What the Fork? Finding Hidden Code Clones in npm

Elizabeth Wyss
University of Kansas
Lawrence, KS, USA
ElizabethWyss@ku.edu

Lorenzo De Carli
Worcester Polytechnic Institute
Worcester, MA, USA
ldecarli@wpi.edu

Drew Davidson
University of Kansas
Lawrence, KS, USA
DrewDavidson@ku.edu

ABSTRACT

This work presents findings and mitigations on an understudied issue, which we term shrinkwrapped clones, that is endemic to the npm software package ecosystem. A shrinkwrapped clone is a package which duplicates, or near-duplicates, the code of another package without any indication or reference to the original package. This phenomenon represents a challenge to the hygiene of package ecosystems, as a clone package may siphon interest from the package being cloned, or create hidden duplicates of vulnerable, insecure code which can fly under the radar of audit processes.

Motivated by these considerations, we propose UNWRAPPER, a mechanism to programmatically detect shrinkwrapped clones and match them to their source package. UNWRAPPER uses a package difference metric based on directory tree similarity, augmented with a prefilter which quickly weeds out packages unlikely to be clones of a target. Overall, our prototype can compare a given package within the entire npm ecosystem (**1,716,061** packages with **20,190,452** different versions) in **72.85** seconds, and it is thus practical for live deployment. Using our tool, we performed an analysis of a subset of npm packages, which resulted in finding up to **6,292** previously unknown shrinkwrapped clones, of which up to **207** carried vulnerabilities from the original package that had already been fixed in the original package. None of such vulnerabilities were discoverable via the standard npm audit process.

ACM Reference Format:
Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2022. What the Fork? Finding Hidden Code Clones in npm. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510168

1 INTRODUCTION

The security and correctness of code stored in package repositories is an important concern because such repositories are crucial to modern software infrastructure. Indeed, language-based package repositories such as npm, pypi, and RubyGems collectively serve billions of packages each week [39]. Much of the popularity of package repositories is due to the package manager frontend, which allows a user to easily import a package by issuing a simple install directive on the command line. While seamless import of external code is convenient, it also creates problems: developers tend to assume code to be reliable rather than vetting prior to import [6], and once a package is imported, latent bugs and vulnerabilities become part of the final application. Importing the "wrong" package may cause significant supply-chain security issues [17].

This work presents findings and mitigations on an understudied issue within the npm package repository[1], which we term *shrinkwrapped clones*. We use this term to refer to packages which are uploaded to a package repository and contain code that is identical, or nearly-identical, to that of an existing (legitimate) package. Specifically, we discover two types of clones: (i) identical clones, which contain source code that is identical to that of an existing package, and (ii) close clones, which make potentially significant syntactic/semantic changes to the code, but generally localized to a small number of files (we refine this definition in the following). This phenomenon is characteristic to npm, as this ecosystem lacks the notion of forks, by which we mean copied code repositories that explicitly link back to their source repositories (as is common for example in GitHub [2]). Instead, shrinkwrapped clones in npm cannot be explicitly linked back their source packages since npm lacks official mechanisms for forking packages.

The phenomenon of shrinkwrapped clones represents a challenge to the hygiene of package repositories; in particular, it contributes to the problem of *confusability* of npm packages. npm contains more than 1.7 million packages, and while the ecosystem provides a robust search interface, it provides no assistance in choosing the most appropriate package to provide a desired functionality. Previous work on typosquatting attacks [39] suggests that it is fairly common for developers to install a package different from the one they intended.

Clones exacerbate these problems. Most obviously, a clone package causes confusion, as clone packages are oftentimes named similarly to the original package, and in many occasions they also reuse their metadata (such as the package description). Users of the package repository may thus misattribute the provenance of a package, giving credit to the wrong developer for creating a particular codebase. However, we also find that a non-trivial number of clone packages are rarely maintained and fail to include updates to the package being copied. The users of the clone are thereby locked into older versions of functionality and, crucially, forgo any bug fixes that are applied to the package being cloned. In effect, the users of the clone are subject to vulnerabilities which have already been patched. Moreover, clones can exist deep within package dependency trees, which means that installing a package that (transitively) depends on a clone also implicitly installs

[1]We choose npm as our repository of interest because it is the largest and most popular.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510168

that clone via npm's automated dependency resolution mechanism. For these reasons, a user may be unaware that they have downloaded a clone, wherein vulnerabilty disclosures fail to propagate, because there is no facility in npm to link a clone to the package that has been cloned.

In this work, we study the scope and impact of shrinkwrapped clones. Detecting shrinkwrapped clones presents several challenges. First, there is no agreed-upon standard on what extent of code-reuse constitutes a code clone [21, 22, 25, 29, 30, 38]. Second, close clones - i.e., clones that make changes to the original package - exhibit local but complex syntactic modifications (e.g. translating CommonJS syntax to ES6). Thus, they are incompatible with many existing clone detectors, which focus on the insertion and deletion of statements [25, 29] (we further review related work in Section 6). Finally, in order to determine whether a package clones any other, the package must be compared to the entirety of npm, which leaves limited to no time for any form of code analysis. We address these challenges with a parametrizable heuristic designed for pairwise package comparison. The heuristic defines shrinkwrapped clones via a tunable file-granularity syntactic distance threshold. With this approach, any two packages whose distance is below the threshold are considered close clones, or identical clones if the distance equals one[2] (we discuss strategies for tuning the threshold in Section 3). The heuristic only considers whole-file hashes and eschews tokenization and any form of lexical analysis, making it computationally efficient - a package can be matched for clones against the entirety of npm in a matter of seconds. Based on this heuristic, we propose UNWRAPPER: a mechanism to detect when a package is a clone of another more popular package.

Explicitly linking a clone to the package from which it came has several benefits. It enhances the provenance information of packages, restoring the connection to the originators of the package. Furthermore, it allows users to avoid clone packages, preventing them from unwittingly using a less-maintained copy of a codebase and directing them to the original. Finally, users can be made aware of any vulnerabilities that may have been reported in the original version of the package. Thus, our approach constitutes a turnkey solution, suitable for operating over the package repository with no manual intervention.

Overall, our work makes the following contributions:

- We identify and characterize the problem of shrinkwrapped clones in the npm package repository.
- We propose UNWRAPPER, a technique to check when a package is a shrinkwrapped clone of any other package.
- We evaluate UNWRAPPER and find that it is effective in practice to identify shrinkwrapped clones with reasonable time and space overhead (the majority of packages can be compared to the existing **20M** package set for clones in **72.85 seconds** using off-the-shelf hardware).
- We report our findings based on the analysis of a subset of npm. Our analyses identified up to **6,292** clones. Up to **2159** relied on vulnerable, outdated dependencies. Furthermore, up to **207** clones directly incorporated vulnerabilities which were not discoverable via the npm audit process.

