On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem

Article · September 2017

4 authors, including:

Raula Gaikovina Kula
Nara Institute of Science and Technology
77 PUBLICATIONS   **694** CITATIONS

SEE PROFILE

Ali Ouni
Osaka University
71 PUBLICATIONS   **1,148** CITATIONS

SEE PROFILE

Katsuro Inoue
Osaka University
364 PUBLICATIONS   **4,991** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project   Collecting, Analyzing, and Evaluating Software Assets for Effective Reuse View project

Project   Micro Process Analysis View project

On the Impact of Micro-Packages: An Empirical Study of the `npm` JavaScript Ecosystem

Raula Gaikovina Kula*, Ali Ouni*, Daniel M. German‡ and Katsuro Inoue*

*Osaka University, Japan
{raula-k, ali, inoue}@ist.osaka-u.ac.jp

‡ University of Victoria, Canada
dmg@turingmachine.org

Abstract—The rise of user-contributed Open Source Software (OSS) ecosystems demonstrate their prevalence in the software engineering discipline. Libraries work together by depending on each other across the ecosystem. From these ecosystems emerges a minimized library called a micro-package. Micro-packages become problematic when breaks in a critical ecosystem dependency ripples its effects to unsuspecting users. In this paper, we investigate the impact of micro-packages in the `npm` JavaScript ecosystem. Specifically, we conducted an empirical investigation with 169,964 JavaScript `npm` packages to understand (i) the widespread phenomena of micro-packages, (ii) the size dependencies inherited by a micro-package and (iii) the developer usage cost (ie., fetch, install, load times) of using a micro-package. Results of the study find that micro-packages form a significant portion of the `npm` ecosystem. Apart from the ease of readability and comprehension, we show that some micro-packages have long dependency chains and incur just as much usage costs as other npm packages. We envision that this work motivates the need for developers to be aware of how sensitive their third-party dependencies are to critical changes in the software ecosystem.

Keywords-third-party libraries; JavaScript; ecosystems

I. INTRODUCTION

User-contributed Open Source Software (OSS) ecosystems have become prevalent in the software engineering discipline, capturing the attention of both practitioners and researchers alike. In recent times, *'collections of third-party software'* ecosystems such as the node package manager (`npm`) JavaScript [7] package ecosystem fosters development for huge amounts of server-side NodeJs and client-side JavaScript applications. According to a study in 2016 [31], the `npm` ecosystem for the NodeJs platform hosts over 230 thousand packages with *'hundreds of millions package installations every week'*.

Jansen *et al.* [16] states that ecosystems emerge when *'large computation tasks [are] split up and shared by collection of small, nearly independent, specialized units depending on each other'*. Based on this concept, we conjecture that third-party software ecosystems like `npm` encourage the philosophy of specialized software within these self-organizing ecosystems. A *micro-package* is the result of when a package becomes *'minimalist'* in its size and performs a single task [21]. For instance, the `negative-zero`[1] package has the trivial task of determining whether or not an input number has a negative-zero value. Micro-packages function as a single

---

[1] https://www.npmjs.com/package/negative-zero

unit by forming *'transitive'* dependencies between dependent packages (i.e., dependency chains) across the ecosystem.

We conjecture that an influx of micro-packages will result in an ecosystem that becomes fragile to any critical dependency changes. This is where breaking one critical dependency in the ecosystem will ripple its effect down the dependency chain to all dependent packages. For example, a breakage by removal of a tiny package called `left-pad` in March 2016 caused waves of dependency breakages throughout the `npm` ecosystem when it *'broke the internet'* of web applications [8]. Although `left-pad` is tasked with simply *'adding left space padding to a html page'*, it was heavily relied upon in the ecosystem by thousands of unaware packages and applications, including the core `Babel` compiler and `Node` JavaScript environment.

In this paper, we investigate the impact of micro-packages within the `npm` JavaScript ecosystem. Specifically, we conducted an empirical investigation with 169,964 `npm` packages to understand (*RQ1:*) the spread of micro-packages across the `npm` ecosystem. We then investigate micro-package usage implications such as the (*RQ2:*) the size of its complex dependency chains and (*RQ3:*) the amount of developer usage costs (ie., fetch, install, load times) incurred by using a micro-package. Findings show that micro-packages account for a significant portion of the ecosystem, with some micro-packages having just as long dependency chain lengths and reachability to other packages in the `npm` ecosystem. Furthermore, our findings indicate that micro-packages have no statistical differences in developer usage costs to the rest of the `npm` packages. We envision that this work motivates the notion of micro-packages and how to increase their resilience to changes in the ecosystem. The study concludes that developers should be aware of how sensitive their third-party dependencies are to critical changes in its ecosystem.

The main contributions of this paper are three-fold and can be summarized as follows: (i) we quantitatively study the phenomena of micro-packages and their impact in a software ecosystem, (ii) we motivate the need to revisit best practices to increase software resilience to their ecosystem changes and (iii) we motivate the need for developers to be aware of how sensitive their third-party dependencies are to changes in a software ecosystem.

II. BASIC CONCEPTS & DEFINITIONS

In this section, we introduce our definition of the different types of `npm` micro-packages and motivate how it can be problematic for developers that use these packages. We then introduce and define two types of micro-packages that we believe create excessive dependencies.

