focus on aligning the incentives of repository administrators and external researchers, and engage with malware detection as a socio-technical system. All parties should attempt to facilitate smoother collaboration and deployment with better tooling and data.

C. Contributions

This work contributes:

- A qualitative analysis of the security goals and logistical requirements of the main Python package repository.
- A dataset of malicious and benign Python packages.
- A comparison of recent automated Python malware-detection tools (previous evaluations did not compare tools on the same dataset).
- Recommendations for both researchers and repository administrators.

D. Security and ethical considerations

This work adheres to the ACM Publications Policy on Research Involving Human Participants and Subjects [16]. All interviewees gave informed consent prior to their interviews and approved the characterization of their responses presented herein; they received a copy of our findings and the opportunity to present feedback before submission. They are identified only by their background and not their name or institutional affiliation. Even still, it may be possible to identify them from unique details; the interviewees acknowledge this risk and consider themselves "public figures."

The malware samples used for analysis are maintained by their collectors [17], [18], and are available on request by the respective researchers to prevent their misuse.

II. Background

Software supply chain attacks have been on the rise over the past ten years [19], [20]. In particular, attackers have increasingly exploited the trust that software developers and downstream consumers have implicitly placed on open-source infrastructure, especially package repositories. Fortunately, repository administrators and a number of allied parties have begun implementing defenses against attacks on the open-source software supply chain, including the automated detection of malicious packages [8]. However, despite decades of research and engineering on general automated malware detection software, open-source software malware detection in this setting is at an early stage in both theory and practice. The nascent state is due in part to a lack of consistent benchmarking and to differences between the goals of malware detection researchers and the needs of repository administrators.

A. Software supply chain attacks

The interval between Ken Thompson's theoretical discussion of backdoored compilers [6] to arguably the first actual software supply chain attack [21] was nearly twenty years. These days, software producers and consumers are fortunate when there are twenty days between known attacks, not to mention the countless unreported attacks [20], [22]. In recent years, notable attacks have included, for instance, SolarWinds, in which the Russian intelligence services injected malware into a widely used network management software [23]. Malicious attacks on the software supply chain have also targeted open-source software producers and consumers. One particularly notable attack was on consumers of event-stream, a popular JavaScript package. In 2018, an attacker took control of the package and introduced malicious code to exfiltrate information from Bitcoin wallets. Though this package averaged nearly two million downloads per week, the attack went undiscovered for nearly two months [24].

a) Open source repositories have been targeted: Open-source software repositories, essentially stores of free and open-source (FOSS) components that facilitate code reuse, have become an important part of the software supply chain over the past couple of decades. Developers use such repositories to search, add, and manage dependencies for the software they write, which allows them to write more complicated software more quickly. For instance, PyPI [25], as of August 2022, contains almost 400,000 projects and nearly 4,000,000 releases. PyPI, like many registries, is governed by a non-profit foundation and is run by a handful of volunteers [26].

Attacks on the software supply chain have, unfortunately, but perhaps inevitably, therefore extended to open-source repositories and their users. Critics argue that such repositories facilitate overreliance on external dependencies, which in turn increases the attack surface for a supply chain attack and increases the leverage of an attacker able to compromise a popular package [27]. In 2022, PyPI removed over 12,000 unique projects, most of which were malware [28].

There are a number of classes of attacks on open-source package repositories. Typosquatting, in which an attacker relies on a user mistyping the name of a more popular package or confusing one package for another, has begun to receive analytical attention [10], [29]. So have account takeover attacks in which an attacker steals credentials or otherwise gains control over a package and then alters the package code [30]. Relatively less appreciated are the threats of "shrinkwrapped clones" [11], which duplicate existing packages in order to attract downloads before introducing malware, and "domain resurrection" attacks in which an attacker uses an abandoned email domain to take control of a package [7]. Attackers also compromise the repositories themselves and not just individual packages [22], [31]. Recent work has created a comprehensive taxonomy of attacks on the OSS supply chain [32].

Malicious packages can operate either at installation time or runtime. Install-time malware can compromise the developers of downstream dependents during development or their build machines during deployment [30]. This is often the case with Python, as Python packages execute arbitrary code on installation via `setup.py` files. Malicious Python packages may also exhibit runtime malicious behavior [30], [33]. Recent cases of "protestware" fit into this category [34], [35].

b) Repository defenses against supply chain attacks: Administrators have not stood idly by while attacks on registries have increased. Recently some major package repositories have implemented two-factor authentication for the maintainers associated with widely used packages as a way to reduce the likelihood of consequential account takeovers [36], [37]. Software signing is a promising technique for preventing attackers from deploying software via compromised accounts. Some repositories are considering integration with Sigstore [38], [39], a project that enables trusted parties to make authenticated claims about software artifacts [40].

These recent developments complement earlier efforts, some still ongoing, to secure repositories (including PyPI) using technologies like The Update Framework (TUF) to enable recovery in the event of compromise [41]–[43]. Reproducible builds [44], a set of techniques to verify that no vulnerabilities or backdoors have been introduced during the build process, is promising but appears hard to achieve for interpreted languages such as Python [45] or JavaScript [46].

Repository administrators have also experimented with scanning for malware [47], [48]. The next subsection discusses the current status of malware detection in OSS repositories.

B. Malware detection in PyPI and other Repositories

Supply chain attacks aim to distribute unwanted software to unwitting downstream dependents, so an automated system that could detect unwanted software could prevent such attacks. The automated malware detection literature, dating back decades, attempts to do exactly that. Here, we examine the approaches most relevant to the package repository setting.

a) Automated malware detection: Malware detection technologies aim to identify malicious or unwanted software in an automated way [49]–[51]. Typical use cases include antivirus software or intrusion detection systems. Some methods are signature-based: they try to match specific malware (for instance, by checking file hashes against a blocklist). Although signature-based approaches can catch specific malware precisely, they do not generalize to new malware samples, even when they are very similar to old ones.

In contrast, anomaly-based methods look for suspicious patterns or behaviors in software [52]. These methods include static and dynamic analysis. Static analysis techniques analyze software without running it—for instance, looking for suspicious imports in a binary's headers. Dynamic analysis techniques execute software (typically in an isolated sandbox environment) to observe actual behavior at runtime. These approaches generalize to unseen malware but risk false positives.

b) Software repository malware detection: In the literature, many approaches attempt to identify malicious packages in package repositories such as npm or PyPI [8], [53]–[55] or in the source code repositories such as GitHub [56], [57]. Most such approaches analyze different aspects of a package using metadata [10], [29], static [14], [58]–[60], or dynamic [8], [47], [61] analysis. These approaches use indicators such as abnormal network connections borrowed from traditional malware detection techniques [62]. *Metadata analysis* techniques examine package metadata to flag suspicious packages. For example, package names and popularity can indicate typosquatting or combosquatting packages [10], [29]. These research projects often present their work in the context of potential deployment by open-source repositories.