Additionally, we publicly release our code and supporting data, freely distributed via the Open Science Framework [5]:

`https://osf.io/jfk3n/?view_only=6f930d1de8704a26903540f75982bffb`

The remainder of our paper is structured as follows. Section 2 reviews background material. Section 3 provides technical details of the clone detection methodology. We describe results of our experiments in Section 4, and discuss our findings in Section 5. We review related work in Section 6, and conclude in Section 7.

2 BACKGROUND

2.1 npm

npm consists of a package manager and a repository for software developed for the Node.js environment. npm is the largest online language-based software ecosystem [41]; it contains millions of publicly available packages and weekly download counts range from hundreds of millions to billions. Similar to other package management tools, the primary goal of npm is to simplify third-party code reuse by managing software dependencies. When a user issues the `npm install` command, the front-end constructs a tree of all required dependencies and then installs each package within the tree without the need for user involvement. npm allows packages to be downloaded freely (either directly or as dependencies of another package), and allows new packages to be uploaded without any external moderation. However, the maintainers have a security team whose goal is to detect and remove explicitly malicious packages, either via internal analysis or by collecting external reports [31].

2.2 Versioning and Forking

Development in npm is characterized by fast-evolving code and extensive code reuse via import of external dependencies, even for trivial functionality [6]. To help prevent breakage in a package due to the evolution of its dependencies, npm automatically freezes dependencies within a compatible version range, and further allows developers to specify exact dependency versions via a process known as *shrinkwrapping*. Using this feature guarantees that dependencies will always be fetched at the version they were originally imported. However, npm offers no official support for developers wishing to modify a third-party package prior to importing it. In other words, there is no explicit concept of a *fork* as in other ecosystems such as GitHub [2]. As such, developers seeking to modify an existing package must download, alter, and then republish the modified package under a new name. For this reason, modified packages on npm are difficult to detect; to make matters worse, when a package is manually duplicated, information attached to that package is lost. Critically, this lost information includes both the original version history and any known vulnerabilities associated to the original package (typically retrievable via `npm audit`). Thus, the act of duplicating a package implicitly hides

[2] npm packages contain a unique-per-package metadata file which limits package distance to a minimum of 1.

```
opts.log.http(
  'fetch',
  `${method.toUpperCase()} ${res.status}
   ${res.url} ${elapsedTime}ms${attemptStr}
   ${cacheStr}`
)
```

Listing 1: Vulnerable code snippet present in unpatched versions of `npm-registry-fetch` and in the most up-to-date version of `@evocateur/npm-registry-fetch`

```
let urlStr
try {
  const { URL } = require('url')
  const url = new URL(res.url)
  if (url.password)
    url.password = '***'

  urlStr = url.toString()
} catch (er) {
  urlStr = res.url
}

opts.log.http(
  'fetch',
  `${method.toUpperCase()} ${res.status}
   ${urlStr} ${elapsedTime}ms${attemptStr}
   ${cacheStr}`
)
```

Listing 2: Nonvulnerable code snippet present in patched versions of `npm-registry-fetch`

important information including authorship, whether more recent versions of the same package exist, and any known security issues.

2.3 Shrinkwrapped Clones

The goals of our work are (i) to measure how frequently developers duplicate—and potentially modify— third-party packages into one of their packages; (ii) to determine whether this process can have negative security implications; and (iii) to devise techniques for identifying the "missing link" between duplicate npm packages and the original package. One question we *do not* investigate is *why* developers perform such duplication; as this would require making inferences or assumptions about developers' intentions and goals. However, we did observe that in many cases duplicates are exact copies of the original package; therefore, including a duplicate accomplishes the same result as shrinkwrapping the original package at the duplicated version. It is for this reason that we refer to these packages as *shrinkwrapped clones*. An example includes the package redux-form-v6, which is an exact copy of redux-form at version **6.4.3**.

While we use shrinkwrapped clones as an umbrella term for all duplicates, we also note that in some cases developers make small functional changes to duplicate code prior to republishing. A manual review of such cases reveal that in the large majority of cases the developer intends to remove functionality (as in the case of `@cypress/listr-verbose-renderer`, which is near-identical to `listr-verbose-renderer` but disables logging) or to implement programmer preferences (`@xtuc/ieee754` is a clone of `iee754` but reconstructed using CommonJS syntax rather than ES2015 syntax).

The observation above warrants further analysis: to what extent *should* modifications made to a package cause a duplicate to not be considered a clone? Rather than addressing the question above—which cannot be answered quantitatively—we empirically define a difference metric and a threshold below which two packages are labeled as clones. We provide details on metric and threshold in Section 3, and further discuss our findings in Section 4.

Metric	**npm**
Total Unique Packages	1,716,061
Total Unique Package Versions	20,190,452
Total Size (Compressed)	13 TB

Table 1: Statistics related to the size of the npm package repository

2.4 Vulnerabilities Introduced by Clones

As described in Section 1, shrinkwrapped clones can carry latent vulnerabilities from the time of their creation. One such example is present in the `@evocateur/npm-registry-fetch` package, which is a clone of the `npm-registry-fetch` package. Versions of `npm-registry-fetch` prior to version 4.0.5 contain a sensitive data exposure vulnerability in which private information such as password values are written to stdout and log files [**?**]. The vulnerable code snippet is depicted in Listing 1, and the relevant patched code snippet that fixes the vulnerability is depicted in Listing 2. `@evocateur/npm-registry-fetch` copied the codebase of `npm-registry-fetch` at version 3.9.1, which contains the known sensitive data exposure vulnerability. Although `@evocateur/npm-registry-fetch` has received two minor updates since it was first published, none of these updates fix the sensitive data exposure vulnerability that still remains in the package's codebase.

2.5 Scale of npm Analysis

Table 1 describes the total size of npm in terms of packages, versions, and storage. It is a nontrivial problem to individually analyze millions of package versions, accounting for over 13 terabytes of package data in total. As such, we focus our efforts on caching metadata and applying heuristics to analyze such metadata at scale. This metadata-based approach is consistent with ongoing industry efforts to analyze large-scale open source package repositories such as the Package Feeds [4] and Kritis [3] projects. We believe that our metadata-based approach is efficient and sufficient in detecting shrinkwrapped clones; we describe our approach in detail in Section 3, and

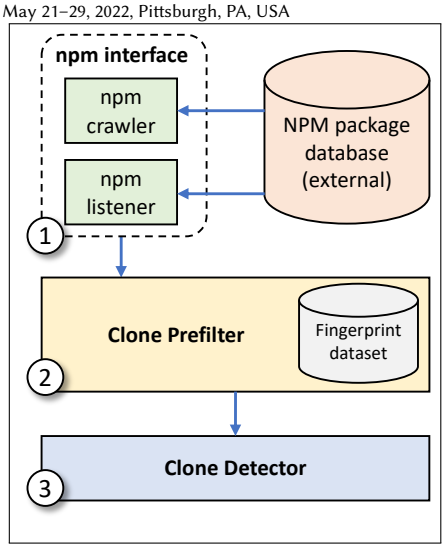

Figure 1: Workflow of UNWRAPPER from collection of packages to classification

we provide justification for the effectiveness of this approach in Section 4.