A. Micro-Packages and a Fragile Ecosystem

We define micro-packages as having a minimalist modular design through encapsulating and delegating complex tasks to other packages within the same ecosystem. Encapsulating complexity through abstraction or information hiding to ease code comprehension and promotes code reuse. Parnas [23] first proposed concepts of information hiding through the formation of modular structures. Baldwin and Clark later [10] propose modularity theory, when elements of a design become split up and assigned to modules according to a formal architecture or plan. In this setting, *'some of these modules remain hidden, while other modules are visible and embody design rules that hidden modules must obey if they are to work together'*.

We conjecture that an influx of micro-packages will result in a fragile ecosystem. As packages evolve, so do their dependencies within this ecosystem. Sometimes these changing dependencies breaks a critical dependency that will impact and ripple changes throughout the ecosystem. Mens *et al.* [21] reports experiences from both practitioners and researchers where managing the complexities of dependencies (i.e., colloquially known as *Dependency Hell*) leads to issues that ripple throughout the ecosystem. Our hypothesis is that a micro-package contributes to a fragile ecosystem by creating excessive dependencies across the ecosystem.

B. Package Interoperability in the `npm` Ecosystem

The JavaScript language adheres to certain specifications[2] that state how a package should be written in order to be *'interoperable among a class of module systems that can be both client and server side, secure or insecure, implemented today or supported by future systems with syntax extensions'*.

For the *npm* ecosystem, according the `npmjs`[3] documentation and `CommonsJS` Group[4] specify that a `npm` package should at least have the following requirements:

- a `package.json` configuration file.
- a JavaScript module.
- an accessible url that contains a gzipped tarball of `package.json` and the module.

A `npm` module refers to any JavaScript file that can be loaded using the `require()` command. Typically, the `package.json` points to an `entry point file`. This entry point file is used to load any modules (i.e., both local and foreign) required by the package.

[2]CommonJS Group has with a goal of building up the JavaScript ecosystem with specifications. Documentation at http://wiki.commonjs.org/wiki/Modules/1.0

[3]Official Website of the npmjs repository at https://docs.npmjs.com/how-npm-works/packages

[4]http://wiki.commonjs.org/wiki/Packages/1.0

Importantly, the JavaScript `require()` function is unable to differentiate between a module or package. Therefore, packages can load internal or external modules/packages that are installed on the `NodeJs` platform environment. For external dependencies, the `require()` function enables interoperability between all declared global functions to be accessible as the Application Programming Interface (API).

C. Micro-packages in the `npm` ecosystem

We now introduce two types of `npm` micro-packages. We first define two types of `npm` micro-packages that (i) perform trivial tasks and (ii) acts as a facade to load foreign module (i.e., third-party) dependencies.

```
1  'use strict';
2  module.exports = function (x) {
3    return x === 0 && 1 / x === -Infinity;
4  };
```

Listing 1. Entry point source code of a the `negative-zero` library to `Check if a number is negative zero`.

Listing 1 shows an example of a micro-package that performs a trivial task. In only four lines of code, the `negative-zero` package has the sole task of determining whether an input number has a value of negative zero. Importantly, the definition of a negative-zero is not prone to change so it is safe to assume that this package is less likely to be evolved in the near future.

```
1  'use strict';
2  module.exports = require('os-homedir')();
```

Listing 2. The `user-home` package entry point source code. As shown the library depends on the `os-homedir` library.

Listing 2 is example of a micro-package that forms a facade to another foreign package. As shown in the listing, `user-home` package is comprised of only two lines of code, but is tasked with determining the user-home folder for any operating system. Furthermore, the listing shows (Line 2) the `user-home` library dependent and loading the `os-homedir` package (shown in Listing 3) to perform complexities of the task.

```
1  'use strict';
2  var os = require('os');
3
4  function homedir() {
5    var env = process.env;
6    var home = env.HOME;
7    var user = env.LOGNAME || env.USER || env.LNAME ||
       env.USERNAME;
8
9    if (process.platform === 'win32') {
10     return env.USERPROFILE || env.HOMEDRIVE + env.
        HOMEPATH || home || null;
11   }
12
13   if (process.platform === 'darwin') {
14     return home || (user ? '/Users/' + user : null);
15   }
16
17   if (process.platform === 'linux') {
18     return home || (process.getuid() === 0 ? '/root'
        : (user ? '/home/' + user : null));
19   }
20
21   return home || null;
```

```
22    }
23    module.exports = typeof os.homedir === 'function'
        ? os.homedir : homedir;
```

Listing 3. `os-homedir` package entry point source code. It contains one function `homedir`.

Listing 3 shows hidden `os-homedir` package that performs the computation for the `user-home` package. Importantly, the `user-home` package is prone to evolutionary dependency changes as operating systems (see Lines 9, 13, 17) may change their platform configurations in the future. In this case, any application that uses package with a dependency on the `user-home` is unaware that it is indirectly impacted by any evolutionary API changes or removal of the `user-home` package.

We leverage the package size attributes of (i) global function APIs and (ii) physical size to propose a method to identify the two types of micro-package defined in this study. In detail, we propose and rationalize these package size metrics:

- *# of FuncCount (FuncCount)* - is a count of API functions implemented in a package (i.e., function in the `entry point file`). Our rationale is a single function is more likely a trivial micro-package, while a facade micro-package is likely to use the `require()` function to directly load its dependencies.
- *# of Lines of Code (SLoC)* - is the count of source code lines in a package (i.e., function in the `entry point file`). We consider both the physical (i.e., physical includes code comments and spacing) and logical lines of code for a library. Our rationale is that trivial or facade micro-packages tend to have smaller sLoC.

We can now use these metrics to rationalize our definition of a micro-package. Formally, for any given Package P, P is a micro-package when $funcCount(P) \leq 1$.