Commercial OSS malware scanners such as Sonatype's automated malware detection system [63] continuously monitor newly added packages for their maliciousness. However, commercial security companies often do not open-source or otherwise release their detection schemes, which prevents researchers from evaluating these approaches.

c) A gap between theory and practice: Many researchers advertise their malware detection techniques as a way to keep unwanted packages out of a repository on an ongoing, automated basis, with high true-positive rates as evidence of their tool's efficacy. However, such tools rarely acknowledge the unique requirements of the setting. The most important criterion is that a check must provide a "useful signal," and false positives can disqualify a check [14]. We observed that there is no research to understand if the requirements of a useful malware check by PyPI administrators would be satisfied by either academic or industrial tools. This motivates our exploratory case study.

III. What Do Experts Say About Scanning PyPI for Malware?

This study includes a qualitative, interview-based, user research component to understand better the challenges in scanning malware for OSS repositories and PyPI in particular. Individuals in the PyPI community, security researchers, and academics already have firsthand experience in scanning open-source packages for malware. Therefore, this study sought to benefit from their first-hand knowledge and better appreciate the promise and challenges of malware scanning in the context of an OSS repository [10], [11].

A. User research interview methodology

The research team conducted three interviews in July and August 2022. This section describes the objectives, the questionnaires and interview format, and the main findings.

a) Research Objectives: We initially sought to answer a narrow question: what are the performance properties of a technical system for malware detection in the context of a large OSS repository? This information was intended to provide input on what technical criteria ought to be used to distinguish practical and impractical approaches to malware detection for PyPI and other similar registries. The goal was to directly aid the PyPI administrators in assessing and improving a malware check system to be operated by PyPI. The research objective evolved as the interviews revealed new perspectives on OSS malware detection.

While interviewees did provide answers to the earlier, more narrow question, the research team ultimately settled on a broader question: what should be the characteristics of a socio-technical system that enables malware detection in the context of a large OSS repository? This framing shifted responsibility for malware detection solely from the shoulders of PyPI administrators to a combination of administrators, security researchers, and academics. Importantly, this perspective also de-emphasizes the technical characteristics of any particular scanning approach and highlights the need for multiple parties to perform malware scanning. More on this perspective will be included in the key findings section (Section III-B).

b) Interview Methodology: The interview questionnaire included five broad questions and a number of sub-questions.

1) What is the history of the current PyPI malware checks?
   - Who was involved?
   - Why did this initiative start?
2) What has been your experience, if any, with the current PyPI malware checks?
   - What has worked and why?
   - What has not worked and why?
   - Does PyPI currently enable the malware checks?
   - How much time do administrators spend on triage?
   - How many alerts do admins manually examine?
3) What are the current plans, if any, for improving the PyPI malware checks?
   - Who is involved?
   - What is the plan?
4) How do you judge the performance of a PyPI malware check system?
   - What criteria do you use?
   - What should the technical requirements be?
5) How would you judge a set of proposed improvements to the PyPI malware check system?
   - What are the interviewee's views on any proposal?

In short, the interview questionnaire largely focused on answering the narrow questions consistent with the project's initial research goal. The interviews were, however, semi-structured, which allowed for the interviewers and interviewees to shift topics as new insights emerged

All interviews were conducted over video teleconference and lasted approximately one hour. The research team did not record the interviews but did take detailed notes. The interviews were analyzed via thematic analysis [64]. A member of the research team identified common themes inductively and then grouped all interview evidence (i.e. quotations) into one or more of the identified categories. The themes are presented below in the following section. The team adhered to popular guidance on interviewing within the context of empirical software engineering research [64].

Because we chose to focus on a single ecosystem (PyPI), the number of individuals with direct experience scanning for malware was limited, which constrained our interviewee sample size. In particular, there are under ten people involved directly in the malware scanning activities in PyPI [26]. The first interviewee had extensive experience in computer security, including malware detection, and had participated in the design and building of the PyPI malware system. The second was closely involved with PyPI security operations, including responding to reports of malware on PyPI. A third interviewee, an academic security researcher, had extensive experience with OSS malware detection.

B. Interview key findings

We distilled four main findings, one related to the narrow technical questions about current malware detection performance for PyPI malware checks and three related to broad questions about OSS malware detection.

Current systems have too many false positives. The current PyPI malware checks produce an overwhelming number of false positives. According to the interviews with the PyPI administrators, this high rate of false positives reduces the usefulness of the system to such an extent that these checks have little to no value. The academic interviewee also emphasized that his own OSS malware research had often encountered issues with high false positive rates and so cautioned that any PyPI malware scanning would similarly deal with this problem. However, this does not indicate that all OSS repository malware scanning is a wasted effort. Both of the PyPI contributors interviewed explicitly mentioned that PyPI's experimentation with malware detection has been valuable, ultimately indicating the difficulty of the task.

Repository administrators are overextended. The burden on a single party, especially PyPI administrators directly, to scan PyPI for malware is prohibitively high [26], [29]. The current procedure for dealing with PyPI malware allocates about twenty minutes per week for a PyPI administrator to review malware reports from outside security researchers and removing true positive cases. PyPI sees about 20,000 new PyPI releases each week [65] and administrators, according to our interviews, respond to about 20 reports of malicious packages in that same time period. If an alert for a package takes about 1 minute to triage, even just the malicious packages consume the entirety of the time allotted for review of suspicious packages. The PyPI administrators have found that the vast majority of reports from outside researchers are true positives. Given

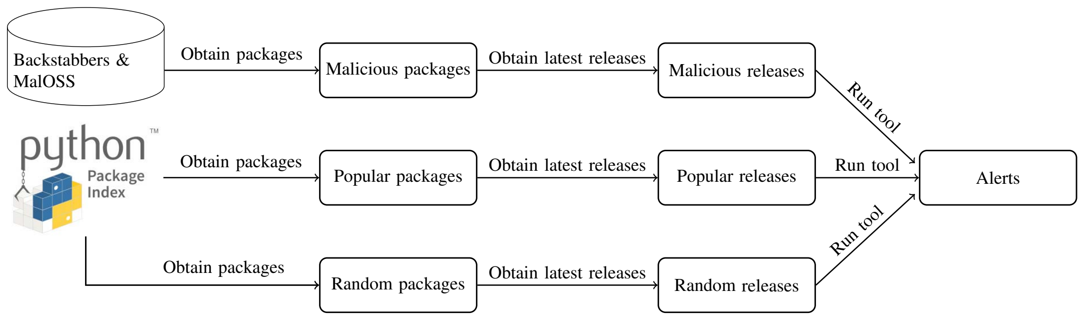


Fig. 1: Diagram of Benchmarking Python Malware Detection Tools on the PyPI packages

this, the interviewees were either reluctant or ambivalent about administrators themselves processing the numerous alerts from an automated system in their relatively scarce volunteer time.

Administrators have other security priorities. While, all things equal, these administrators want to eliminate malware from PyPI, they nonetheless emphasized that only a small portion of PyPI malware actually harms users, noting that many malicious packages have almost no downloads after accounting for automated mirrors of the entire repository. In a typical example [66], researchers discovered a number of malicious packages with under 200 downloads on average over 20 days; according to our interviewees, this is consistent with the rate at which automated mirrors scan the repository. Further, the administrators have little interest in the arms race of automated malware detection [67]: they believe that a sophisticated actor can always subvert such checks. Instead, they prefer to use malware detection techniques only to eliminate frequent, low-effort attacks. This allows them to focus on what they consider to be higher-impact security initiatives, like multifactor authentication or cryptographic package signing.

