# On the feasibility of detecting injections in malicious npm packages

Simone Scalco
University of Trento
Trento, Italy
simone.scalco@studenti.unitn.it

Duc-Ly Vu
FPT University
Ho Chi Minh City, Vietnam
lyvd@fe.edu.vn

Ranindya Paramitha
University of Trento
Trento, Italy
ranindya.paramitha@unitn.it

Fabio Massacci
University of Trento
Trento, Italy
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
fabio.massacci@ieee.org

## ABSTRACT

Open-source packages typically have their source code available on a source code repository (e.g., on GitHub), but developers prefer to use pre-built artifacts directly from the package repositories (such as npm for JavaScript). Between the source code and the distributed artifacts, there could be differences that pose security risks (e.g., attackers deploy malicious code during package installation) in the software supply chain. Existing package scanners focus on the entire artifact of a package to detect this kind of attacks. These procedures are not only time consuming, but also generate high irrelevant alerts (FPs). An approach called LastPyMile by Vu et al. (ESEC/FSE'21) has been shown to be effective in detecting discrepancies and reducing false alerts in vetting Python packages on PyPI by focusing only on the differences between the source and the package. In this work, we propose to port that approach to scan JavaScript packages in the npm ecosystem. We presented a preliminary evaluation of our implementation on a set of real malicious npm packages and the top popular packages. The results show that while being 20.7x faster than GIT-LOG approach, our approach managed to reduce the percentage of false alerts produced by package scanner by 69%.

## CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems;**; • **Security and privacy → Software security engineering;**.

## KEYWORDS

Open source software, software supply chain, JavaScript, npm

**ACM Reference Format:**
Simone Scalco, Duc-Ly Vu, Ranindya Paramitha, and Fabio Massacci. 2022. On the feasibility of detecting injections in malicious npm packages. In *The 17th International Conference on Availability, Reliability and Security (ARES 2022), August 23–26, 2022, Vienna, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3538969.3543815

## 1 INTRODUCTION

Nowadays, FOSS (Free and Open Source Software) has become a fundamental part of the software supply chain [17]. FOSS enables users and developers to audit, review, and even modify the source code to integrate additional features. In software development process, developers tend to use third-party dependencies to speed up their development. In practice, Pashchenko et al. [21] reported that developers rely on certain quality and popularity factors (e.g., number of stars in the software repository, number of contributors, etc.) to decide whether to use the project as a dependency or not. The security aspect, however, is only considered afterwards and enforced depending on company policies.

To assess the quality and security of a software dependency, developers could manually review its source code in a source code repository (e.g. on Github [8]). Once the code is checked, developers can build or compile the source into a built artifact that users can download and install directly. However, the process of manually compiling the source code usually requires knowledge of the build systems. When it comes to big projects, it is time-consuming because such project sometimes depends on many other dependencies. For the practicality, developers prefer to use pre-built packages from a package repository, such as npm [14] for Javascript.

In theory, such practice assumes that no modifications have been introduced in the *last mile* between the source code and the package. However, recent software supply chain attacks have successfully exploited this assumption by injecting malicious code into the downstream projects. Given a high imbalance between the number of uploaded packages and maintainers in the existing ecosystem [23], the popular package repository like npm has become a highly-targeted malware distribution channel for attackers. Therefore, an effective and efficient reviewing process in the supply chain is needed to check for malicious code injections.

The source code of a project is often altered in the build process, and attackers could intentionally insert many modifications (legitimate or malicious) at any point of the chain. Those discrepancies can be introduced by manual or automated build tools [20] or by attackers (e.g., package hijacking attacks). Package end-users have limited ways of finding and correcting these defects to avoid exploitation. One approach for identifying malicious npm packages is by scanning the entire package to find the malicious scripts. This approach has been proven to cause many false positives [20, 22]. A key observation is that only a small part of the source code is modified in code injection attacks. The study by Vu et al. [20] has shown that it is efficient and effective to check only the discrepancies between source code and package for identifying malicious code injection.

However, the approach by Vu et al. [20] is specific for Python packages on PyPI. There are certain challenges for adopting the approach to packages in other ecosystems such as npm. In this work, we aim to replicate [20] to detect injections in npm packages using discrepancies between source and package for npm artifacts. We then aim to answer the following question:

**RQ:** *Can we effectively and efficiently identify the code injected into malicious npm artifacts?*

In order to answer the question, we first aim at replicating [20] by implementing a new tool called LastJSMile, in order to detect code injections in npm packages. To test the approach, we then use a dataset containing malicious npm packages. Moreover, to show the efficiency of our approach we compare our solution with the alternative solution called GIT-LOG mentioned in [20]. GIT-LOG is the approach typically used by developers to trace the presence of a particular line of code in the source code repository. Finally, we integrate the malware rules used in the existing package scanner OSSGadget Detect Backdoor and use the combined approach to test the malicious packages in the *Backstabber* dataset and the popular packages in the npm ecosystem. The alerts generated by our experiments are manually validated to confirm the effectiveness of the proposed approach on both correctly identifying malicious code and reducing the number of false alerts.