3 UNWRAPPER DESIGN

Based on our observations described in Section 2, we design UNWRAPPER to identify instances of shrinkwrapped clones that exist within npm. In this section, we describe our goals for this tool and how our design meets those goals.

3.1 UNWRAPPER Assumptions and Goals

At a high level, we are interested in identifying instances of packages that are *similar* to another known package. Furthermore, we also intend to propose practical techniques to identify these objects at ecosystem scale. We assume that while a developer may fail to explicitly acknowledge the source of a shrinkwrapped clone, no attempt is made to obfuscate similarity at the code level. This assumption is based on the lack of incentive to do so; most npm package code is provided under permissive licenses which allow reuse [24], and there is no direct negative consequences for code duplication.

We envision UNWRAPPER being used to *retroactively* detect shrinkwrapped clones that are already present in npm, restoring the provenance of the package code. Using UNWRAPPER in this way allows users of npm to detect whether packages that they depend on are shrinkwrapped clones and to suggest the original package that they may prefer. We also envision UNWRAPPER being used to *proactively* detect if a new package being added to npm is a clone at the time it is published.

The popularity of npm makes analysis challenging – the number of existing packages already in the repository is significant. Furthermore, we observe that a key problem with shrinkwrapped clones is that they are clones of non-current versions of other packages. Thus, UNWRAPPER needs to match any package of interest against the entirety of the npm ecosystem, including all versions of all packages. Additionally, approximately 850 new packages are uploaded daily to npm [15]. UNWRAPPER must remain able to process all new packages without slowing down their deployment.

3.2 Design Overview

Based on the discussion above, we design UNWRAPPER with the goal of being able to scale with the growth of a repository; it must be relatively fast and lightweight in its analysis and be capable of running in an environment separate from the repository itself. The overall UNWRAPPER pipeline is depicted in Figure 1. Packages are initially acquired using an npm crawler and listener (module **1** in Figure 1). The task of determining whether a package is a clone of another is primarily carried by the Clone Detector component (module **3**). Our approach to shrinkwrapped clone detection leverages the identification of differences between directory trees of candidate original-clone pairs–where each file node is labeled by its name and checksum, and no other information about the file is considered. This approach does not require performing any code analysis other than checksum computation and is consistent with the assumption of non-adversarial settings.

However, we found that while this approach is efficient, it is too time-intensive for live analysis of new package uploads. Thus, we augmented it with a prefiltering step (module **2**). This prefilter sits between the package dataset and the clone detector, and quickly weeds out packages unlikely to be shrinkwrapped clones. In the following, we describe each component in detail.

3.3 npm Interface

The frontend of UNWRAPPER interfaces with npm to collect information from every version of every package that is currently available and analyzes new packages in a timely fashion. We accomplished package collection by implementing an npm crawler which downloads and stores each package version locally. This approach allows packages to be accessed rapidly and without burdening the repository with analysis requests. Since changes to packages are necessarily made as new versions, the package database remains accurate for all existing package versions and only needs to be updated with new versions as they are added to the repository. To support updates, we implement an npm listener that triggers whenever a new package – or a new version of an existing package – is added to the repository. New entries are automatically added to the local package database and queued for detection as shrinkwrapped clones. The listener uses the npm webhook system and is automatically notified of updates without the need for polling. Using this system means that new packages and versions are added to the analysis queue in real-time.

3.4 Clone Detector

This component takes in a shrinkwrapped clone package candidate pair, and evaluates their similarity by computing a domain-specific pairwise difference metric which we term

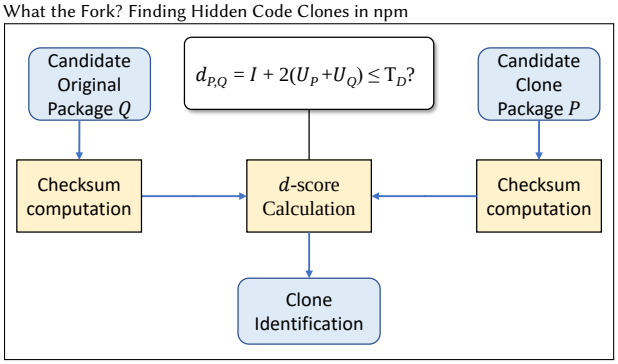

Figure 2: Operation of the Clone Detector

d-score. More precisely, the difference between two packages P and Q is defined as:

$$d_{P,Q} = I + 2(U_P + U_Q)$$

where I is the number of identically named files with different checksums amongst both packages, and U_P, U_Q are the number of files unique (name-wise) to only one of the packages. As such, packages with a lower difference score are more similar than packages with a higher difference score, and packages with a difference score of one are semantically identical, differing only in their package.json metadata file which is guaranteed to be unique to each package. A candidate package is considered to be a clone of the input package if their pairwise d-score is less than a given threshold T_D across *any* version of either package. We present the design of our clone detector in Figure 2.

3.4.1 Difference Threshold. We determine an appropriate difference threshold by manually building an initial dataset of clone packages. To do so, we analyze the extent to which repository URLs[3] are duplicated across the 10,000 most popular npm packages. The initial analysis returned thousands of hits; the large majority of such hits are caused by packages which are sub-modules to other packages (e.g. the `lodash._getarray` package is a sub-module of the `lodash` package which exports the `getArray` function from `lodash`). To filter out these sub-module packages, we removed packages that shared maintainers with the packages that duplicated their repository URLs. Applying this filter left 38 packages, and manual review of these packages empirically showed all of them to contain codebases that were initially copied over from the packages that their duplicated repository URLs originate from.

We then examined the distribution of d-scores within the sample of 38 packages (plotted in Figure 3). **34 out of 38** clones have d-scores at or below 11. Conversely, analysis of a set of package pairs which do not include clones reveal d-scores in the tens or hundreds. The four identified clone packages with d-scores greater than 11 all made substantial changes to the core functionality of the packages that they cloned and had d-scores no less than 17. Thus, to minimize false positives, we

[3]npm packages can optionally express a source repository URL, such as a GitHub page, as metadata (we further analyze the effectiveness of source repository URL duplication as a clone detection signal in Section 4.3).