A more practical approach to PyPI malware emphasizes collective protection. Because PyPI administrators believe that building and operating their own malware detection system would lead to unacceptably high false positive rates (a belief consistent with the research findings presented later), the PyPI administrators have embraced an alternative system. In this alternative system, which we label a socio-technical malware detection system, there exists a symbiotic relationship between PyPI and software security companies and individuals performing security research. A set of researchers, some at software security companies and some as individuals, independently scan PyPI for malware. While the exact number of security researchers and companies engaged in this activity is unknown, there are several companies that publicly report this activity including Checkmarx, Datadog, JFrog, Phylum, and Sonatype. These parties create and maintain their own technical infrastructure to perform automated scans of PyPI. Importantly, these parties also undertake the tedious and time-consuming work of sifting through the results, identifying false positives and true positives. Once true positives are identified, these security researchers email the PyPI administrators and identify the offending package or packages. The PyPI administrators ask reporting parties to point to the offending package, package version, and code line by using a web app designed for this purpose: https://inspector.pypi.io/.

These parties, especially the software security companies, likely perform this scanning to gain attention for their companies, products, and services. PyPI administrators benefit because these parties maintain their own scanning infrastructure and undertake the tedious work of sifting through false positives themselves. This system has notable advantages: it is free from complicated and burdensome contracts and involves the willing participation of all parties. However, this same freedom also means that there are no explicit performance and reliability guarantees of this overall system. Nevertheless, one PyPI administrator has publicly declared that in 2022 there were over 12,000 unique projects removed from PyPI, and that these were instances of spam, typosquatting, dependency confusion, exfiltration, or malware. That same administrator notes that there were 27,000 projects removed in 2021, 500 in 2020, 65 in 2019, 137 in 2018, and 38 in 2017 [28].

C. *Technical requirements for malware detection systems*

We asked the interviewees about the requirements for an automated, repository-side malware scanning system. First, they reported that the scanner must have a binary outcome, possibly via a threshold over a numerical score. Approaches that simply return raw alert data and require ad-hoc data analysis, according to the interviews, are too time-consuming. However, administrators do prefer to review any packages they take down manually, so further details must be accessible. Second, engineering resources are limited, though computational ones are less so. Consequently, a scanner that executes code via dynamic analysis is unrealistic due to security concerns around maintaining a secure sandbox even though resources to run such a scanner at scale are available. Third, any approach must effectively have a false positive rate of zero, though this is an extremely demanding requirement.

IV. How do malware detection approaches perform?

In this section, we experimentally assess the reported false positives issue associated with Python malware detection that the interviewees discussed. To do this, the research team created a benchmark dataset containing both malicious and

TABLE I: Python malware detection tools considered for benchmarking.
The selected tool must satisfy all the conditions in the three columns.

Approach	**Source code available**	**Anomaly-based detection**	**Detection rules available**
Bandit4Mal [59]	✓	✓	✓
OSSGadget's OSS Detect Backdoor [58]	✓	✓	✓
PyPI Malware Checks [14]	✓	✓	✓
Aura [68]	✓	✓	
JFROG XRAY [69]		*?*	
OSSF package-analysis [47]	✓	✓	
MalOSS [8]	✓	✓	
pypi-scan [70]	✓	✓	
Sonatype automated malware detection [63]		*?*	
TypoGard [10]			✓

benign PyPI packages and then evaluated the current PyPI malware checks in addition to two other selected Python malware detection approaches in Table I.

A. Approach

Figure 1 depicts the benchmarking methodology. We first chose a set of Python malware detection approaches to benchmark using the criteria discussed below. We then collected both malicious and benign PyPI packages. The benign dataset can be further subdivided into popular packages and random PyPI packages. Finally, we scanned each package with each tool, recording all alerts produced. An alert associated with a malicious package is considered a true positive, while an alert on a benign package is a false positive.

a) Tool selection: We used the work of Ladisa et al. [32] and surveyed existing literature to create a list of candidate Python malware detection tools. Notably, we did *not* include vulnerability scanning tools such as Security.py, Hawkeye, and Salus because these packages (by design) tend to have many findings on benign code, and this work studies only deliberately malicious software. While deliberately-introduced vulnerabilities represent a real attack vector, distinguishing these from accidental ones is difficult, and we leave characterization of deliberate vulnerabilities to future work. We used the following criteria for selecting tools to benchmark:

1) *Anomaly-based malware detection tools.* While several other tools (e.g., [10], [29]) analyze package metadata such as package name or package downloads, we focus on tools that analyze package code as it is more reliable.
2) *Source code available.* Our benchmarking analysis requires access to the tool's source code. Specifically, the details about the detection technique (e.g., which kind of artifact or a file that a tool processes or ignores) must be available.
3) *Detection rules available.* Some tools, such as OSSF package-analysis [47], only provide raw analytical results. Evaluating those tools requires a researcher to write their own detection rules, a potential source of analytical bias. Therefore, this benchmarking exercise only considered the tools that include detection rules.

Table I contains the Python malware scanners considered for inclusion in this study. The tools range from simple heuristics and static analysis tools to complex dynamic analysis tools. Most, though not all, are available on GitHub. Three tools met all inclusion rules: the PyPI malware checks [14], Bandit4Mal [59], and OSSGadget's OSS Detect Backdoor [58].

The PyPI malware checks were pushed in February 2020 by PyPA, as part of a modular pipeline capable of running many different types of malware analysis [71]. The only deployed malware check, a "setup pattern check," uses regular-expression-based YARA rules [72] to scan `setup.py` files (which run on package installation). This check includes four rules that alert suspicious imports and API calls in `setup.py` files. In our experiments, we used the default PyPI malware check rules in combination with a tool called yara-scanner [73] to scan the packages. In our interviews, repository administrators reported that they do not actively use the alerts generated by the checks in this pipeline.

OSS Detect Backdoor [58] is a tool in a collection of tools called OSSGadget developed by Microsoft in June 2020. The collection provides 12 utilities to analyze different aspects of open-source projects (e.g., health metrics). In our experiment, we used the default OSS Detect Backdoor ruleset, which includes 41 rules checking that check for regular-expression-based malicious patterns (e.g., a reverse shell) of every text file in a package. The tool currently supports scanning live packages from 15 package repositories, local packages, and generic URLs linked to packages.

Bandit4Mal [59] is a custom version of Bandit [74] that includes 45 rules to capture malicious patterns in a package specifically. Because Bandit4Mal relies on AST analysis, it must target a specific Python version. We use the Python 3 version of Bandit4Mal because of its widely [75].

The PyPI malware checks run only on `setup.py` files, on the theory that Python malware commonly runs at install-time and those suspicious behaviors are easier to catch in these files. However, some creators of Python malware inject malicious code into other files executed at runtime [45]. Therefore, in our evaluation (Section IV), we first ran each tool only on `setup.py` files and then, second, on all Python files and recorded our evaluations separately.