The rest of the paper is organized as the following: the motivating example of a malicious artifact is discussed in §2. We then discussed more about software supply chain attacks and the current mitigation techniques in §3. Section §4 discusses in more detail how we implemented LastPyMile for npm. We described the dataset we use for evaluation in §5, and our evaluation results are then discussed in §6. We explained the threat to validity of our preliminary experiment in §7, and lastly we concluded our study and described the possible future works in §8.

## 2 MOTIVATING EXAMPLE

Figure 1 shows an attack on the version 3.7.2 of the popular package `eslint-scope` [1]. In this attack, attackers managed to hijack the npm credential of an `ESLint` maintainer and published the malicious version, which included a brand new module called *build.js* and a modified version of the existing file *package.json*. The malicious code shown in Figure 1 was injected into the npm repository,

```
try{
    var https=require('https');
    https.get({'hostname':'pastebin.com',path:'/raw/XLeVP82h', headers:{'User-Agent':
            'Mozilla/5.0 (Windows NT 6.1; rv:52.0) Gecko/20100101 Firefox/52.0'
            , Accept:'text/html,application/xhtml+xml,application/xml;q
            =0.9,*/*;q=0.8'}},(r)=>{
        r.setEncoding('utf8');
        r.on('data',(c)=>{
            eval(c);
        });
        r.on('error',()=>{});

    }).on('error',()=>{});
}catch(e){}
```

*The red-highlighted texts are the malicious code the attacker injected to steal npm credentials.*

**Figure 1: Malicious code injected in the `lib/build.js` file of *eslint-scope-3.7.2.***

so it persisted until the users of the package noticed it and reported it. As a consequence, more than 4500 users' credentials were stolen [5]. A recent study by Zahan et al. [24] shows that 2818 maintainer accounts associated with an expired domain, allowing an attacker to hijack 8494 packages by taking over the npm accounts.

Existing package security scanners, for example OSS Detect Backdoor would produce many false alerts when applied to scan the whole artifact (version) of a package (see Table 7). Hence, in this case, one should only focus on the two involved files instead of all 12 files of the legitimate artifact. On the other hand, NPM-AUDIT only scans the metadata of a package (e.g., the name of package dependencies), which would fail to detect the malicious code injected. Hence, the challenge for being effective in identifying malicious code is to pinpoint the injected (malicious) code that can be used as an input to existing scanners and humans.

## 3 BACKGROUND

**Npm**. Npm is a package manager that is the largest public repository for JavaScipt packages. At the time of this writing, it hosts more than 2.4 million packages. Developers can publish and install packages from npm using the package installer utility called NPM. Npm packages usually contain a file called *package.json* to facilitate others to manage and install them. Unfortunately, this file is the main target in many software supply chain attacks [16, 24].

**Software supply chain attacks**. Software supply chain attacks are characterized by deliberately injecting malicious code into a software product (e.g., a specific artifact of an npm package) to infect the end-users further down the chain [10]. A single package can be counted as a "supplier" for several other open source software projects which use its code as dependencies. This chaining nature makes popular packages very attractive to attackers. Attackers typically use three attack types in the software supply chain attacks [16]: package typosquatting, package combosquatting, and package hijacking [16, 23]. In typosquatting, attackers could publish malicious modules to the npm registry with names that look like existing popular modules. The intention is to fool users into installing them, either by driving them to do so through targeted actions or just by mistake (a typo). Similarly, in combosquatting, attackers use a combination of a popular package name and another set of characters or a rearrangement of the same name to fool end users. Meanwhile, in package hijacking attacks, bad actors gain

[1] https://www.npmjs.com/package/eslint-scope

**Table 1: Existing tools for analyzing npm packages**

*The current state of the art tools for mitigating software supply chain attacks are limited and produce lots of false alerts*

| Package scanner | Detection Granularity | Technique used |
|---|---|---|
| npm-audit [4] | Metadata of dependencies | Static |
| MalOSS [6] | Package | Static and Dynamic |
| OSSGadget [13] | Artifact | Static |
| Ferreira et al. [7] | Package | Static |
| Ohm et al [15] | Artifact | Static |
| Zahan et al [24] | Metadata | Static |
| Liang et al [11] | Package | Static |

access to developers accounts (e.g. maintainer account for an npm package) and publish compromised versions of a package.

**Detecting malicious npm packages**. Table 1 summarizes the existing approaches used for checking the maliciousness of npm packages. Several approaches analyze the metadata of a package [4, 24]. For example, npm provides a tool called NPM-AUDIT [4] to scan the description of the dependencies in a project against a database of known vulnerabilities. Zahan et al. [24] defined some signals that could indicate malicious package, such as the presence of install scripts. However, this tool does not identify the code of a package that is deliberately injected by attackers in code injection attacks.