III. EMPIRICAL STUDY

Our main goal is to better understand the role and impact of micro-packages in the `npm` ecosystem. We conducted an empirical investigation with three guiding research questions regarding micro-packages.

RQ1 relates to how widespread the phenomena of micro-packages exists in the `npm` ecosystem. In this research question analyzes the size of `npm` packages to determine whether micro-packages form a significant portion of the ecosystem.

RQ2 and RQ3 relates the evaluation of some perceptions on when using a micro-package. From a complexity of dependencies viewpoint, RQ2 investigate the size of dependency chains developers incur when using a micro-package. For RQ3, we then examine fetch, install and load usage costs incurred by a web developer when using a micro-package as one of its dependencies.

A. *RQ1*: **What is the size of a `npm` package?**

1) Motivation: Our motivation for RQ1 is to understand how widespread the micro-package phenomena is in the ecosystem. In this research question, we analyze `npm` package size attributes to survey whether or not micro-packages form a significant portion of packages in the ecosystem.

Table I
DATASET COLLECTED NPM CORPUS FOR RQ1

	Dataset statistics
Snapshot of `npm` ecosystem	July-1st-2016 ~ July-15th-2016
# downloads	186,507 libraries
Size (GB)	200GB
# js entry point files analyzed	169,964 files

2) Research Method: Our research method to answer RQ1 is by statistical analysis and manual validation. It follows a two step of data collection and then analysis of the results. In the first step, we collect and process `npm` libraries to extract their size attributes (i.e., FuncCount and SLoC) for each package.

For the analysis step, we describe the npm size metrics distribution using statistical metrics (i.e., Min., Mean (μ), Median ($\bar{x}$), Max.) and plot on a histogram. We then manually investigate and pull out examples of the results to qualitatively explain the reasoning behind our conclusions. Finally, we identify how many npm packages fit our definition of micro-packages. We then utilize boxplots plots with the sLoC metric to compare any statistical differences between micro-packages to the rest of the packages.

3) Dataset: Similar to the collection method of [31], we use the npm registry[5] to procure an offline copy of the publicly available packages *npm* ecosystem.

Table I presents statistics of the final collected dataset after a quality pre-processing was performed to remove invalid or erroneous `package.json` metafiles[6]. As shown in the table, after downloading 186,507 `npm` packages, 169,964 packages remain for our analysis. Since `npm` ecosystem policies allows unrestricted access, pre-processing was a quality filter to remove invalid packages.

```
1  {
2    "name": "user-home",
3    "version": "2.0.0",
4    "description": "Get path to user home directory",
5    "license": "MIT",
6    "repository": "sindresorhus/user-home",
7    "author": {
8    "name": "Sindre Sorhus",
9    "email": "sindresorhus@gmail.com",
10   "url": "sindresorhus.com"
11   },
12   ....
13   },
14   "main": [
15   "index.js"
16   ],
17   ....
18   "dependencies": {
19   "os-homedir": "^1.0.0"
20   },
21   ...
22  }
```

Listing 4. Snippet from the *package.json*. Note that lines 14 and 15 correspond to the entry point file that is accessible for a client user.

---

[5]https://registry.npmjs.org/-/all

[6]A blog on the official npmjs website details some of the perils of the npm front-end packaging at http://blog.npmjs.org/post/101775448305/npm-and-front-end-packaging
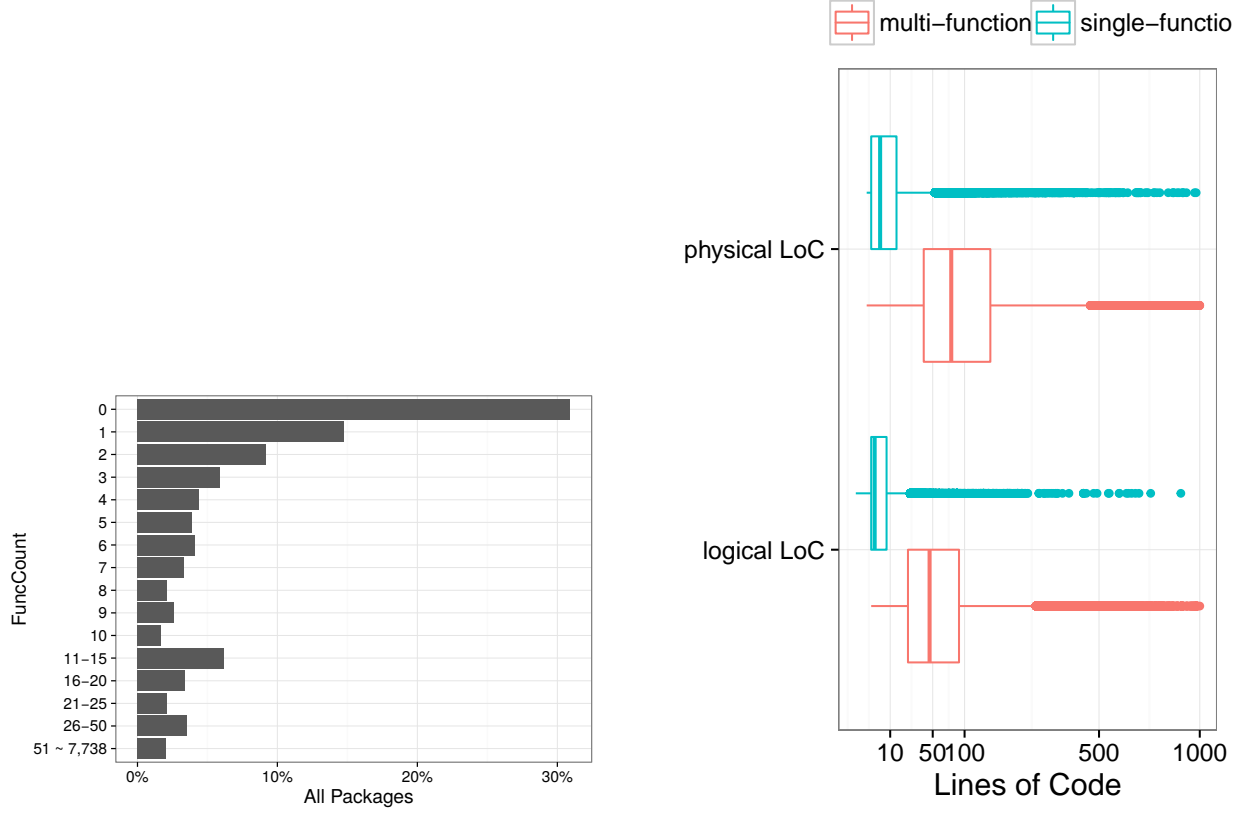
(a) JavaScript libraries grouped by FuncCount