B. Benchmark dataset

Our dataset comprises malicious and benign Python packages collected from real-world attacks and PyPI. We used as many malicious PyPI packages as possible and chose roughly 1,400 popular or widely depended upon packages as a benign dataset. We measured true positive/false negative rates against the malicious packages, and true negative/false positive rates against the benign packages. From this, we can extrapolate to real-world ratios of malicious/benign packages.

a) A malicious Python package dataset: Assembling a malicious Python package dataset was straightforward due to recent data collection efforts. The Backstabber's Knife Collection dataset [17], [30] (commit `22bd76`) contains 107 Python packages previously identified as malware. The MalOSS dataset [8], [18] (commit `2349402e`) contains 140 valid examples. Both datasets have been collected manually from and represent real-world attacks. The MalOSS dataset also includes malicious packages detected by the authors while scanning PyPI and other package repositories [8].

The benchmark dataset includes only one instance if a package was present in both datasets. We also removed one duplicate of a similarly named package with the same contents. In addition, the malicious dataset omitted three packages that do not contain any Python files. Only the latest version of each package was included. In total, the dataset contains 168 malicious Python packages.

b) Benign Python package datasets: Ideally, this benchmark dataset would include a set of Python packages that is universally viewed as malware-free. Unfortunately, there is no such existing dataset in the literature. This work, therefore, proposes two different second-best approaches.

Popular PyPI packages: Following Zahan et al. [7], we created a combined dataset of the 1,000 most downloaded [76] and 1,000 most widely depended upon Python packages. At the time of publication (6 months after the collection of the dataset), none of these packages have, to our knowledge, been found to be malicious. We excluded three packages containing no Python files. After deduplicating packages found in both datasets, the benign dataset contained 1,430 packages.

Popular packages may be different from a typical Python package: potentially better engineered and more conformant to standard Python programming practices. Consequently, using only popular packages as the benign dataset might lead to unrealistic benchmark results since these packages might be relatively easy for detection tools to classify as benign [61].

Random PyPI packages: The second step was to select the most recent version of 1,000 randomly chosen Python packages. We excluded the packages that lacked any Python files and any packages that were no longer available at the time of analysis. This led to a benign dataset of 986 packages. While there is a chance that some of these packages are malicious, the chance that more than a handful of these packages are malicious is small. Conservatively assuming 2,000 undetected malicious packages on PyPI (of 400,000 total), the probability that more than 10 packages in this set are malicious is less than 2 percent; this bounds the potential error in our false positive and true negative rates at around 1 %.

While we were unable to audit the 45 million lines of code in the popular packages dataset in Table II, we did perform spot checks on a small sample of benign packages flagged as malware. For instance, the ostensibly benign popular packages `pymupdf` and `configargparse` were flagged as malicious by one of the scanners. None of the spot checks revealed malicious code, although they contained suspicious patterns.

TABLE II: Descriptive Statistics of the Benchmark Dataset
Line counts from `scc` [77]. We use only the latest release of each package.

Dataset	Packages	Python Files	Lines of Code
Malicious	168	1,339	228,192
Benign (popular)	1,430	164,223	45,254,876
Benign (random)	986	16,832	2,770,978

Table II details the number of packages in each benchmark sub-dataset. Note that for each package, we obtain the latest release. The number of Python files and the number of lines of code of the popular packages are significantly larger than the random and malicious packages because popular packages tend to provide much more functionality.

C. Benchmarking Evaluation

Table III reports the number of packages with one or more alerts for all datasets using all approaches described in Section IV. In this section, we report the results of the analysis of the alerts generated by the malware detection tools.

a) Scanners catch more than half of malicious packages: All checks, when run on only `setup.py` files, fired alerts (true positive) for over 50 % of packages, with OSS detect backdoor the lowest (50.6 %) and Bandit4Mal the highest (66.7 %). When including all Python files, the checks detected over 85 % of malicious packages. All approaches, no matter the test corpus, had true positive rates above 50 %.

b) False positive rates are high (sometimes higher than true positive rates): Using a single alert as our threshold, the lowest false positive rates were found checking only `setup.py` files. However, we found false positive rates of at least 14.9 % (PyPI malware checks, random packages) and as high as 79.5 % (Bandit4Mal, popular packages).

The false positive rate increases when checking all files (since the alerts that fire are a superset of those for the `setup.py` alerts). The positivity rate was highest for the popular packages, which tended to have more files and more lines of code than the malicious or random packages. Bandit4Mal and OSS Detect Backdoor had the greatest false positive rates (96.6 %) when scanning all Python files. In fact, the false positive rate for popular packages with these tools was *higher* than the true positive rate for malicious packages.

c) Positive packages had many alerts: These checks may fire multiple alerts per package, and in many cases they did. A user of these checks might have to investigate all such alerts to render a verdict on a suspicious package.

Scanning only `setup.py` files, we find that all methods have a median of 3 or fewer alerts among the benign packages which triggered *any* alerts. However, the maximum number of alerts among benign packages (which triggered at least one alert) was high: 45, 77, and 75 for the PyPI checks, OSS Detect

TABLE III: Packages with at least one alert, by tool (%).

	PyPI checks		OSS Detect Backdoor		Bandit4Mal	
Dataset	`setup.py` (%)	`*.py` (%)	`setup.py` (%)	`*.py` (%)	`setup.py` (%)	`*.py` (%)
Malicious	58.9	85.7	50.6	85.1	66.7	90.5
Benign (popular)	33.2	94.2	41.7	96.6	79.5	96.6
Benign (random)	14.9	67.8	33.1	77.2	70.2	84.6

Backdoor, and Bandit4Mal, respectively. For malicious packages, the numbers were not necessarily higher: the maximum number of alerts was 3 for the PyPI malware checks.

When scanning all Python files, the number of alerts increases dramatically. The median number of alerts (for popular benign packages packages with at least one alert) was 10, 85, and 19 for the PyPI checks, OSS Detect Backdoor, and Bandit4Mal, respectively. The PyPI checks fired 5,377 alerts for one popular package, and Bandit4Mal fired 16,155 alerts for one as well. But the noisiest was OSS Detect Backdoor, which fired 124,267 for one benign (popular) package.

d) Increasing alerting thresholds drops true positive rates while false positive rates remain nontrivial: The above results may have over-estimated false positive rates, since we used the lowest possible threshold of one alert. It is possible that some other threshold could lead to superior performance results of the tools in terms of balancing the tradeoff between the number of true positives and false positives.