Liang [11] used anomaly detection techniques on a set of features (package metadata and code) to flag suspicious packages. However, this approach performs poorly on small-sized malicious artifacts that share most of the code with the legitimate packages. On the other hand, Oss Detect Backdoor [13] relies on a set of regular expressions and performs pattern matching or string matching in order to identify potential malicious lines of code within an artifact. The tool however generates a lot of false alerts, including False Positives (FP) and False Negatives (FN). Several approaches [18, 23] rely on package names to raise alarms on potential suspicious packages. However, further verification (e.g., code-based) is needed to assert the maliciousness of those packages.

MalOSS [6] extracts various features of a package artifact using metadata, static, and dynamic analysis. However, this approach is resource-intensive, which makes it challenging to be integrated into the security pipeline of a very active ecosystem like npm. Ohm et al [15] proposed a set of signatures extracted from known malicious packages. They then identify the clusters of packages that share the same source code using the Markov Cluster Algorithm. This approach however does not generalize to the malicious packages with unknown signatures.

**Detecting code injections in npm packages.** In software supply chain attacks, attackers tend to inject malicious code into existing files of a package artifact to remain hidden from the humans. Vu et al. [20] shows that in code injection attacks, attackers tend to modify only a tiny portion of the legitimate artifact. They then proposed an approach called LastPyMile to detect code injections in PyPI packages by comparing the source and package. However, on that study, they only evaluated the effectiveness of the approach on PyPI packages. Also, they only tested LastPyMile on the typosquatting and combosquatting attacks.

**Summary:** The current approaches for mitigating software supply chain attacks in the npm ecosystem (Table 1) have several limitations, including high number of false positives. We propose a solution, inspired by [20], to effectively identify malicious code in npm packages.

## 4 PROPOSED SOLUTION

We ported the LastPyMile algorithm [20] to detect code discrepancies between JavaScript packages on the npm ecosystem and their source code repositories on Github. Figure 2 shows the four main steps in our implementation:

**Step 1: Finding the source code repository of a npm package.** To identify the differences between the code in package and source code repositories, we first need to locate the source code repository of an npm package (e.g., a Github URL) [20, 22]. Inspired by [19], we query the online npm registry[2] as the main data source. The registry contains the metadata of a package in JSON format and we assume that it is a safe source since it is provided by the npm team itself. At the same time, we retrieve the list of package versions to be processed in Step 3. For each package $\boldsymbol{p}$, we proceed as follows:

- Open the URL https://registry.npmjs.org/$\boldsymbol{p}$, and retrieve the metadata of the package in JSON format.
- Retrieve and parse the URL of the repository by searching the fields "repository/url".
- Retrieve, parse, and sanitize the URL found in the field "homepage".

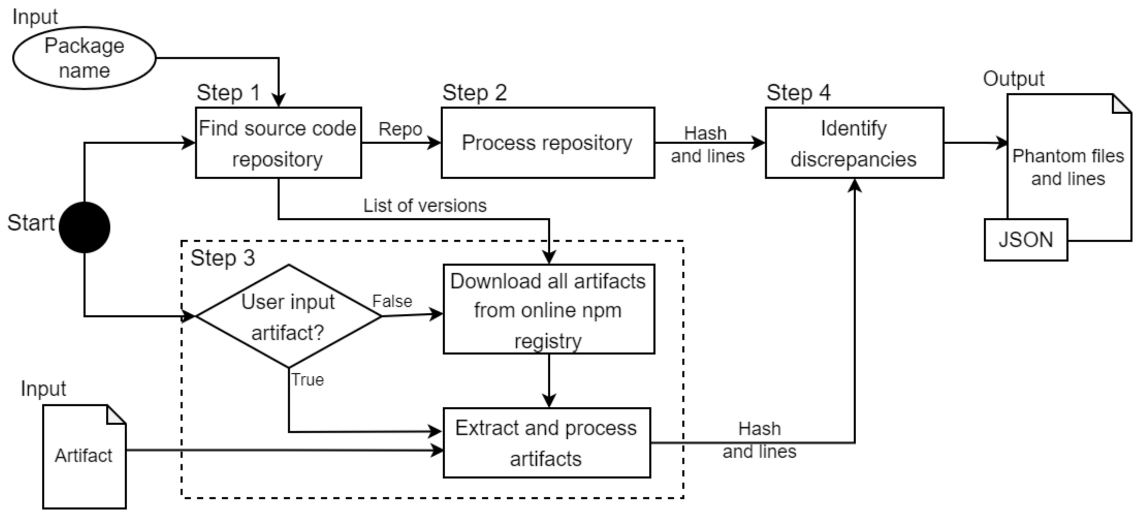
**Step 2: Collecting file hashes and lines from source code repository.** We compute all the hashes (SHA256) and collect the lines of all the files present in all the commits in the history of the source code repository of a package. We use the python library `GitPython`[3] to process all the past commits. To improve the run time overhead, we process all the commits of every branch in the source code repository in parallel using the `multiprocessing` library [4]. The outcome of this step is all the hashes and lines of the files of all commits.