(b) Comparing Lines of Code (sLoC) between micro-packages and other packages.

Figure 1.   Results for RQ1 package size metrics (a) FuncCount and (b) sLoC

TABLE II
LIBRARY DESIGN SIZE METRIC STATISTICS FOR RQ1

	Min.	Median ($\bar{x}$)	Mean (μ)	Max.
FuncCount	0.00	2.00	8.65	7,738.00
physical SLoC	1.00	29.00	112.30	137,700.00
logical SLoC	0.00	17.00	63.63	129,100.00

Listing 4 depicts the `package.json` metadata file used to identify and extract the `entry point file`. This entry point will determine the available API of a package. From this metafile, we extract the main field (Lines 14 and 15), which identifies the entry point. Note that the `name` attribute is a unique identifier for any library. We then use the `esprima` npm package [1] to extract our package size metrics. `esprima` is a high performance and highly popular[7] JavaScript analysis tool that can construct a JavaScript syntax tree and loads all related class files.

4) Findings: Table II and Figure 1 report the results of RQ1. Table II shows the summary of statistics our package size metrics, while the figure is a visual distribution of the funcCount and sLoC metrics.

From a topological viewpoint, we observe from Table II that

libraries have a median of two functions per library. The result shows that most `npm` packages contain a small number of API functions available from the entry point file. Furthermore, we observe that the median size of the source code is between 17 loc (logical) and 29 loc (physical). Looking at the largest package, we find that the largest in terms of both API and lines of code is the (`tesseract.js-core`)[8] package. This package contains 5,951 functions and 137,700 lines of code.

A key observation from Figure 1(a) is that micro-packages account for up to 47% of all `npm` libraries. We can observe from the figure that 32% of all packages are without function (i.e., $FuncCount = 0$). Our manual validation shows that many of these packages belong to the `facade` micro-package type. However, some of these micro-packages are used to only local modules so are not bridging APIs to other external packages. For example, the `12env` package contains only the single line:

```
1 module.exports = require('./lib/config')();
```

that is tasked to load the internal `config.js` module that performs the computation of this package. Further analysis

---

[7]as of Feb 10, recorded over 17,700,000 downloads from the npm

[8]Size of the code can be summarized by the description *'if you're a big fan of manual memory management and slow, blocking computation, then you're at the right place!'* https://github.com/naptha/tesseract.js-core

reveals `config.js` to have a single API function with 71 lines of code. Although the config module is part of the package, we conclude according to our micro-package definitions it still has an excessive interoperability dependency.

Figure 1(b) shows a statistical measure of the physical sizes of micro-packages compared to the rest of the npm packages. We observe that micro-packages are statistically smaller in size. We report micro-packages have a lower median of lines of code (physical ($\bar{x} = 5LoC$) and logical($\bar{x} = 3LoC$)) This result is not surprising and matches with our intuitions of micro-packages. From the figure, we also observe that not all micro-packages contain small code. For instance, the `icao`[9] package has 8,255 lines of code. Further analysis concludes that most of the code include hard-coded variable constants.

In summary for RQ1, we propose our package size metrics. To answer RQ1: *our findings show that micro-packages account for a significant 47% of all packages, with 32% acting as a facade to create excessive dependencies within the ecosystem. Interestingly, most* npm *packages are designed to be smaller in size, with up to 2 functions and 29 lines of code per package.*

B. *RQ2:* **What is the dependency chain size for a `npm` package?**

1) Motivation: Our motivation for RQ2 is to understand the extent of how micro-packages form dependency chains across the ecosystem. In this research question, we calculate the dependency chains of npm packages to survey whether or not micro-packages result in an increase of dependencies. Therefore, we investigate whether or not there is a difference in dependency complexity for micro-packages. RQ2 includes testing the hypothesis that *micro-packages include lesser dependency complexity than other packages in the npm ecosystem.*

2) Research Method: Our research method to answer RQ2 is by using network analysis[30] with statistical analysis of the network generated metrics and then manual validation to understand our results. It is performed in two steps. The first step involves the construction of npm network that models and represents dependency chains in the npm ecosystem.