Figure 2 uses a variety of alert thresholds to plot the true positive rate for the malicious packages and the false positive rates for both benign package datasets. This figure contains two sub-graphs per detection tool, one for the rates when scanning only `setup.py` files and another when all Python files are scanned. In general, Figure 2 suggests that a low threshold of alerts generated many false positives, especially when using Bandit4Mal and OSS Detect Backdoor on all Python files. On the other hand, setting a threshold higher than 4 alerts makes the tools report fewer malicious packages, or even none in the case of PyPI malware checks on `setup.py` files. Across all the tools, we observed that 3 alerts seems to be an optimal threshold for the tools to balance the number of malicious packages identified and false positives.

e) Some rules are more effective than others: To understand why these tools flagged so many benign packages as malicious, we broke down the specific rules that were triggered in the case of the PyPI malware checks. Figure 3 shows the distribution of the alerts for each rule in the `setup.py` files of the three datasets. We observed that `metaprogramming_in_setup` is the most common rule triggered in the popular and random packages. However, malicious packages contain the highest percentage of `networking_in_setup` alerts. This indicates the indicators of a networking event could provide a higher confidence of maliciousness. We found a similar frequency of the `process_spawn_in_setup` and `subprocess_in_setup` rules in all the datasets, which suggests that using this rule is not so effective in distinguishing malicious and benign code.

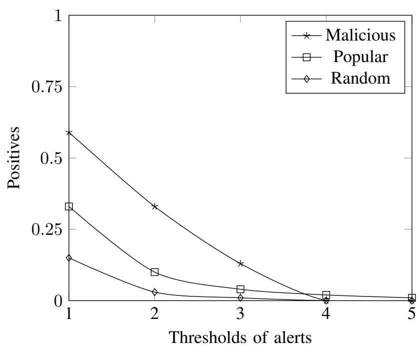
f) The tools ran in a reasonable amount of time, especially for `setup.py` files: Tested on a laptop with 4 CPU cores and 8 GB RAM, the PyPI checks and OSS Detect Backdoor ran with a median time around 2 seconds per package for all datasets and a maximum run time of 26 minutes for one popular package (the package `ansible`). Also, we observed that OSS Detect Backdoor ran slightly slower than PyPI malware checks (0.8 seconds, on average). Bandit4Mal took less than a second to process a malicious or random package, on average. However, the tool runs with a median time of nearly 5 seconds for the popular package dataset and a maximum run time of 2.5 hours for one popular package.

V. Discussion

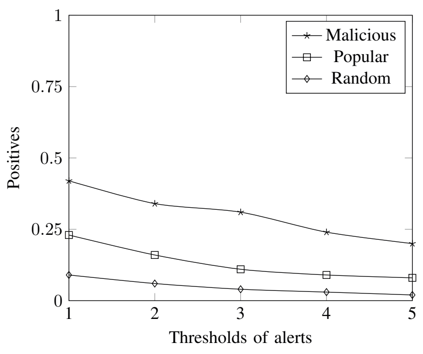
In our qualitative research (Section III), we find that the motivation and even the setting of malware detection for package repositories differ in practice from those assumed in the literature. In particular, removing all malware from PyPI is a low priority for administrators relative to other past, planned, and ongoing security interventions, as the interviewees report that many malicious packages are downloaded very rarely, and consequently have minimal impact on real developers. For instance, in a representative sample of 29 malware packages [66], researchers report that packages were downloaded under 200 times on average over the course of 20 days; we independently confirmed similar download numbers for other reported packages [78]. The PyPI administrators suggest that most or all of these downloads are likely from the many automated mirrors which download every package on PyPI. Based on this, we formulate requirements for an automated malware scanner deployed in this setting. In Section IV, we investigate existing tools in the context of these requirements, evaluating both the performance (with a particular focus on false positive rates across tunable thresholds) and ease of deployment. We find that the tools we evaluate do not meet these requirements. However, these tools may still be useful in the broader repository malware detection ecosystem.

A. Requirements for repository-side malware checks

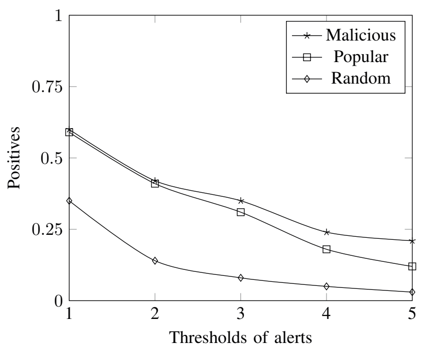
There are two types of repository-side malware checks: *blocking*, in which a published package is not accepted until after a scan, or *passive*, in which releases are scanned after-the-fact. Both require ongoing intervention by repository administrators, who are often volunteers or employees of a nonprofit and under-resourced. For any blocking checks, there must be an appeal process. For passive checks, positive results



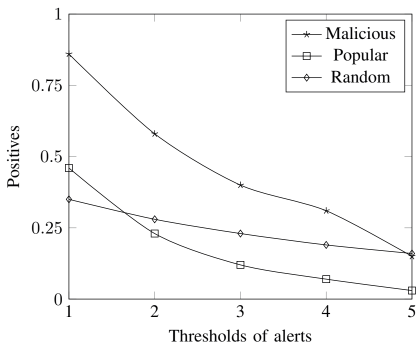
(a) PyPI checks, `setup.py` only



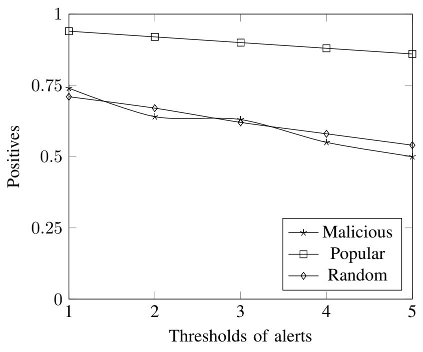
(b) OSS Detect Backdoor, `setup.py` only



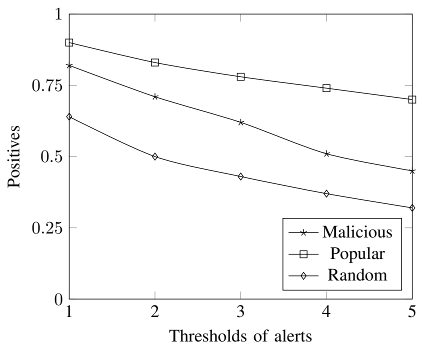
(c) Bandit4Mal, `setup.py` only



(d) PyPI checks, `*.py`



(e) OSS Detect Backdoor, `*.py`



(f) Bandit4Mal, `*.py`

Fig. 2: Percentage of packages with at least the given threshold of alerts.

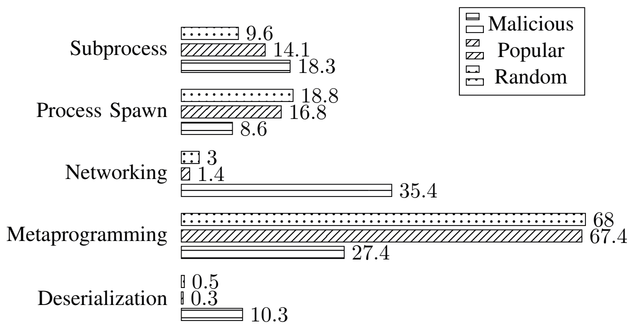


Fig. 3: Distribution of the rules in the three datasets
Note: This analysis analyzed only `setup.py` files for each package. The x-axis is on log-scale.

trigger alerts to administrators; after reviewing alerts, the administrators remove any packages they deem malicious. The following requirements apply to both settings.