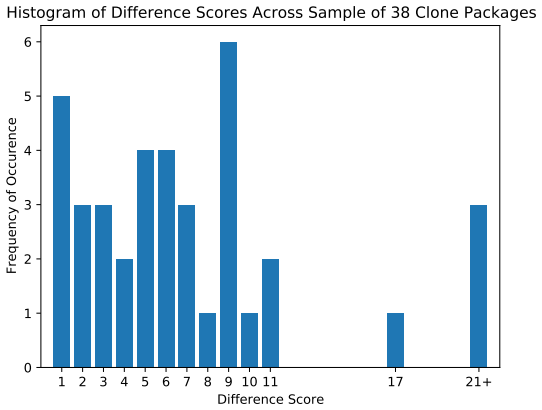

Figure 3: Distribution of difference scores across a sample of 38 manually identified clone packages

Clone File Tree Size	**d-score Threshold**
1	1
2	2
3	4
4	6
5	8
6	10
7+	11

Table 2: d-score threshold by clone file tree size

empirically choose **11** as the d-score threshold below which two packages are labeled as shrinkwrapped clones.

This d-score threshold is further tightened in cases where packages have very few files as to prevent small packages from being falsely over-reported as shrinkwrapped clones of other small packages. Table 2 depicts our determined d-score threshold as a function of file tree size. These thresholds were selected empirically by measuring the d-scores in cases where small packages were falsely identified as shrinkwrapped clones of other small packages.

3.5 Clone Prefilter

Our design goal of supporting real-time detection of shrinkwrapped clones requires a scalable analysis that accounts for the size and rapid growth of npm. Handling the scale of npm is a technical challenge in its own right. Our core similarity metric requires a pairwise comparison against each package version, of which there are over 20 million at the time of this writing. In recognition of this scalability challenge, we implement a clone prefilter mechanism which rapidly determines if a package is a candidate for pairwise similarity metric checking.

The first design goal for the prefilter is to speed up matching against the pre-existing package dataset. Furthermore, it should eliminate packages from consideration that are unlikely to be marked as clones during the full clone detection pass.

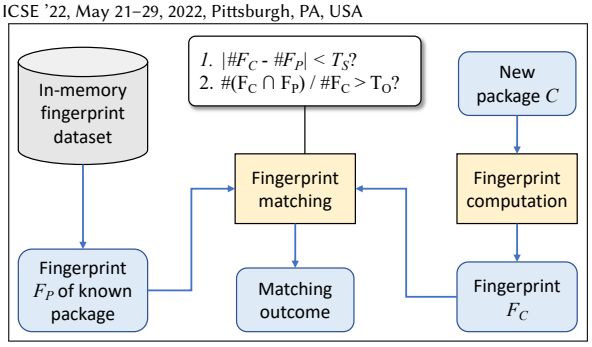

Figure 4: Functioning of the Clone Prefilter

In service of speeding up matching in the prefilter, we aim to precompute as much information as possible about known packages in the repository. However, given the scale of the npm ecosystem, it is also necessary to ensure that: (i) precomputing the necessary information for the entire set of existing packages does not become a significant bottleneck; and (ii) precomputed information has a memory footprint compatible with off-the-shelf hardware. We present a prefilter design meeting those conditions below; the approach is also depicted in Figure 4.

Like the clone detector, the prefilter uses folder structure similarity to identify potential clones; however, the similarity metric is purely based on file names (i.e. it uses no information concerning file content). Specifically, a package P is represented as a *fingerprint* F_P defined as $F_P = \{h(p_1), ..., h(p_n)\}$, where each of $p_1, ..., p_n$ is a package file path and $h(p) = sha256(p)_{1,32}$, i.e., the 32-bit prefix of the sha256 hash of the path[4]. Computing fingerprints for 20M npm packages takes **29.4 hours**. The resulting fingerprint set requires **80.2 GB** of in-memory storage. Therefore, this design meets our practicality requirements above.

Given a set of package fingerprints, the prefilter determines whether a target package C is a candidate clone for an existing package P as follows. First, it computes $|\#F_C - \#F_P|$, i.e., the difference in number of files in each packages. If the result is above a threshold T_S, the candidate is dropped. Second, it computes $\#(F_C \cap F_P)/\#F_C$, i.e., the overlap between fingerprints normalized by the number of files in the candidate clone. If the result is below a threshold T_O, the candidate is dropped. Additionally, any packages that share maintainers with package C are dropped (this is to prevent submodule packages and maintainer-intended duplicate packages from falsely being identified as clones), and further, if package C contains no source code files, it is excluded from consideration (this is to prevent trivial packages with no code from falsely being identified as clones); otherwise, the candidate is forwarded to the clone detector as depicted in Figure 1. Based on ROC curve analysis, we found that setting $T_S = 2$ and $T_O = 0.8$ results in a good balance of minimizing false negatives and false positives. In a production deployment, these parameters can be further tuned to satisfy the desired ROC characteristic of the prefilter.

[4]We use this hash function for convenience in our Python prototype; the functioning of the prefilter is largely orthogonal to the choice of function.

Table 3.1	**Prefilter Performance**		
Metric	**Min Time**	**Max Time**	**Avg Time**
Add a Package to Prefilter	0.566 ms	442 ms	5.24 ms
Test a Package Against Prefilter	54.22 s	203.79 s	70.81 s

Table 3.2	**Clone Detector Performance**		
Metric	**Min Time**	**Max Time**	**Avg Time**
First Test	174 ms	5,820 ms	319 ms
Additional Tests	168 ms	3,840 ms	255 ms

Table 3: Performance metrics related to our shrinkwrapped clone detector and prefilter

The rationale for choosing this particular combination of tests is based on the goal of approximating the d-score computation, which identifies packages as similar if they have most of their files in common. At the same time, the prefilter ignores file content, resulting in a vastly more efficient computation.

4 EVALUATION

To evaluate the practicality and effectiveness of our shrinkwrapped clones detection pipeline, we focus on answering three research questions:

- **RQ1:** Is the pipeline's performance satisfactory in generating prefilter fingerprints for the entire npm package repository?
- **RQ2:** Does the pipeline offer real-time performance capable of scaling with the growth rate of npm?
- **RQ3:** Is the pipeline effective in discriminating shrinkwrapped clones from novel packages?

In the remainder of this section, we detail our methodology used to answer these research questions, we discuss our results, and we present an analysis of detected clone packages.

4.1 Performance

We measure the minimum, maximum, and average time that our shrinkwrapped clone detector and prefilter require to perform their operations, and we present the measured performance metrics in Table 3. All measurements are collected across 1,000 independent trials on a CentOS Linux 8 server with an Intel Xeon Gold processor operating at 2.1 GHz.

The rest of this subsection discusses the online and offline performance of our shrinkwrapped clone detection pipeline as laid out in RQ1 and RQ2.