The second step involves analysis of the generated npm graph network. To do this, we propose a set of dependency metrics derived from the network analysis to describe a npm package dependency complexity. We then utilize box-plots and violin plots with each dependency metric to depict the differences between micro-packages to the rest of the packages. Finally, in our analysis we test the hypothesis whether or not micro-package has greater or lesser dependency complexity that other packages. To assess the significance of dependency metric differences, we use the Wilcoxon–Mann–Whitney and Cohen's *d* test [27] as they do not require the assumption of normal distribution. The null hypothesis (H_0) states that

[9]is a library that looks up hard-coded International Civil Aviation Organization airport codes. The website is at https://github.com/KenanY/icao/blob/master/index.js
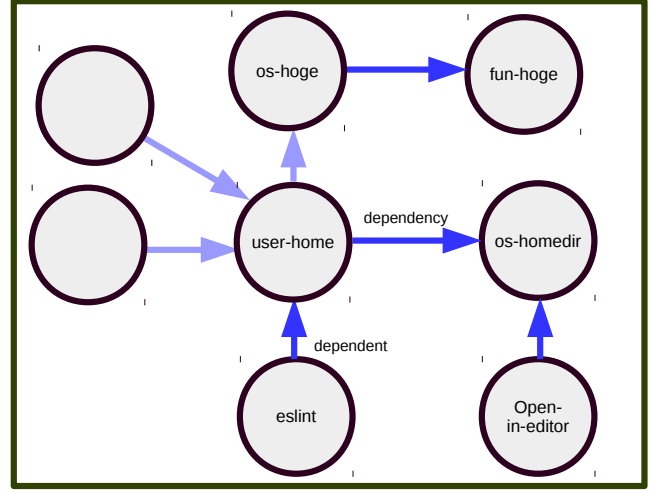

Figure 2. Modeling a chain of dependencies for the `eslint` package as a directed acyclic graph network. For better examples, we added `os-hoge` and `fun-hoge` libraries

either micro-packages include greater or lesser dependency complexity[10] If the null hypothesis is accepted, we then determine which population has the higher means to state our hypothesis. The alternate hypothesis (H_1) is that *micro-packages include identical dependency complexities to other packages.* Furthermore, to assess the difference magnitude, we studied the effect size based on Cohen's *d*. The effect size is considered: (1) small if $0.2 \leqslant d < 0.5$, (2) medium if $0.5 \leqslant d < 0.8$, or (3) large if $d \geqslant 0.8$.

3) Modeling the npm *ecosystem network:* Figure 2 serves as our graph-based model of dependencies in the npm ecosystem. In this model, we depict graph nodes as each a unique package and edges as directed dependencies. Formally, we define our npm ecosystem network as a directed graph $G = (V, E)$ where V is a set of nodes (=packages) and $E \subseteq V \times V$ is a set of edges (=dependencies). Since the graph is directed, E has an ordering: $\{u, v\} \neq \{v, u\}$. For instance, we represent the `eslint` package dependency on `user-home` package. We observe that the `eslint` package is transitively dependent on `os-homedir` through its dependence on the `user-home` package.

Furthermore, we define dependency metrics to describe the direct and transitive dependencies. We first introduce the following direct dependency metrics:

- *Dependents (Incoming Degree)* - is the number of packages that directly depend on a package (incoming dependencies). In figure 2 `user-home` has three dependent (i.e., includes the `eslint` package). Dependents is a useful measure of package reuse within the ecosystem.
- *Dependencies (Outgoing Degree)* - is the number of packages on which a package directly depends on (outgoing dependencies). In figure 2 `user-home` has two dependencies (i.e., `os-homedir` and `os-hoge`).

We then introduce our transitive dependency metrics:

- *Chain Length (Eccentricity score)* - is a normalized measure to evaluate the longest chain of dependencies for a library dependency. This metric counts the longest path for a given graph node to any of its reachable nodes. As shown in figure 2, `eslint` has a chain length of three (i.e., *eslint → user-home → os-hoge → fun-hoge*).
- *Reach Dependencies (# of reachable nodes)* - is the number of libraries that are within its reachable path. For library `eslint` has four reachable nodes (i.e., {`user-home`, `os-homedir`, `os-hoge` and `fun-hoge`}).

```
1  {
2    "name": "user-home",
3    "version": "2.0.0",
4    ....
5    "dependencies": {
6    "os-homedir": "^1.0.0"
7    },
8    ...
9  }
```

Listing 5. Snippet from the *package.json* of the `user-home` in RQ1 (See Section III-A2) that shows the header and dependency relation to `os-homedir` library.

As shown in Listing 5, we extract each dependency (i.e., Lines 5 and 6.) from the `package.json` metadata file to generate our npm graph network. For each package, we collect all runtime dependencies, development and optional dependencies.

Data: V is a package graph node E is a dependency edge, *packageList* lists all collected packages, *getDeps(package)* list all dependencies for a package lib ...
1 Initialize *packageList* with all packages;
2 Initialize npmGraph G;
3 **for** *each package in packageList* **do**
4   **if** *pacakge lib in G* **then**
5 | node $V_{lib} \leftarrow getNpmGraphNode(G, lib)$;
6   **else**
7 | create node $V_{lib} \leftarrow createNode(lib)$;
8 | add node V_{lib} to G;
9   **end**
10   dependencyList $\leftarrow$ getDeps(lib);
11   **for** *each library dep in dependencyList* **do**
12 | create node $V_{dep} \leftarrow createNode(dep)$;
13 | add node V_{dep} to G;
14 | create edge $E \leftarrow createEdge(V_{dep}, V_{lib})$;
15 | add edge E to G;
16   **end**
17 **end**

Algorithm 1: npm Ecosystem Network Algorithm

Algorithm 1 details the algorithm we use to construct and generate the `npm` graph network. The key idea in building the *npm* network is to append each package as a new graph node to existing packages in the network. In this sense, a node is a package with that is either dependent or depended upon by other package nodes in the network. So for each new package, we first determine if it exists in the graph (Step 4) as a depended node from another package. If its does not exist