"Zero" false-positives. Malicious packages are relatively rare (under 3 % of all packages in 2022 [28]); consequently, even with very low false positive rates and a perfect true positive rate (an unrealistic goal with current methods), the majority of alerts for an automated system will be false positives. Administrators are unwilling to spend more than a minute or so per alert (ideally even less), or review alerts in real-time (for instance, in the middle of the night). The ideal tool would have a "zero" false-positive rate: at most 0.001 % for a passive check, and even lower for a blocking one.

Low effort to adopt and maintain. Engineering and maintenance efforts limit the checks that can be deployed: static checks, which do not require running potentially malicious code, are feasible, while dynamic checks, which must run in a carefully-built sandbox, are not. Similarly, interviewees expressed skepticism about "research-grade" software, which they view as insufficiently robust for a load-bearing role in their repository.

Low-effort to use. Administrators want to make very few value judgments and prefer their interaction with a report of malware to involve only a quick confirmation that the code looks malicious. Malware checks should have a binary decision (possibly based on a tunable threshold) and present a guess about whether a package is malicious, not just a list of suspicious features. However, checks should expose evidence of malicious behavior, as this allows administrators to confirm before taking the package down. Fortunately, this setting allows trade-offs.

False negatives are acceptable. Repository administrators believe that no automated system can detect all or even most attacks. However, such systems are not useless: PyPI sees a steady, large volume of low-effort attacks, and even a system with limited sensitivity would help to clean these up.

Computing resources are generous. Any blocking checks must run in at most seconds, since users expect packages to be available immediately after upload. However, passive checks can take much longer. Further, PyPI has access to a large amount of cloud computing resources, and could deploy even computationally expensive passive checks.

B. Current techniques fall short

The evaluated malware detection tools had several traits which immediately disqualify them for inclusion in a repository malware analysis pipeline. These tools are flexible by supporting user-supplied suspicious patterns and allowing users to make their own decision rules. However, this flexibility is a fault as it increases the burden of maintenance and decision making for repository administrators.

They emitted alerts for between 15 % and 97 % of benign packages. The distinction between malicious and benign code is subtle. Opening a reverse shell might be useful in a debugging setting, but unwanted on a production machine. Techniques associated with malware, like obfuscation, are also used in web programming to save bandwidth and to slow down reverse engineering of proprietary code. In general, whether code is malicious depends on whether the behavior is wanted, which in turn depends on who the downstream users are. While it is not clear whether improved techniques could dramatically lower false positive rates, it is unsurprising that current techniques do have such high rates of false positives.

These tools did have some advantages. They were all static checks and reasonably easy to run in an automated pipeline. They ran in times appropriate for passive checks, though not blocking checks. Overall, such tools are unsuitable for deployment as repository-side malware scanners. However, they may still be useful elsewhere.

C. Alternative: a socio-technical malware detection system

There *are* malware checks for PyPI, but they involve a symbiotic socio-technical system comprising people and organizations, not just computers. For-profit companies and research institutions gain reputation and prestige by finding malware; they report this to the administrators in exchange for the opportunity to publicize their work. Administrators outsource malware detection to researchers who are willing to troll through numerous false alerts in order to evaluate their methods or claim credit for discovery. This system works well enough to allow administrators to focus on other security work.

In such a system, the onus falls on researchers to find malware. These researchers develop and evaluate new tools. They also manually deal with false positives. Such a researcher may find a 15 % false positive rate acceptable, at least in the short term, and they can use any number of techniques in their toolkit for malware hunting.

In some ways, this "solution" mirrors broader trends in open-source software related to the involvement of companies in open-source software and the ability of disparate parties to coordinate and divide tasks via online communication [79], [80]. Nonetheless, there is likely still ample room, according to all interviewees, to reduce transaction costs and improve collaboration between PyPI, other OSS registries, enterprises, and individuals scanning these repositories for malware.

VI. Recommendations

This section provides recommendations for academics and security researchers to improve the quality and usability of OSS malware detection tools, especially for PyPI.

Most effort should be elsewhere. Our interviewees reported long lists of proposed security improvements and limited ability to implement these. Some are a poor fit for researchers: they require money, time, or large-scale (but not particularly novel) engineering projects. But others require skills that security researchers have: threat modeling, systems design, and analysis, or cryptographic implementation.

Researchers should design for malware detection as practiced. The above requirements rule out even the very best academic tools. Efforts to meet these requirements are likely futile. However, giving up on this ambition frees researchers to experiment with the tools that work for them: they no longer need to write purely non-interactive scanners or even aspire to low false-positive rates. Designing tools for researchers removes constraints and allows experimental methods.

Improve the socio-technical system. The ecosystem which has organically emerged to deal with the threat of malware on PyPI involves a number of organizations and individuals, each with their own goals. Perhaps by accident, the goals of these individuals are mostly aligned; a system deliberately designed, for instance, in collaboration with researchers could have even better outcomes. However, administrators were wary of perverse incentives: monetary rewards for reporting malware may inspire unscrupulous researchers to plant malware for themselves to "find."

Consider the goals of repository administrators. Removing all malware is a low priority for administrators. They worry much more about critical packages [36] or packages that users may accidentally install via a typo. While suspicious code by itself is too weak of a signal on which to remove a package, suspicious code in a suspiciously-named package might be. Combined metadata and anomaly-based indicators may be a more useful signal than either on their own. Similarly, administrators prioritize low false positive rates over catching all or most attacks; this may indicate that signature-based malware detection is, in fact, appropriate, despite the trend in the academic literature over the past decades to the contrary.

Improve usability of OSS malware detection tools. Existing systems emit long lists of "alerts," which a human user must investigate and interpret. Instead, the output should be actionable: a concrete prediction that a package warrants further investigation. Additionally, further investigation should be as easy and useful as possible. Research on human-computer interaction may help create tools that give useful signals while still allowing researchers or administrators to quickly drill down and root out malware.

Open science principles can aid OSS malware detection. We omitted several tools because their detection rules were missing or their source was unavailable. Researchers should strive for reproducibility, shared datasets, and consistent interfaces which make cross-tool comparison easy. We make a minor contribution in this area by publishing metadata that allows easily reconstructing our dataset.

Funders should embrace this ecosystem. While administrators maintain relationships with researchers and act on their guidance, they have a number of shovel-ready ideas [81] to make this system more efficient. For instance, PyPI does retain all packages removed as malware, but the metadata is unavailable. Making such a dataset available to identified researchers would enable training and honing much better research tools. Similarly, repositories should facilitate scanning via a "firehose"-like stream of updates or bulk datasets of all packages (for instance, torrents). Companies who have built businesses on open-source software, nonprofits, and governments should contribute engineering resources or funding to enable implementation.

VII. Threats to Validity

The interview sample could be affected by selection bias or survivor bias. While the research team intentionally interviewed those parties that were most closely involved with the PyPI malware detection system and related efforts, the research did not interview all possible parties. There could therefore be selection bias in the interview sample. Additionally, it also possible that some relevant parties are no longer involved with PyPI and were thus not discovered during the team's background research, and therefore the current interview sample could be affected by survivor bias.