**Step 3: Collecting file hashes and lines of package artifacts.** If the user provides an artifact to our tool as one of the inputs, we proceed to process that particular artifact. Otherwise, our tool will iterate all the versions of a package produced by Step 1 and download all built artifacts. For each package $\boldsymbol{p}$, we proceed as follows:

- Open the URL https://registry.npmjs.org/$\boldsymbol{p}$, and retrieve the JSON containing the metadata of the package.
- Retrieve the latest version of the package by searching the fields "dist-tags/latest".
- Retrieve all the available versions of a package by searching the field "versions".
- Open the URL https://registry.npmjs.org/$\boldsymbol{p}$/$\boldsymbol{v}$ for each version $\boldsymbol{v}$.

We then extract each artifact, compute the hashes (SHA256) of the files, and collect all the lines of the files. In contrast to PyPI packages, most npm packages exist in the tarball (.tgz) format.

[2]https://registry.npmjs.org/
[3]https://github.com/gitpython-developers/GitPython
[4]https://docs.python.org/3/library/multiprocessing.html



*Our implementation consists of four main steps. The inputs are package name and (optional) artifact, while the final output is a JSON file containing phantom files and lines. Source code repository refers specifically to Github repository, and the hashing algorithm used is SHA-256.*

**Figure 2: The workflow of identifying code injections in npm packages**

**Step 4: Identifying phantom files and hashes.** We compare the list of hashes and lines of each version of a package obtained in Step 3 with those in the source code repository obtained in Step 2. Basically, the information retrieved in Step 2 would be useful to check whether there are files in the analyzed artifact that have never been on the repository. The final output is then stored in a JSON file containing the absent files and lines (that are not present in the history of the source code repository) of every processed artifact. Specifically, when a phantom file is found (a file not present in the history of the repository) then the program checks for phantom lines (lines that are not present in the history of the repository). We then generalize the term ***phantom files*** for (1) those absent files *and* (2) the files that contain ***phantom lines*** (absent lines) [20].

## 5 DATASET OF MALICIOUS PACKAGES

To evaluate the effectiveness of our implementation, we used the living malware dataset for open-software supply chain attacks, called *Backstabber* [16]. This dataset contains the malicious artifacts collected from different attacks in major package repositories including PyPI, npm, and RubyGems. We analyze 361 malicious npm artifacts with three types of software supply chain attacks: typosquatting, combosquatting, and package hijacking. We focused on this three types of attacks because they are very relevant in the npm ecosystem.

During our investigation, 226 of the total artifacts (62.6%) were not typosquatting, combosquatting, or package hijacking, therefore we excluded those samples from the analysis. Among the remaining 135 artifacts, we excluded 16 artifacts (11.8%) that used expired repository links. At the end, we analyzed and ran the tool on 119 artifacts as shown in #Artf. column in Table 4. All of these malicious artifacts have been removed by the npm team from the public repository for security reasons.

We focus on the ways attackers inject malicious code into existing artifacts of the packages in the dataset. In particular we see two types of injection:

- modified files: existing files get lines added/deleted by attackers
- brand new files: completely new files are added by attackers

We discuss first (1) the distribution of all injected files generally and then focus on (2) the brand new files.

**Distribution of all injected files.** Table 2 shows the top ten files that are touched by the attackers. The *package.json* file is the most common file because attackers change it to change the package name in typosquatting and combosquatting attacks. We observed that all 119 malicious artifacts contain a modified version of the *package.json* file. This file is commonly used in the npm environment to specify the details about the package itself. From those 119 artifacts, there are 92 cases (77.3%) in which the *package.json* file is used to launch malicious installation scripts as part of the build procedure of the package. On the other hand, in 50 cases, those installation scripts are located in a modified JavaScript file (mostly in the *index.js* file) that is usually available when distributing npm packages.

In 12 cases (10.1%), the malicious code was integrated directly in the *package.json* file. In those instances, the attackers hid malicious bash commands to the post installation features. The commands launched in this file are mostly the ones that delete all the current and parent directories. In typosquatting and combosquatting attacks, the *README* file was used to replicate a popular package description for a malicious artifact. Those are used in conjunction with the *LICENSE* files to make the malicious artifact more similar to the legitimate one. Finally, a small fraction of the packages integrated other additional JavaScript files (e.g. *update.js*, *app.js*, *support.js*) that contain malicious code. In most of those cases, the attacker was able to hijack a legitimate package and submit a new

**Table 2: Top different phantom files in the dataset of malicious npm packages**

*#Package: the number of package with a specific name of phantom files. We observed that attackers rely on package.json file to launch malicious scripts on all artifacts. The percentages are based on the total number of phantom files across packages. Other phantom-file names (the rest 71.7%) occur less than 0.5% each.*

| Phantom-file name | #Package | Percentage (%) |
|---|---|---|
| package.json | 119 | 12.6 |
| index.js | 40 | 4.2 |
| README.md | 35 | 3.7 |
| package-setup.js | 30 | 3.2 |
| LICENSE | 10 | 1.0 |
| install.js | 9 | 0.9 |
| update.js | 7 | 0.7 |
| bower.js | 7 | 0.7 |
| app.js | 5 | 0.5 |
| .bower | 5 | 0.5 |
| support.js | 5 | 0.5 |