Table III
SUMMARY STATISTICS OF THE NPM ECOSYSTEM NETWORK FOR RQ2

	npm ecosystem network
# nodes	169,964
# edges	986,075
Average Chain Length	5.89

Table IV
DEPENDENCY METRIC STATISTICS OF THE COLLECTED NPM FOR RQ2

	Min.	Median ($\bar{x}$)	Mean (μ)	Max.
Dependents	0.00	3.00	5.81	22,750.00
Dependencies	0.00	0.00	5.80	41,550.00
Chain Length	0.00	0.00	1.03	20.00
Reach Dependents	0.00	0.00	563.90	121,600.0

(Steps 7-8), we then proceed to create a new node for this package and then check its dependent whether they also exist on the graph (Steps 11-15).

Table III shows that our `npm` graph network comprises of 169,964 graph nodes and 986,075 graph edges with an average chain length of 5.89. We use python scripts to implement the algorithm and the Neo4j graph database to store the npm network (i.e., *py2neo* [6], the *neo4j* [6]. The *gephi* tool [2] was used to generate the dependents, dependencies and chain length dependency metrics. Furthermore, we used a Depth First Search (DFS) query in Neo4j to generate the Reach Dependencies metric. For a package node v with a package name $v.name$, we use the Neo4j cypher query[11]: `Match v{name:v.name} -[*..x]->(w) where NOT v-w RETURN v.name, count(DISTINCT(v))` to find all packages that are within its dependency reach in the ecosystem.

5) Findings: Table IV shows the summary statistics for the dependency metrics generated for the `npm` network. From a topological viewpoint, we observe from that most packages had a median of three dependents per with no dependencies. With a manual validation, we find that the `everything` package has the most dependents in the network. Interestingly, this package is labeled by many JS developers as a *'hoarder'*[12] library. With over 22,750 other incoming dependency relations from other libraries, this package offer no useful function but to access every other package on the network. Conversely, we observe the very useful `mocha`[13] package as the most depended upon package (i.e., Dependencies = 41,570). This testing framework validates the notion that `npm` library developers indeed run and perform test on their npm packages.

Figure 3 shows the statistical distribution for all direct dependency metrics across the `npm` ecosystem. The key observation from Figure 3(a) is that only 9% of all packages have no dependents (i.e., Dependents=0), inversely indicating that almost 91% of libraries are being dependent upon by another

---

[11]For realistic computation-time costs, we set a threshold of our reachable node to four chains of dependencies (x=5). As the average chain length is 5.89 it should have minimal effect on the result (See Table III)
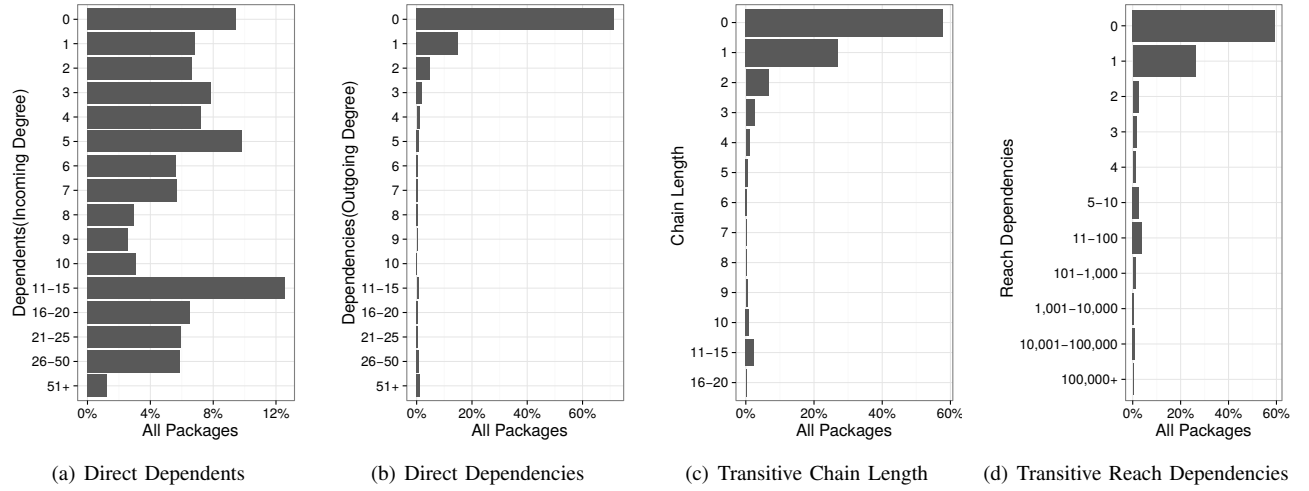
[12]blog discussing this at https://github.com/jfhbrook/hoarders/issues/2

[13]website at https://github.com/mochajs/mocha

(a) Direct Dependents   (b) Direct Dependencies   (c) Transitive Chain Length   (d) Transitive Reach Dependencies

Figure 3. Results for RQ2 direct and transitive dependency metrics for the `npm` ecosystem network