Prefilter Fingerprint Generation: Since shrinkwrapped clone detection relies on information about the file structure of every version of every npm package, our pipeline must be able to retrofit the entire npm package repository with reasonable performance. From analyzing every package publicly available on npm, we find that there exists 20,190,452 unique versions of packages on npm. All of these unique package versions must be added to our prefilter's fingerprint database in order for

our shrinkwrapped clone detection pipeline to function. Table 3.1, row 3, details the time to add a package to the prefilter. Given that our prefilter can add a single package version to its fingerprint database in an average time of 5.24 ms, generating a fingerprint database that contains every unique package version across npm requires 29.39 hours of CPU time. To lessen the required time, this process of adding packages to our prefilter's fingerprint database can be parallelized across multiple CPU cores to reduce the required time to a mere fraction of 29.39 hours. With parallelization in mind, we believe that this performance in generating prefilter fingerprints for the entire npm package repository is reasonable since retroactively generating the prefilter's fingerprint database is a process that only needs to be done once.

Real-time Clone Detection: Because of npm's rapid growth in packages per day, real-time shrinkwrapped clone detection requires performance that scales with the growth rate of npm. At the time of writing, npm is growing at rate of just over 850 new packages per day [15]. In order to keep up with a new package uploaded to npm, our pipeline requires that the new package is added to the prefilter and tested against the prefilter (ref. to Table 3.1), and then any positives reported by the prefilter must be verified with the clone detector. Table 3.2 details the time for the clone detector to determine whether a package is a clone of another. Since a candidate clone is typically matched against multiple potential matches from the prefilter, we report analysis times both for the first test and subsequent ones. Subsequent tests are typically faster as file hashes for the candidate clone need only to be computed once.

Given the performance results listed in Table 3, our shrinkwrapped clone detection pipeline can perform its required operations on just a single CPU core within 1/850th of a day as long as the prefilter reports fewer than 122 positives on average. In practice, we find that the majority of packages can be tested against our entire shrinkwrapped clone detection pipeline within 72.85 seconds. We note that our shrinkwrapped clone detection process is highly parallelizable and could easily utilize multiple CPU cores if the growth rate of npm or the total positives reported by the prefilter were to increase. As such, we consider the real-time performance of our pipeline to be more than reasonable with respect to the scale and growth rate of npm.

4.2 Effectiveness of Clone Detection

We now analyze the effectiveness of our tools in detecting shrinkwrapped clones as described in RQ3.

Clone Detector: Due to the inherent lack of ground truth regarding whether a package is a clone of another package, we rely on random sampling and manual vetting to verify the effectiveness of our shrinkwrapped clone detector. Our true positive verification process is as follows: We first collect a random sample of packages that our clone detector positively identifies as (non-identical) clones of another package. Then, both packages in the identified original-clone relationship are manually examined in terms of their file tree structures and file contents. Lastly, we mark identified clone packages as false positives if their clone relationship cannot clearly be identified from their package files. In a random sample of one hundred packages that our detector reports as clones, We find a total of 94 true positives and 6 false positives.

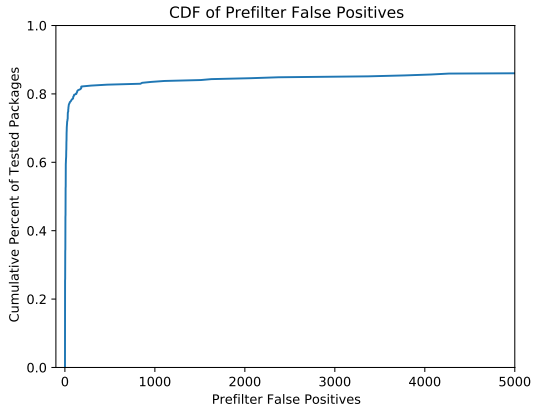

Figure 5: Distribution of prefilter false positives across all tested clone packages

We find that the false positives reported by our clone detector share a set of common properties that increase the difficulty of clone detection. They are all small packages with limited functionality, they contain very few files, and they have short and nondescriptive names–such as copy, merge, and capitalize. As such, these packages are similar to other small packages that provide independent implementations of similar functionality, and our clone detector can misreport them as clones. Despite the existence of these few false positives, we believe that our clone detector's precision of 94% is satisfactory in accurately detecting shrinkwrapped clones.

Prefilter: We utilize clone packages identified by our detector as the basis for ground truth in evaluating the effectiveness of our prefilter. We randomly sample 1,000 identified clone packages, test them against our prefilter, and then record the total number and kind of positives reported by the prefilter. From this experiment, we find that the recall of our prefilter (i.e., the percentage of correctly identified known clone relationships) is 95.3%. We present the cumulative distribution function of observed false positives identified by our prefilter in Figure 5. From this distribution, we find that the median number of false positives reported by our prefilter is 8, although there exists a small portion of input packages that generate false positive quantities in the thousands. This is consistent with the prefilter design goal of maximizing recall at the cost of precision. Given that the prefilter is merely the first step in our shrinkwrapped clone detection pipeline which involves verifying positives using our clone detector, we believe that the quality and quantity of reported positives is quite reasonable, especially with the parallelizable performance of our clone detector.

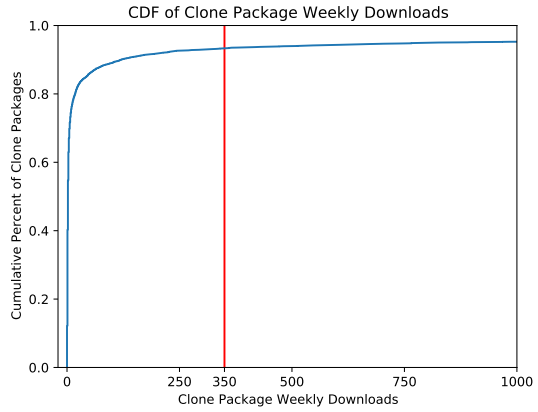

Figure 6: Distribution of total weekly downloads over all identified clone packages. The 6.7% of clone packages right of the red line have more than 350 weekly downloads and are likely installed by real users

Table 4.1	**Clone Dependents on npm**	
Dependent Type	**Identical (348)**	**Close (5,944)**
Total Dependents	397	6,496
Dependents by Same Maintainers	160	2,588
Dependents by Different Maintainers	237	3,908

Table 4.2	**Clone Repository URLs**	
URL Type	**Identical (348)**	**Close (5,944)**
Copied URL	210	3,153
Unique URL	121	2,602
No URL	17	189

Table 4: Statistics categorizing identical clones and close clones

4.3 Analysis of Clone Packages

Lastly, we analyze packages that our pipeline identifies as clones in order to quantify and categorize the posture of shrinkwrapped clones across the npm package ecosystem.

Quantifying Total Clones Across the npm Registry: To quantify how many clones potentially exist within the entire npm package registry, we collect a random sample of 6,000 npm packages and test our shrinkwrapped clone detection pipeline against those packages to estimate an upper bound of how many clones exist in the entirety of the npm package registry. Out of the 6,000 randomly sampled packages, our detection pipeline identifies 626, or roughly 10.4% of analyzed packages, to be clones of another npm package. By extrapolating this ratio to the entire npm package ecosystem (1,716,061 packages), we estimate that as many as 178,470 npm packages could be shrinkwrapped clones of other packages or be cloned by other packages. This sheer quantity of packages only amplifies the impact and dangers imposed by the existence of shrinkwrapped clones.