The malicious dataset may not accurately represent malicious packages in the wild. Not all malicious PyPI packages are publicly known. Backstabber's Knife Collection and MalOSS samples are the largest OSS supply-chain malware repositories available for researchers upon request. However, our interviewees report removing about 100 malicious packages per month, suggesting that these datasets capture only a small portion of total PyPI malware. The 168 packages in the analyzed dataset likely exhibit some sampling bias, since they were those that researchers were able to track down; a randomly-sampled dataset of malware might contain samples with different characteristics.

The benchmarking analysis excluded a number of OSS malware detection tools. Several other OSS malware detection tools exist but were excluded from the study either because the tool had unavailable source code, did not use anomaly-based analysis, or lacked published detection rules. Also, there are OSS malware detection tools designed for other programming language ecosystems (e.g., [54] for npm) that could potentially be used, after engineering modifications, to analyze PyPI packages. Future evaluations could therefore benchmark more and different OSS malware detection tools.

More sophisticated decision rules might improve performance. This analysis used decision rules related to only the number of alerts for each package, finding that false positive rates were unacceptably high even when using a threshold that eliminated most true positives. More sophisticated rules might use signals like the type of alert to deliver better performance.

This evaluation only analyzed the latest versions of the packages. Most attackers target the latest versions of a package, which maximizes the number of victims, so the benchmark analysis only examined the latest version of each package. Some attackers, however, inject malicious code into older versions of a package, possibly hoping that the developers who use older versions are less cautious about security [8].

These results may not generalize to other package repositories. This work presents a case study of the Python ecosystem. We suspect that many of the findings generalize: we are unfamiliar with malware detection tools in any programming languages that reach the low false positive rates our work requires. Nonetheless, it is possible that other repositories may have different levels of staffing or different security priorities. Future work may compare both malware detection tooling and software repository operations across ecosystems.

VIII. Conclusion and Future Work

This paper's benchmarking of approaches to Python malware detection in a repository setting resulted in a clear finding: high false positive rates uniformly characterize these approaches, which imposes a high burden on any single organization that decides to police PyPI from malware.

Fortunately, our interviews revealed a socio-technical system involving both repository administrators and security researchers jointly engaged in PyPI malware detection. This coalition enables security researchers, with a higher tolerance for false positives than repository administrators, to specialize in the malware detection component of the system. This arrangement is a feature, not a bug. Parties interested in strengthening repository defenses against malware, at PyPI and potentially other OSS repositories too, should seek to strengthen this system and reduce the coordination and transaction costs between the different parties.

Our code and replication data are available on GitHub at https://github.com/lyvd/bad-snakes-icse23-artifacts.

References

[1] A. Dewdney, "Computer recreations: In the game called Core War hostile programs engage in a battle of bits," *Scientific American*, vol. 250, no. 5, pp. 14–23, May 1984.

[2] M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, *Malware detection*. Springer Science & Business Media, 2007, vol. 27.

[3] A. Mohanta and A. Saldanha, *Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware*. Springer, 2020.

[4] I. Xplore, "International conference on malicious and unwanted software (malware)," https://ieeexplore.ieee.org/xpl/conhome/1002525/all-proceedings, 2008.

[5] 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)*, 2019, pp. 1393–1410.

[6] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[7] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. 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)*. IEEE, 2022, pp. 331–340.

[8] 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 *Proc. of NDSS’21*, 2021.

[9] R. K. Vaidya, L. De Carli, D. Davidson, and V. Rastogi, “Security issues in language-based sofware ecosystems,” *arXiv preprint arXiv:1903.02613*, 2019.

[10] M. Taylor, R. Vaidya, D. Davidson, L. D. Carli, and V. Rastogi, “Defending against package typosquatting,” in *International Conference on Network and System Security*. Springer, 2020, pp. 112–131.

[11] E. Wyss, L. De Carli, and D. Davidson, “What the fork? finding hidden code clones in npm,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 2415–2426.

[12] M. Borins, “Top-100 npm package maintainers now require 2fa, and additional security-focused improvements to npm,” https://github.blog/2022-02-01-top-100-npm-package-maintainers-require-2fa-additional-security/, 2022.

[13] D. Ingram and J. Chester, “Your favorite software repositories, now working together,” https://openssf.org/blog/2022/04/19/your-favorite-software-repositories-now-working-together/, 2022.

[14] Warehouse, “Malware checks,” https://warehouse.readthedocs.io/development/malware-checks/#malware-checks, 2020.

[15] P. S. Foundation, “Reporting a security issue,” https://pypi.org/security/, 2022.

[16] ACM, “Acm publications policy on research involving human participants and subjects,” https://www.acm.org/publications/policies/research-involving-human-participants-and-subjects, 2021.

[17] M. Ohm, “Backstabbers knife collection homepage,” https://dasfreak.github.io/Backstabbers-Knife-Collection/, 2020.

[18] R. Duan, “Maloss homepage,” https://github.com/osssanitizer/maloss, 2020.

[19] T. Herr, J. Lee, W. Loomis, and S. Scott, “Breaking trust: Shades of crisis across an insecure software supply chain,” https://www.atlanticcouncil.org/in-depth-research-reports/report/breaking-trust-shades-of-crisis-across-an-insecure-software-supply-chain/, 2020, accessed: 2020-07-30.

[20] D. Geer, B. Tozer, and J. S. Meyers, “Counting broken links: A quant’s view of software supply chain security,” in *USENIX; Login:, Vol. 45, no. 4*, 2020.

[21] C. M. U. Software Engineering Institute, “1999 cert advisories,” https://resources.sei.cmu.edu/asset_files/WhitePaper/1999_019_001_496184.pdf, 1999.

[22] I. Labs, “Software supply chain compromises - a living dataset,” https://github.com/IQTLabs/software-supply-chain-compromises, 2020.

[23] GAO, “Federal response to solarwinds and microsoft exchange incidents,” https://www.gao.gov/products/gao-22-104746, 2022.

[24] I. Arvanitis, G. Ntousakis, S. Ioannidis, and N. Vasilakis, “A systematic analysis of the event-stream incident,” in *Proceedings of the 15th European Workshop on Systems Security*, 2022, pp. 22–28.

[25] Python Software Foundation, “Python Package Index,” https://pypi.org/.

[26] PyPA, “Warehouse codebase,” https://warehouse.pypa.io/application.html, 2018.

[27] D. Haney, “Npm & left-pad: Have we forgotten how to program?” https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/, 2017.

[28] D. Ingram, “Twitter: in 2022, the pypi team removed >12,000 unique projects. each were instances of spam, typosquatting, dependency confusion, exfiltration and/or malware.” https://twitter.com/di_codes/status/1610781657128108033, 2023.

[29] 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.

[30] M. Ohm, H. Plate, A. Sykosch, and M. Meier, “Backstabber’s knife collection: A review of open source software supply chain attacks,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2020, pp. 23–43.

[31] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, “A look in the mirror: Attacks on package managers,” in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 565–574.