(a) Direct Dependents   (b) Direct Dependencies   (c) Transitive Chain Length   (d) Transitive Reach Dependencies
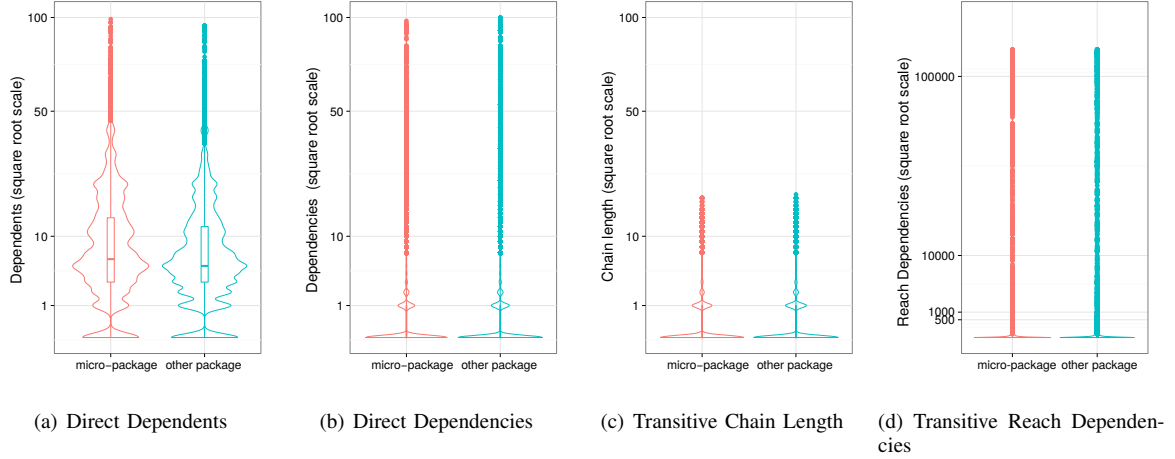
Figure 4. Micro-package dependency metric comparisons against the other packages in the `npm` ecosystem network.

package in the ecosystem. Findings also provide evidence that more than 12% of the libraries have about 11 to 15 dependents. Figure 3(b) confirms the results in Table IV, that clearly showing that 65% of packages in the npm ecosystem show no dependencies to other packages.

Figure 3(c) and Figure 3(d) we show through our transitive dependency metrics that around 60% of packages are not transitively reachable from another package. With a manual validation we find two packages that are critical to the `npm` ecosystem. The network analysis shows that `fs-walk`[14] having the longest transitive dependency chain length (i.e., Chain length = 20). This useful library is used for synchronous and asynchronous recursive directory browsing. Its long dependency chain provides evident that it is a critical package within the `npm` ecosystem as it is part of multiple dependency chains. We also report the very popular and useful AST-based pattern checker tool `eslint`[15] to be the most reachable by any other

package on the `npm` ecosystem.

Testing our hypothesis that *either micro-pacakage include greater or lesser dependency complexity* shows that dependency complexities of a micro-package is not so trivial, with some micro-packages have just as long chain lengths and reach dependencies as the rest of the packages. Although visually in Figure 4 there seems no transitive dependency differences between micro-packages and the rest of packages, results of the Wilcoxon–Mann–Whitney tests show otherwise. Table V show that a micro-package are more likely to be reused, indicated by statistically more dependents than the rest of the packages (i.e., Dependent accepted H_0 for micro-package). Furthermore, the results show micro-packages are more likely to reach other packages (i.e., Reach Dependencies accepted H_0 for micro-packages) in the `npm` ecosystem.

[14]website at https://github.com/confcompass/fs-walk
[15]https://www.npmjs.com/package/eslint

	H_0	$> \mu$	p-value	d
Dependents	accept	micro-package	< 0.01	0.09 (S)
Dependencies	reject	identical	< 0.13	0.13 (S)
Chain Length	accept	other packages	< 0.001	0.01 (S)
Reach Dep.	accept	micro-package	< 0.002	0.01 (S)

In summary for RQ2, we propose and model a *npm* dependency network for the `npm` ecosystem. To answer RQ2: *our findings depict micro-packages dependencies as not so trivial, with some micro-packages having just as long dependency chain lengths and reachability to other packages in the* `npm` *ecosystem.*

C. RQ3: What is the developer usage cost of a `npm` package?

1) Motivation: Our motivation for RQ3 is to understand the whether micro-packages provide any usage benefits in terms of server-side NodeJs performance for developer. In a 2015 developer blog[16], web developers express how the loading times affects the application performance when using JavaScript based modules on the NodeJs platform environment. According to the NodeJs documentation[17] NodeJs uses a complex searching algorithm to locate the requested source code each time the `require()` function is invoked. Therefore, we investigate whether there is a difference in the package loading times for micro-packages compared to the rest of the packages. RQ3 includes testing the hypothesis that *micro-packages incur developer usage costs of a micro library are no less than the rest of packages in the* `npm` *ecosystem.*

2) Research Method: Our research method to answer RQ3 is by using controlled experiment simulations with statistical analysis and manual validation to understand our results. It is performed in two steps. The first step comprises of candidates `npm` packages eligible for the experiment. In our preliminary experiments, encountered packages that some malicious packages caused disruptions to our experiment environment, particularly related to malicious libraries such as rimrafall[18]. We found other documented vulnerable threats to our experiments [19]. To mitigate this, two authors manually checked the package name for keywords that raise suspicions to dubious packages. After 8 hours of scanning the package lists, we were able to manually validate 20,000 packages based on sufficient popular downloads usage and a recognizable package name.