We further analyze the 626 packages positively identified as shrinkwrapped clones by our detection pipeline to quantify the extent to which name-similar clones likely exist within the npm package ecosystem. Clones that are similar in name to the packages that they clone pose a more serious threat to the health of npm given that they lead to much greater confusability in package provenance. We find that 175 out of the 626 identified shrinkwrapped clones, or approximately 28%, have package names such that the cloned package's name is a substring of the clone package's name. While there certainly exists shrinkwrapped clones with dissimilar names, we believe that focusing our analysis on the clones with the most potential to cause harm is the right direction for this work.

Identical Clones and Close Clones: In this analysis, we distinguish between two distinct subsets of clones that our pipeline detects: *identical clones* and *close clones*. Identical clones are identical in contents–character by character–to a specific version of another package (although they may differ in metadata). In contrast, close clones make some sort of modification or extension to the packages that they clone. From analyzing similarly named packages across the entire npm package registry, our shrinkwrapped clones detection pipeline identifies 348 identical clones and 5,944 close clones that are publicly available on npm.

Clone Popularity: The relative popularity of packages can be inferred from npm since the registry publicly provides the weekly download counts of all packages; Figure 6 depicts the distribution of weekly download counts across all identified clone packages. However, weekly download counts do not accurately represent the quantity of real users of a package since npm mirrors and bots routinely download packages for storage and analysis. The npm development team estimates that packages with fewer than fifty downloads per day are likely never installed by a real user [**?**]. From this metric of fifty downloads per day, we identify clones that have weekly download counts greater than 350 as likely installed by real users, and clones that have fewer than 350 weekly downloads as low-impact packages that are likely never installed by real users. We find that 21 out of the identified 348 identical clones and 399 out of the identified 5,944 close clones have more than 350 weekly downloads (ranging from a few hundred to more than ten million) and are very likely to impact real users.

Clone Dependents: Due to package dependencies across npm, it is possible that packages are indirectly installed and utilized as part of other packages. As such, we analyze the extent to which packages across npm depend on identified clone packages to further quantify the use of shrinkwrapped clones. Table 4.1 details npm packages that depend on identified clone packages. We find that a total of 397 packages depend on one or more of the identified 348 identical clones, and a total of 6,496 packages depend on one or more of the identified 5,944 close clones. Upon further examination, we find that 160 out of the 397 identical clone dependents are packages that are additionally developed by the same maintainers as the identical clone,

Table 5	**Clone Vulnerability & Popularity**	
Clone Type	**Identical (348)**	**Close (5,944)**
Likely Downloaded	21	399
More Vulnerable	62	2,304
Likely Downloaded AND More Vulnerable	4	148
More Vulnerable AND Vulnerabilities Undetected by Audit	17	190
Likely Downloaded AND More Vulnerable AND Vulnerabilities Undetected by Audit	0	8

Table 5: Measured popularity and vulnerability statistics of identical clones and close clones

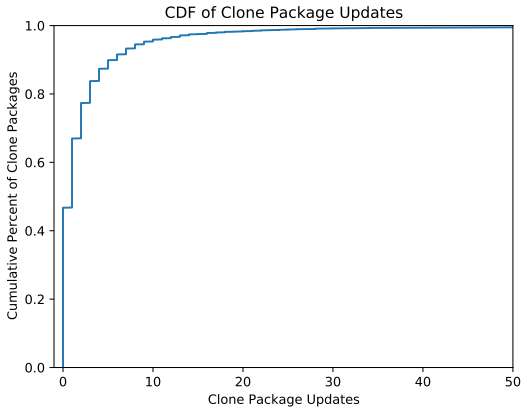

Figure 7: Distribution of total package updates over all identified clone packages

and 2,588 out of the 6,496 close clone dependents are packages that are also developed by the same maintainers as the close clone. This tendency for package maintainers to utilize their own shrinkwrapped clone packages as dependencies in additional packages that they develop could suggest one reason as to why package maintainers create clone packages–they intend to use their functionality for development purposes. Due to the existence of a large quantity of shrinkwrapped clone dependents, the total impact of clone packages is increased, since clone packages are installed unknowingly whenever a user installs a package that depends on a clone.

Clone Maintenance: Packages that are rarely or never maintained pose issues of health to the npm package ecosystem because it is less likely that their bugs and security vulnerabilities are ever addressed. We examine the update history of clone packages to determine if these packages are well-maintained or published yet forgotten. Figure 7 illustrates the distribution of total package updates across all identified clone packages. We find that 209 out of the identified 348 identical clones and 2,744 out of the identified 5,944 close clones have never received a single package update. This lack of maintenance poses a serious threat to the health of the npm registry.

Latent Vulnerabilities in Clones: The greatest danger that shrinkwrapped clones pose is that they can contain old vulnerabilities with known exploits that used to be present in the packages that they clone. We analyze the extent to which clones contain latent vulnerabilities by scanning the npm advisory database for vulnerabilities in the clone's dependencies, in the relevant version of the cloned package, and in the clone itself. We find that 62 out of the identified 348 identical clones and 2,304 out of the identified 5,944 close clones contain vulnerabilities that are not present in the most up-to-date version of the cloned package. Most of the identified vulnerabilities are located in outdated versions of clone dependencies, and these kinds of vulnerabilities could be resolved *if* a user executes an npm audit to fix known vulnerabilities within their installed packages. However, vulnerabilities that exist directly in the code of a cloned package version will also exist in the code of the corresponding clone, and even an npm audit cannot detect these vulnerabilities since the npm frontend lacks awareness of clone relationships. We find that 17 out of the 62 more vulnerable identical clones and 190 out of the 2,304 more vulnerable close clones potentially contain these kinds of latent vulnerabilities that are undetectable by standard auditing procedure.

Because it is possible that a close clone package maintainer could independently discover and fix a vulnerability copied over from a cloned package, we randomly sample 20 of the 190 close clones containing potentially unreported vulnerabilities, and we manually verify whether the vulnerability found in the cloned package is still present in the clone package. We find that in 18 cases, the vulnerability is present in the clone and undetected by an npm audit, and we find that in 2 cases, the package identified as a clone was falsely identified as a clone. Hence, these previously undetected vulnerabilities are quite prevalent, and even a security-conscious npm user cannot fix them with the auditing tools provided by npm.