[32] P. Ladisa, H. Plate, M. Martinez, and O. Barais, “Taxonomy of attacks on open-source software supply chains,” *arXiv preprint arXiv:2204.04008*, 2022.

[33] Lutoma, “Psa: There is a fake version of this package on pypi with malicious code,” https://github.com/dateutil/dateutil/issues/984, 2019, accessed 6 February 2020.

[34] L. H. Newman, “The fragile open source ecosystem isn’t ready for ‘protestware’,” https://www.wired.com/story/open-source-sabotage-protestware/, 2022.

[35] R. G. Kula and C. Treude, “In war and peace: The impact of world politics on so ware ecosystems,” *arXiv preprint arXiv:2208.01393*, 2022.

[36] “Pypi 2fa security key giveaway,” https://pypi.org/security-key-giveaway/, 2022.

[37] “Multi-factor authentication rollout,” https://github.com/rubygems/rfcs/blob/master/text/0007-mfa-rollout.md, 2022.

[38] T. L. Foundation, “A new standard for signing, verifying and protecting software,” https://www.sigstore.dev/, 2021.

[39] 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*, 2022, pp. 2353–2367.

[40] T. R. Philip Harrison, Fredrik Skogman, “Link npm packages to the originating source code repository and build,” https://github.com/npm/rfcs/blob/e000b367d9e595bc694893c3d845df269f9b875f/accepted/0049-link-packages-to-source-and-build.md, 2022.

[41] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, “Survivable key compromise in software update systems,” in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 61–72.

[42] K. et al., “Pep 458 – secure pypi downloads with signed repository metadata,” https://peps.python.org/pep-0458/, 2019.

[43] ——, “Pep 480 – surviving a compromise of pypi: End-to-end signing of packages,” https://peps.python.org/pep-0480/, 2014.

[44] Debian, “Reproducible builds,” https://reproducible-builds.org/, 2019, accessed: 2020-08-17.

[45] 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*, 2021, pp. 780–792.

[46] 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)*. IEEE, 2020, pp. 677–681.

[47] OpenSSF, “Open source package analysis,” https://github.com/ossf/package-analysis, 2020.

[48] Microsoft, “Collection of tools for analyzing open source packages.” https://github.com/microsoft/OSSGadget, 2019.

[49] N. Idika and A. P. Mathur, “A survey of malware detection techniques,” *Purdue University*, vol. 48, no. 2, pp. 32–46, 2007.

[50] Ö. A. Aslan and R. Samet, “A comprehensive review on malware detection approaches,” *IEEE Access*, vol. 8, pp. 6249–6271, 2020.

[51] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, “A survey on malware detection using data mining techniques,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, pp. 1–40, 2017.

[52] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, “A survey on heuristic malware detection techniques,” in *The 5th Conference on Information and Knowledge Technology*. IEEE, 2013, pp. 113–120.

[53] A. Sejfia and M. Schäfer, “Practical automated detection of malicious npm packages,” *arXiv preprint arXiv:2202.13953*, 2022.

[54] M. Ohm, F. Boes, C. Bungartz, and M. Meier, “On the feasibility of supervised machine learning for the detection of malicious software packages,” in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, 2022, pp. 1–10.

[55] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, “Towards using source code repositories to identify software supply chain attacks,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 2093–2095.

[56] D. Gonzalez, T. Zimmermann, P. Godefroid, and M. Schäfer, “Anomalicious: Automated detection of anomalous and potentially malicious commits on github,” in *2021 IEEE/ACM 43rd International Conference*

on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2021, pp. 258–267.

[57] A. Cao and B. Dolan-Gavitt, "What the fork? finding and analyzing malware in github forks," in *Proc. of NDSS'22*, 2022.

[58] Microsoft, "Oss detect backdoor," https://github.com/microsoft/OSSGadget/wiki/OSS-Detect-Backdoor, 2019.

[59] D.-L. Vu, "A fork of bandit tool with patterns to identifying malicious python code." https://github.com/lyvd/bandit4mal, 2020.

[60] Bertus, "Detecting cyber attacks in the python package index (pypi)," https://medium.com/@bertusk/detecting-cyber-attacks-in-the-python-package-index-pypi-61ab2b585c67, 2018, accessed 18 January 2020.

[61] J. Wright, "Hunting for malicious packages on pypi," https://jordan-wright.com/blog/post/2020-11-12-hunting-for-malicious-packages-on-pypi/, 2020.

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

[63] Sonatype, "Sonatype delivers first of its kind, automated malware prevention for open source libraries," https://www.sonatype.com/press-releases/next-generation-nexus-intelligence, 2019.

[64] A. Ko, *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds. O'Reilly, 2010.

[65] P. S. Foundation, "Python package index dataset," https://console.cloud.google.com/bigquery?p=bigquery-public-data&d=pypi&page=dataset&pli=1&project=moonlit-web-219515, 2021.

[66] Phylum Research Team, "Phylum discovers dozens more pypi packages attempting to deliver w4sp stealer in ongoing supply-chain attack," https://blog.phylum.io/phylum-discovers-dozens-more-pypi-packages-attempting-to-deliver-w4sp-stealer-in-ongoing-supply-chain-attack, 2022.

[67] J. Waldo and K. Mansted, "Ending the cybersecurity arms race," https://www.belfercenter.org/publication/ending-cybersecurity-arms-race, 2018.

[68] M. Carnogursky, "Attacks on package managers," https://is.muni.cz/th/y41ft/, 2019.

[69] JFROG, "Jfrog xray," https://jfrog.com/xray/, 2022.

[70] IQTLabs, "pypi-scan: A tool for scanning the python package index for typosquatters," https://www.iqt.org/pypi-scan/, 2020.

[71] PyPI, "Tooling for automated detection of malware," https://github.com/pypi/warehouse/pull/7377, 2020.

[72] VirusTotal, "Writing yara rules," https://yara.readthedocs.io/en/stable/writingrules.html, 2021.

[73] A. Ecosystem, "Yara scanner," https://github.com/ace-ecosystem/yara_scanner, 2019.

[74] PyCQA, "Bandit is a tool designed to find common security issues in python code." https://bandit.readthedocs.io/en/latest/, 2019.

[75] JetBrains, "Python developers survey 2021 results," https://lp.jetbrains.com/python-developers-survey-2021/, 2021.

[76] H. van Kemenade, "Top pypi packages," https://hugovk.github.io/top-pypi-packages/.

[77] B. Boyter, "Sloc cloc and code (scc)," https://github.com/boyter/scc, 2019.

[78] Sonatype, "State of the software supply chain 2021," https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021, 2021.

[79] D. Geer and G. P. Sieniawski, "Who will pay the piper for open source software maintenance? can we increase reliability as we increase reliance?" *login Usenix Mag.*, vol. 45, no. 2, 2020.

[80] C. M. Schweik and R. C. English, *Internet success: a study of open-source software commons*. MIT Press, 2012.

[81] P. S. Foundation, "Productionize malware detection," https://github.com/psf/fundable-packaging-improvements/blob/master/FUNDABLES.md#productionize-malware-detection, 2022.