**Table 3: Discrepancies in files of malicious npm packages**

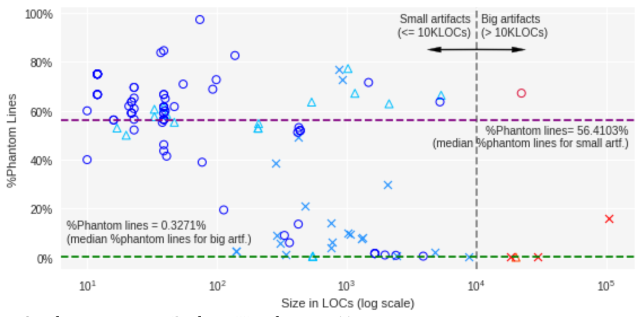
*'Src' (Source) denotes the number of LOC in the source code repository. Dark-gray-colored rows denote instances when attackers introduced brand new files, and light-gray-colored rows denote instances when attackers rewrote existing file completely.*

| Filepath | Attack Type | Number of LOC | | |
|---|---|---|---|---|
| | | Src. | Deleted | Injected |
| *commander-js-2.19.84/package.json* | Combosquatting | 79 | -79 | +1 |
| *commander-js-2.19.84/setup.js* | | 0 | 0 | +1 |
| *commander-js-2.19.84/update.js* | | 0 | 0 | +13 |
| *crossenv-6.1.1/package.json* | Combosquatting | 54 | -54 | +7 |
| *crossenv-6.1.1/package-setup.js* | | 0 | 0 | +17 |
| *eslint-scope-3.7.2/package.json* | Package Hijacking | 62 | -16 | +6 |
| *eslint-scope-3.7.2/lib/build.js* | | 0 | 0 | +9 |
| *kraken-api-0.1.8/package.json* | Package Hijacking | 33 | 0 | +3 |
| *kraken-api-0.1.8/lint.js* | | 0 | 0 | +15 |
| *chalc-2.0.0/package.json* | Typosquatting | 78 | -74 | +7 |
| *chalc-2.0.0/index.js* | | 213 | -213 | +12 |
| *uglyfi-js-3.4.6/package.json* | Typosquatting | 56 | -56 | +1 |
| *uglyfi-js-3.4.6/lib/coprophagan.js* | | 0 | 0 | +21 |

**Table 4: Average distribution of phantom files and lines found by our implementation**

***%Artf.*** *denotes the number of artifacts with specific attack type in our evaluation dataset.* ***%All Files Injected*** *denotes the percentage of artifacts that* ***all*** *files are phantom (either brand new or contain phantom lines).* ***% ≥50% Lines Injected*** *denotes the percentage of artifacts that more than equal to 50% of their LOCs are phantom.*

| Type | #Artf. | Avg. %Phantom | | %All Files | % ≥50% Lines |
|---|---|---|---|---|---|
| | | Files | Lines | Injected | Injected |
| Combosquatting | 74 | 94.6% | 54.5% | 91.9% | 78.4% |
| Package Hijacking | 24 | 47.8% | 16.2% | 8.3% | 8.3% |
| Typosquatting | 21 | 85.0% | 50.0% | 81.0% | 81.0% |
| All | 119 | 83.5% | 46.0% | 73.1% | 64.7% |



*○: Combosquatting, ×: Package Hijacking, △: Typosquatting.*
*Blue marks denote small artifacts (less than equal 10KLOCs) and red marks denote big artifacts (greater than 10KLOCs). Package hijacking cases tend to happen in bigger artifacts, while combosquatting is more in smaller artifacts and typosquatting is more evenly spread.*

**Figure 3: Percentage of Phantom lines of malicious npm artifacts in comparison with the artifact size.**

version containing a new JavaScript file added as a post-installation script (leaving the other files unchanged).

**Brand new files.** Seventy one malicious artifacts (59.7%) in the dataset contain brand new files that do not exist before. For instance, as shown in Table 3, the malicious artifact `commander-js-2.19.84` contains two brand new files, namely *setup.js* and *update.js*. The file *setup.js* is added as a post-installation script in the *package.json*. This particular script will spawn a new process in which the malicious code in *update.js* will be called. Another interesting example of this case is the malicious artifact `kraken-api-0.1.8` which not only mimics the legitimate artifact in all files, but also introduces a brand new file: *lint.js*. This file, as confirmed by our manual validation, contains a malicious code as a post-installation script.

Table 4 shows that in typosquatting and combosquatting attacks, attackers significantly introduced brand new files compared to the package hijacking attack. This is likely because the package hijacking attack is considered to be more complex and stealth than the squatting attacks. We also noticed that, in certain instances (e.g., `commander-js-2.19.84`), attackers removed all existing lines and add one or more new malicious lines.