We also examine the intersection between clone package popularity and clone package vulnerabilities to determine whether real users are likely impacted by these vulnerabilities. These intersecting subsets of identified clones are described in Table 5. We find that 4 out of the 21 identical clones with more than 350 weekly downloads and 148 out of the 399 close clones with more than 350 weekly downloads contain latent vulnerabilities not present in the most up-to-date version of the cloned package. We further discovered that 8 of the 148 vulnerable close clones likely downloaded by real users also contain vulnerabilities that are not detected by an npm audit, and one of these 8 clones, `@evocateur/npm-registry-fetch`, has more than one million weekly downloads. As such, we conclude that clone vulnerabilities pose an imminent threat to the npm package ecosystem because they are exposing real

users to known exploits of old vulnerabilities and are evading reasonable detection by security-conscious users.

Clone Repositories: In npm, package metadata often contains a repository URL pointing to where the package code resides, typically in the form of a GitHub URL. We analyze whether identified clones copy the repository URL of the package that they clone to determine if repository URLs could serve as a sufficient signal in detecting shrinkwrapped clones. The breakdown of repository URLs across identified clone packages is detailed in Table 4.2. Because identical clones can define their own distinct metadata, we find that out of the identified 348 identical clones, 210 copy the repository URL of the cloned package, 121 have a unique repository URL, and 17 do not provide a repository URL. Out of the 5,944 identified close clones, we find that 3,153 copy the repository URL of the cloned package, 2,602 have a unique repository URL, and 189 do not provide a repository URL. We compare these URL ratios to the npm registry as a whole, where 65% of packages provide a repository URL and 35% of packages do not. Hence, package repository URLs can provide some good insight into validating package clone relationships, but they are not sufficient as a clone detection signal.

5 DISCUSSION

5.1 Disadvantages and Benefits of Clones

The existence of identical clones offers no benefit to the npm ecosystem since npm already allows shrinkwrapping dependencies, where specific versions of packages can be specified as dependencies. Identical clones simply introduce more potentially vulnerable and less-documented packages to npm, and in particular, identical clones offer no functional benefit over using a shrinkwrapped version of the cloned package.

The existence of close clones offers some benefit to developers requiring modified packages, but npm lacks direct support for this, such as through forking packages. This leads to clones on npm being poorly documented, prone to latent vulnerabilities, and lacking maintenance.

In both cases, having a technique which can identify clones can offer benefits. Creators of identical clones can be redirected to use the npm shrinkwrap feature instead. Clones with functional differences can be explicitly linked to their original packages so that an auditing team discovering a vulnerability in one package can check whether the same exists in the other.

5.2 Future Work

While this work focuses on the detection and classification of shrinkwrapped clones, we identify several directions for future work, most notably in mitigating the dangers that shrinkwrapped clones pose.

Registry Integration of Forking. We note that registry support for forking packages and tracking fork relationships would help to mitigate many of the ecosystem health concerns introduced by shrinkwrapped clones. With explicitly labeled fork relationships, package confusability would be reduced, the update history and authorship of fork packages would be transparent, and vulnerabilities discovered in forked packages would also be detectable in fork packages.

Forking integration could be implemented directly into npm with the assistance of registry maintainers, or it could be implemented independently in a mirror of the npm registry. The greatest challenge posed in accomplishing this integration is in manually retrofitting existing clone packages as official forks. We believe that our clone detection pipeline, UNWRAPPER, would aid greatly in this process, although it is still a challenging problem of scale to tackle given the total size of the npm package registry.

Clone Update Patching. We consider the implementation of patching tools capable of applying security critical updates to clone packages to be another worthwhile direction for future work. With knowledge of clone relationships and critical updates to cloned packages, code patches for old vulnerabilities and bugs are identifiable. These patches can then be applied to clone packages, thus resolving critical issues that have previously been fixed in their corresponding cloned packages.

The greatest challenge posed by this approach is that programatically altering code, especially in highly dynamic programming languages such as JavaScript, is notoriously difficult. Although many patches applied to packages are very straightforward in nature, a patching tool will likely encounter many patches that are difficult or impossible to programatically resolve. As such, we leave navigating this challenge as an avenue for future work.

6 RELATED WORK

Detection of Code Clones: There is a significant body of work on the detection of code clones, i.e., instances where a software is duplicated without maintaining clear attribution of the original source code. Early efforts were based on extraction of lexical features as fingerprints. Moss [37], by Schleimer et al., is based on a winnowing algorithm to select small sections of source code, which are then used as fingerprints. Many similar approaches utilize n-grams to extract fingerprints. Smith and Horwitz [38] propose a clone detection algorithm based on the identification of least-frequent n-grams. Kim et al. [25] use similarities in locality-disjoint n-grams for clone detection. Ishio et al. [21] propose a clone detection approach based on identification of minimum-valued sets of n-gram hashes. NIL [29], by Nakagawa et al., utilizes the longest common subsequence of n-grams to detect clones with extensive modifications.

Deckard [22], by Jiang et al., uses a novel AST clustering algorithm for identification of similar code. CLEVER [30], by Nayrolles and Hamou-Lhadj, detects clones based on similarities within code blocks modified by commits in version control systems. For a review of other similar works, we refer the reader to Merlo et al. [9]. More recent work investigates the application of modern machine learning techniques to the problem [42, 45].

Many of the above techniques focus on clones that insert and delete statements of code, yet within the domain of npm we observe significant clones that exhibit complex syntactic modifications that lie outside the models of existing clone

detectors. In our work, we eschew code-based features for metadata analysis due to these domain-specific challenges and performance requirements in order to operate at scale. Efficiently and effectively integrating the techniques above for clone detection is an interesting direction for future work.

Provenance Inference: Our work specifically attempts to address which package has been cloned, as opposed to simply classifying a package as a clone. In this regard, our work has some similarities to previous works that attempt to infer the provenance of code. A related problem in provenance inference is that of authorship attribution [23]: mapping a software sample, either in binary [7, 12] or source code [10], to the developer who created it. Our work explicitly considers the effect of similarity on the security and stability of the language-based ecosystem in which the clone appears. Furthermore, authorship information alone is not sufficient to detect clones, since an author may create numerous legitimate packages, and many packages are the result of many distinct contributors.

Characterization of Package Repositories: Previous research has investigated the structure and evolution of various package repositories [19, 36, 43]. However, these works do not specifically analyze or discuss their security. Unlike these past works, our work seeks to characterize and address a specific security-relevant phenomenon present in the npm package repository (which we believe generalizes to other repositories as well).

Security and Stability of Package Repositories: A number of previously published works have investigated the effect of dependencies on the security and stability of software stored in package repositories, chiefly npm [6, 14, 16, 20, 26, 34, 41, 46]. Other package managers have also been an object of study [8, 13, 28]. More generally, poorly vetted dependencies represent an example of supply-chain security issue, a topic that has been discussed extensively [11, 27, 35, 40, 44] and has recently received renewed attention [1]. Finally, some recently proposed techniques aim at directly identifying malicious packages via code and/or metadata analysis [17, 18, 39]. Rather than analyzing existing dependencies and vulnerabilities, our work focuses on identifying hidden relationships—and thus potential security issues—between packages. To our knowledge, our work is the first to identify the shrinkwrapped clone phenomenon as a cause of security and stability issues.

7 CONCLUSION

The hygiene of package repositories is an important concern for the usability of language ecosystems. Programmers rely on the ecosystem to discern which packages are appropriate for their requirements, to deliver patches to package code, and to communicate vulnerabilities or bugs. Package maintainers rely on the ecosystem to credit their contributions appropriately. Maintaining package provenance is key to these capabilities.

In this work, we describe a phenomenon that we call shrinkwrapped clones. This phenomenon threatens the hygiene of the repository by obscuring the provenance of individual packages, weakening the security and usability of the entire language ecosystem. We analyze npm, and show that shrinkwrapped clones are observable. We show the harms of shrinkwrapped clones by reporting on instances that we found of clone packages that present older, vulnerable versions of other packages. Furthermore, we discover cases in which obscuring the provenance of these packages limits the reporting of security flaws and the deployment of patches in practice.

To address these issues, we have developed an analysis that is capable of detecting shrinkwrapped clones and restoring the provenance of a package. We implemented our analysis in a prototype tool we call UNWRAPPER, which is capable of operating efficiently, in real-time at the scale of the entire npm repository. Tools such as UNWRAPPER are a step towards mitigating the threat of shrinkwrapped clones and improving the hygiene of language ecosystems.

8 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful feedback that greatly aided us in improving this work. This work was partially supported by a generous gift from the Google Open Source Security Team.

REFERENCES

[1] 2021. Executive Order on Improving the Nation's Cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.
[2] 2021. Fork a Repo. https://docs.github.com/en/get-started/quickstart/fork-a-repo
[3] 2021. Grafeas Kritis. https://github.com/grafeas/kritis
[4] 2021. OSSF Package Feeds. https://github.com/ossf/package-feeds
[5] 2022. Open Science Framework. https://osf.io
[6] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on Npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 385–395.
[7] Saed Alrabaee, Paria Shirani, Lingyu Wang, Mourad Debbabi, and Aiman Hanna. 2018. On Leveraging Coding Habits for Effective Binary Authorship Attribution. In *ESORICS*.
[8] Anish Athalye, Rumen Hristov, Tran Nguyen, and Qui Nguyen. 2014. *Package Manager Security*. Technical Report. https://pdfs.semanticscholar.org/d398/d240e916079e418b77ebb4b3730d7e959b15.pdf
[9] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* 33, 9 (Sept. 2007), 577–591.
[10] Steven Burrows, Alexandra L. Uitdenbogerd, and Andrew Turpin. 2009. Application of Information Retrieval Techniques for Source Code Authorship Attribution. In *DASFAA*.
[11] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *SANER*.
[12] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2018. When Coding Style Survives Compilation: De-Anonymizing Programmers from Executable Binaries. In *NDSS*.
[13] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. 2008. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security*. 565–574.
[14] Kyriakos Chatzidimitriou, Michail Papamichail, Themistoklis Diamantopoulos, Michail Tsapanos, and Andreas Symeonidis. 2018. Npm-miner: An infrastructure for measuring the quality of the npm registry. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 42–45.
[15] Erik DeBill. 2021. Modulecounts. http://www.modulecounts.com/
[16] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network.

In *Proceedings of the 15th International Conference on Mining Software Repositories*. 181–191.

[17] 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. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium*. Internet Society.

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

[19] Daniel M German, Bram Adams, and Ahmed E Hassan. 2013. The evolution of the R software ecosystem. In *CSMR*.

[20] Joseph Hejderup. 2015. *In Dependencies We Trust: How vulnerable are dependencies in software modules?* Master's thesis. Delft University of Technology.

[21] Takashi Ishio, Yusuke Sakaguchi, Kaoru Ito, and Katsuro Inoue. 2017. Source File Set Search for Clone-and-Own Reuse Analysis. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 257–268. https://doi.org/10.1109/MSR.2017.19

[22] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *ICSE*. 96–105.

[23] Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina. 2019. Code Authorship Attribution: Methods and Challenges. *Comput. Surveys* 52, 1 (Feb. 2019), 1–36.

[24] Dulanka Karunasena. 2021. How I Analyzed All NPM Dependency Licenses in One Go. https://blog.bitsrc.io/how-i-analyzed-all-npm-dependency-licenses-in-one-go-18de0f7244bc

[25] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/1081706.1081737

[26] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 121–134. https://www.usenix.org/conference/raid2020/presentation/koishybayev

[27] R. G. Kula, C. D. Roover, D. German, T. Ishio, and K. Inoue. 2014. Visualizing the Evolution of Systems and Their Library Dependencies. In *IEEE VISSOFT*.

[28] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. 2016. Diplomat: Using delegations to protect community repositories. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 567–581.

[29] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. 2021. NIL: Large-Scale Detection of Large-Variance Clones. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 830–841. https://doi.org/10.1145/3468264.3468564

[30] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. CLEVER: Combining Code Metrics with Clone Detection for Just-in-Time Fault Prevention and Resolution in Large Industrial Projects. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 153–164. https://doi.org/10.1145/3196398.3196438

[31] NPM Blog Archive 2020. Npm Blog Archive: A Day in the Life of Npm Security. https://blog.npmjs.org/post/190665497245/a-day-in-the-life-of-npm-security.html

[33]]npm-registry-fetch-advisory npmjs.com. [n. d.]. npm advisory 1544. https://www.npmjs.com/advisories/1544.

[33]]npm-download-count npmjs.org. [n. d.]. numeric precision matters: how npm download counts work (accessed 02/2021). https://blog.npmjs.org/post/92574016600/numeric-precision-matters-how-npm-download-counts-work.

[34] Brian Pfretzschner and Lotfi ben Othmane. 2017. Identification of Dependency-based Attacks on Node.Js. In *ARES*.

[35] H. Plate, S. E. Ponta, and A. Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *ICSME*.

[36] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2013. The maven repository dataset of metrics, changes, and dependencies. In *MSR*.

[37] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *SIGMOD*. 10.

[38] Randy Smith and Susan Horwitz. 2009. Detecting and Measuring Similarity in Code Clones. In *IWSC*. 7.

[39] 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.

[40] Jørgen Tellnes. 2013. *Dependencies: No Software is an Island.* Master's thesis. The University of Bergen.

[41] Ruturaj K. Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. 2019. Security Issues in Language-based Sofware Ecosystems. *CoRR* abs/1903.02613 (2019). arXiv:1903.02613 http://arxiv.org/abs/1903.02613

[42] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *IEEE SANER*. 261–271.

[43] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *MSR*.

[44] A. A. Younis, Y. K. Malaiya, and I. Ray. 2014. Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability. In *HASE*.

[45] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural Detection of Semantic Code Clones Via Tree-Based Convolution. In *IEEE/ACM ICPC*. 70–80.

[46] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 995–1010.