Interested on this phenomena, we want to observe whether it is related to the artifact size. For this purpose, we observe the distribution of the percentage of phantom lines in comparison with the size of npm artifacts in Figure 3. We then found that package hijacking cases tend to happen in bigger artifacts than combosquatting and typosquatting. In more detail, we observed that the percentage of the phantom lines for smaller artifacts (less than equal ten KLOCs) tend to be higher than bigger artifacts.

## 6 RESULTS

### 6.1 Performance efficiency

To find such discrepancies shown in Table 3, a possible approach is to use GIT-LOG command [3]. This command iterates over all commits in a repository to find information whether a specific line is present at some point in the history of the related repository. In most cases, the result contains the information about the commit in which the line was introduced.

To measure the performance of our approach compared to GIT-LOG, we run both approaches on the top ten popular npm packages [1] and measure the time required for the analysis. As shown in Table 5, on average, our approach only takes 10.6% of GIT-LOG's processing time. The GIT-LOG command does not scale for large packages such as `commander` or `request` because it iterates over all revisions each time it is invoked.

Our implementation inherits [20] by pre-processing all commits in a repository and scanning all the code only once. This approach allows our tool to scale well even with a large amount of versions in the same packages to check. On the other hand, GIT-LOG iterates through all revisions each time it is invoked, hence it does not scale for big packages, especially the packages linked to big repositories. Indeed, by default, GIT-LOG approach requires more resources and

**Table 5: Performance comparison of git log approach vs our tool's approach on top 10 selected packages**

*The evaluation was done for all versions on each package. The number for GIT LOG in `commander` and `request` package are logical estimations as they went over our time limit (3h) to process all versions (the product of the execution time for 10 versions and the total number of versions). "How much faster" column shows how much faster is our tool's performance (LOC/sec) vs. GIT-LOG.*

| Package | Size (LOC) | GIT-LOG (sec) | Our tool (sec) | How much faster |
|---|---|---|---|---|
| chalk | 24 992 | 1807 | 83 | 21.8x |
| commander | 257 066 | 29 780* | 837 | 35.6x |
| debug | 92 591 | 6188 | 471 | 13.1x |
| loose-envify | 1248 | 72 | 17 | 4.2x |
| minimist | 24 162 | 1141 | 48 | 23.8x |
| ms | 34 713 | 1927 | 49 | 39.3x |
| object-assign | 2484 | 130 | 51 | 2.5x |
| prop-types | 44 223 | 3097 | 472 | 6.6x |
| request | 544 463 | 65 660* | 1935 | 33.9x |
| tslib | 29 610 | 1942 | 75 | 25.9x |
| Average | 105 555 | 11174 | 376 | 20.7x |

time to process packages. To illustrate the improvement, we compare the smallest package (`loose-envify`) and the biggest package (`request`) on our ten samples, where `request` is 436.3x the size of `loose-envify`. In this case, GIT-LOG took 911.9x longer to process `request`, but our implementation only took 113.8x longer. On average, our implementation is 20.7x faster than GIT-LOG, which even exceed the performance efficiency of LASTPYMILE (16x faster than GIT-LOG).

Current approaches on identifying malicious packages, such as GIT-LOG, happen to be time-consuming and resource-intensive as they scan the entire package. The performance evaluation shows that the approach adopted by our tool is in average **20.7x faster** than the GIT-LOG approach.

## 6.2 Effectiveness when combined with other npm package scanners

To evaluate the effectiveness of our tool, we conducted a comparison analysis using OSS DETECT BACKDOOR [12] (ODB in short), one of OSS GADGET's [13] utilities that is able to identify potential backdoors and malicious code within an artifact. We selected OSS DETECT BACKDOOR among the other state-of-the-art tools for many reasons: it is a practical approach that is lightweight, scales well (the performance does not change depending on the size of the artifact), and scans the whole code of a package.

We first tried to run ODB on malicious packages in the dataset. However, the tool was not able to directly process the packages due to technical issues happening when launching the tool. Therefore, we built a lightweight scanner using rules from OSS DETECT BACKDOOR [12], which are 37 regular-expressions-based rules. We then ran the scanner on the phantom files/lines (files/lines returned by our tool) and on the whole artifact. The generated alerts then went through our manual validation, where two researchers independently validated the alerts. They then looked at the results together and discussed with the third researcher (who was not involved in the preliminary validation process) to resolve any conflicts.

Table 6 shows the alerts generated by ODB on the latest versions of top npm packages [1]. We observed that running ODB rules on the whole artifacts resulted in many alerts. On the other hand, the scanning of phantom files produced by our tools produced no

**Table 6: OSS DETECT BACKDOOR's rules for top 10 popular legitimate npm packages in whole artifact vs. phantom files.**

*We observed that even when the artifact is legitimate, the rules from OSS DETECT BACKDOOR still generate false alerts on certain files (e.g. markdown files) when scanning the whole artifact.*

| Artifact | #Whole Artifact | #Phantom files only |
|---|---|---|
| chalk-5.0.1 | 10 | 0 |
| commander-9.0.0 | 18 | 0 |
| debug-4.3.3 | 4 | 0 |
| loose-envify-1.4.0 | 3 | 0 |
| minimist-1.2.5 | 0 | 0 |
| ms-2.1.3 | 0 | 0 |
| object-assign-4.1.1 | 0 | 0 |
| prop-types-15.8.1 | 0 | 0 |
| request-2.88.2 | 106 | 0 |
| tslib-2.3.1 | 3 | 0 |

**Table 7: OSS DETECT BACKDOOR's rules for malicious packages in whole artifact vs. phantom files.**

*#Total: the total alerts produced by the rules. #FP: LOCs that are classified as malicious while actually they are not. We used the rules from OSS DETECT BACKDOOR tool for this evaluation. In particular, we observed that the number of false alerts is significantly reduced when scanning only phantom files instead of the whole artifact.*

| Artifact | Whole artifact | | | Selected by our tool | | |
|---|---|---|---|---|---|---|
| | #Total | #FP | %FP | #Total | #FP | %FP |
| colour-string-1.5.3 | 7 | 5 | 71.4% | 2 | 0 | 0.0% |
| commander-js-2.19.84 | 9 | 8 | 88.9% | 1 | 0 | 0.0% |
| eslint-scope-3.7.2 | 29 | 28 | 96.5% | 1 | 0 | 0.0% |
| foever-0.15.3 | 47 | 45 | 95.7% | 3 | 1 | 33.3% |
| grunt-radical-0.0.14 | 16 | 15 | 93.7% | 2 | 1 | 50.0% |
| kraken-api-0.1.8 | 7 | 7 | 100% | 2 | 2 | 100% |
| react-datepicker-plus-2.4.2 | 103 | 84 | 81.5% | 20 | 1 | 5.0% |
| sailclothjs-1.2.6 | 27 | 24 | 88.9% | 4 | 1 | 25.0% |
| uglyfi-js-3.4.6 | 52 | 50 | 96.1% | 2 | 0 | 0.0% |
| yeoman-genrator-3.1.1 | 18 | 16 | 88.9% | 2 | 0 | 0.0% |
| Average | | | 90.2% | | | 21.3% |

alerts. Our manual validation suggests that all the alerts generated by ODB are false alerts (FPs). For example, many of the alerts are related to the README files (because they contain markdown code that trigger the backdoor patterns in the rules) which make them false alerts. On the other hand, there are alerts on JavaScript code because there are certain patterns in the code that trigger ODB's rules as they are too generic (e.g. ".get" or ".platform" matching strings). This result shows that our approach can improve the tool performance in distinguishing legitimate packages and malicious packages in npm ecosystem, as LASTPYMILE also did for Python ecosystem.

Next, to evaluate the effectiveness of our solution in identifying malicious code, we ran a similar evaluation on the malicious npm artifacts in Backstabber dataset [16]. Table 7 shows the alerts produced by running ODB rules on the whole malicious artifacts and the phantom files produced by our tool.

Our manual validation confirms that 90.2% of the alerts generated by scanning the whole artifacts with ODB's rules are false positives (FPs) while scanning only the phantom files (the files flagged by our tool) with the same rules drops the false positive rate to 21.3%. On average, scanning the phantom files only produced 13.9%. This result aligns with LASTPYMILE in reducing the number of FP alerts. Regarding false negatives (FN), ODB only missed the malicious code (TP) in `kraken-api-0.1.8` and found all of them in other artifacts (recall ~100%). All the TPs found while scanning the whole artifact are also found while scanning only phantom files, or in other words: we do not lose any TP with this approach. Even in `kraken-api-0.1.8`, ODB already missed the TPs (FNs) while scanning the whole artifact. Therefore, we would argue that our proposed solution does not degrade the recall of the analysis tool, while significantly improving the precision.

Although scanning the whole artifacts provide much more coverage, scanning only the phantom (injected) files significantly produces **(4.2x) less false alerts (FPs)**, while providing the same number of detected TP files **(precision improved significantly while recall stays the same)**.

## 7 THREATS TO VALIDITY

*The limitations of the Backstabber dataset [16].* The dataset has unbalanced proportions of attack types. However, it could be considered sufficient for our analysis, as a preliminary analysis. Expanding the evaluation dataset would be a promising future work to evaluate the effectiveness of our tool.

*We consider only repositories hosted on Github.* Although almost all of the samples are packages currently hosted on Github, there may be few instances that are hosted on other platforms. However, we would argue that our evaluation is general enough, as Github is the largest code host with more than 190 million repositories, of which JavaScript covers 14.1%, the second on the most-used-language rank [2].

*The possibility of developers moving around the code in repository.* This behaviour impacts the performance and results produced by our tool because it changes the file hashes. With this kind of changes, our tool would report those files as unseen, and developers would need to check them manually to know which lines have been introduced. However, this phenomena seems quite rare and only a small number of files are affected.

*Only checking the absent code from the repository.* Even though we did not find any package doing so in the dataset, malicious code could be added in the source repository of a package [9]. This case is out of the scope of our study, as the malicious code can be spotted by developers who review the code.

## 8 CONCLUSION AND FUTURE WORKS

We have ported LastPyMile approach to detect discrepancies between source code and artifacts in npm ecosystem. The approach has been tested on malicious artifacts and the top ten legitimate packages in npm. Our preliminary evaluations on both legitimate and malicious artifacts suggest the feasibility of the integration with existing package scanners such as ODB. Our implementation is efficient (20.7x faster than git-log) while also scales to large packages such as the `commander` package.

On the ten malicious npm artifacts in the *Backstabber* dataset, the combination of our tool with the existing ODB's malware rules managed to reduce the ratio of false alerts from 90.2% to 21.3%, a four fold drop, compared to using ODB directly. Furthermore, scanning the latest versions of the top ten popular npm packages using our approach resulted no false positives, as it should be for legitimate packages.

Future step is to scale the evaluations to more JavaScript packages in the npm ecosystem and all malicious packages in the *Backstabber* dataset. As a natural step, we plan to develop the malware detection rules to scan the discrepancies between npm packages and their source code repositories. We also plan to publish the implementation as an open-source project and integrate it with the security pipeline used by npm.

## ACKNOWLEDGMENTS

This research was done while Duc-Ly Vu was with the University of Trento. This work has been partly supported by the European Union H2020 Program under the Grant 952647 (AssureMOSS - www.assuremoss.eu).

## REFERENCES

[1] 2019. Most popular npm packages. https://gist.github.com/anvaka/8e8fa57c7ee1350e3491. Accessed: 2022-03-10.
[2] 2021. GitHut 2.0: A small place to discover languages in Github. https://madnight.github.io/githut/#/pull_requests/2021/4. Accessed: 2022-03-20.
[3] [n.d.]. Git log. https://git-scm.com/docs/git-log/. Accessed: 2022-03-10.
[4] [n.d.]. NPM-Audit. https://docs.npmjs.com/cli/v8/commands/npm-audit/. Accessed: 2022-03-10.
[5] Catalin Cimpanu. 2018. Compromised JavaScript Package Caught Stealing npm Credentials. https://www.bleepingcomputer.com/news/security/compromised-javascript-package-caught-stealing-npm-credentials/. (2018).
[6] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. https://arxiv.org/pdf/2002.01139.pdf. (2021). https://doi.org/10.48550/arXiv.2002.01139
[7] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing malicious package updates in NPM with a lightweight permission system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1334–1346. https://doi.org/10.1109/ICSE43902.2021.00121
[8] github. 2020. GitHub. https://github.com/. Accessed: 2022-03-09.
[9] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. 2021. 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, 258–267. https://doi.org/10.1109/ICSE-SEIP52600.2021.00035
[10] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2022. Taxonomy of Attacks on Open-Source Software Supply Chains. *arXiv preprint arXiv:2204.04008* (2022).
[11] Genpei Liang, Xiangyu Zhou, Qingyu Wang, Yutong Du, and Cheng Huang. 2021. Malicious Packages Lurking in User-Friendly Python Package Index. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 606–613. https://doi.org/10.1109/TrustCom53373.2021.00091
[12] Microsoft. 2020. OSS Detect Backdoor. https://github.com/microsoft/OSSGadget/wiki/OSS-Detect-Backdoor.
[13] Microsoft. 2020. OSS Gadget: Collection of tools for analyzing open source packages. https://github.com/microsoft/OSSGadget.
[14] npm Inc. 2019. npm. https://www.npmjs.com/. Accessed: 2022-03-08.
[15] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. 2020. Supporting the Detection of Software Supply Chain Attacks through Unsupervised Signature Generation. *arXiv preprint arXiv:2011.02235* (2020).
[16] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In *Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 23–43. https://doi.org/10.1007/978-3-030-52683-2_2
[17] Synopsys. 2021. Open Source Security and Risk Analysis Report. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.pdf. (2021).
[18] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending against package typosquatting. In *International Conference on Network and System Security*. Springer, 112–131. https://doi.org/10.1007/978-3-030-65745-1_7
[19] Duc-Ly Vu. 2021. py2src: Towards the Automatic (and Reliable) Identification of Sources for PyPI Package. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1394–1396. https://doi.org/10.1109/ASE51524.2021.9678526

[20] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. 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*. 780–792.

[21] Duc-Ly Vu, Ivan Pashchenko, and Fabio Massacci. 2020. A qualitative study of dependency management and its security implications. In *2020 ACM SIGSAC Conference on Computer and Communications Security*. IEEE, 1513–1531. https://doi.org/10.1145/3372297.3417232

[22] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2093–2095. https://doi.org/10.1145/3372297.3420015

[23] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and combosquatting attacks on the python ecosystem. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 509–514. https://doi.org/10.1109/EuroSPW51379.2020.00074

[24] Nusrat Zahan, Laurie Williams, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, and Chandra Maddila. 2021. What are Weak Links in the npm Supply Chain? *arXiv preprint arXiv:2112.10165* (2021).