---

[16]The blog discusses how the `require()` function impacts testing time due to its slow loading into the environment: at https://kev.inburke.com/kevin/node-require-is-dog-slow/

[17]https://nodejs.org/api/modules.html#loading_from_node_modules_Folders

[18]details found at: https://github.com/joaojeronimo/rimrafall

[19]http://blog.npmjs.org/post/141702881055/package-install-scripts-vulnerability and a list at https://snyk.io/vuln?type=npm

For the analysis step, we describe the developer usage cost distribution using statistical metrics (i.e., Min., Mean (μ), Median ($\bar{x}$), Max.). We then manually investigate and pull out examples of the results to qualitatively explain the reasoning behind our conclusions. The second step involves analysis of the experiment results to test the hypothesis that incurred usage costs of a micro-package are no less than the rest of packages in the `npm` ecosystem. To assess the significance of dependency metric differences, we use the Wilcoxon–Mann–Whitney and Cohen's d test [27] as they do not require the assumption of normal distribution. The null hypothesis (H_0) states *either micro-packages incur either greater or lesser developer usage costs*[20]. The alternate hypothesis (H_1) is that *both micro-packages incur identical developer usage costs compared to the rest of the packages.* If the null hypothesis is rejected, we then conclude that micro-packages share identical usage costs the rest of the packages. Furthermore, to assess the difference magnitude, we studied the effect size based on Cohen's d. The effect size is considered: (1) small if $0.2 \leqslant d < 0.5$, (2) medium if $0.5 \leqslant d < 0.8$, or (3) large if $d \geqslant 0.8$.

Data: $packageList$ lists all collected packages,
1 **for** *each package in packageList* **do**
2   clean NodeJs env.← clean lib folder and clear cache;
3   Fetch Package ← url from npm repository;
4   Install Package ← into the local NodeJs;
5   Load Package ← use require () to load into local NodeJs;
6 **end**

Algorithm 2: Developer Usage Scenario for RQ3

3) Controlled Experiment: Algorithm 2 details the controlled experiment scenario that simulates the usage of an `npm` package. In sequential order, we show how a developer uses the standard protocols to: (i) fetch an npm package from the NPM ecosystem repositories (i.e., Line 3), (ii) install the package and finally (i.e., Line 4) (iii) load the package into the NodeJs environment using the `require()` function (i.e., Line 5). We reset the NodeJs to ensure a controlled environment each time a scenario is run (i.e., Line 2). To monitor the module loading execution time, we use the `require-profiler` npm package [9] to measure the execution time for NodeJs the modules loading.

Table VI details the technical specifications showing that

---

[20]We set the confidence limit α as 0.01

Table VI
CONTROLLED EXPERIMENT SPECIFICATIONS FOR RQ3

	Dataset statistics
Experiment Period	Sept-9th-2016 $\sim$ Sept-12th-2016
Download Speed	151.96 Mbps
Client Machine	Intel(R)Xeon(R), 8-core@3.60GHz
	32 GB RAM
Operating System	Ubuntu ver. 14. 04
server-side JS runtime env.	`NodeJs` ver. 4.2.6
Collected Packages	20,000 `npm` packages
Executed Packages	14,126 `npm` packages

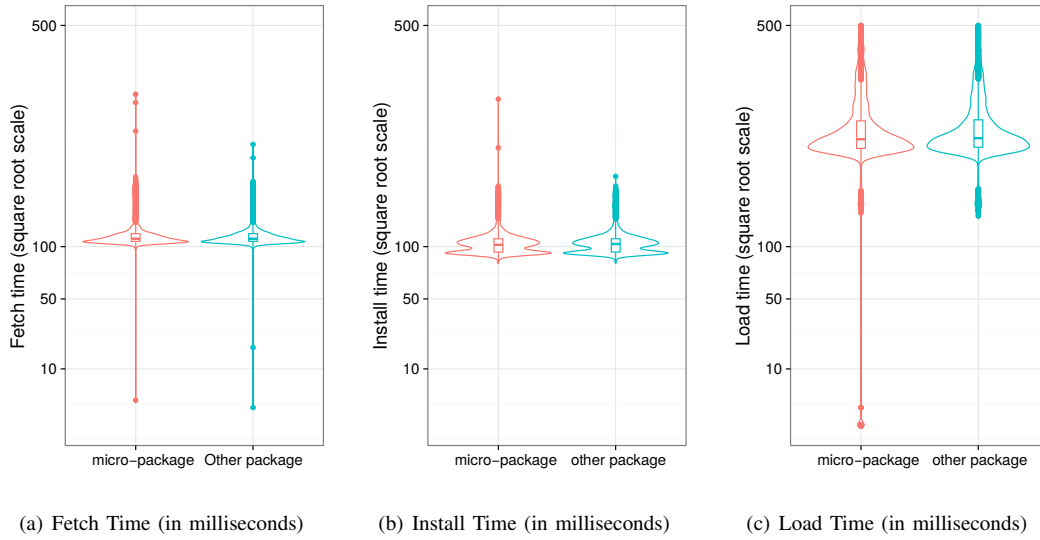(a) Fetch Time (in milliseconds)   (b) Install Time (in milliseconds)   (c) Load Time (in milliseconds)

Figure 5.   Micro-package developer usage cost metric comparisons against the rest of packages in the `npm` ecosystem.

Table VII
DEVELOPER USAGE COST STATISTICS OF THE COLLECTED NPM (IN MILLISECONDS)

	Min.	Median ($\bar{x}$)	Mean (μ)	Max.
Fetch time	1.00	109.00	113.80	623.00
Install time	82.00	103.00	105.20	1,619.00
Load time	0.00	242.00	322.30	328,600.00

Table VIII
STATISTICAL TESTS OF DEVELOPER USAGE COST METRICS. H_0 STATES THAT BOTH LIBRARY TYPES HAVE SIGNIFICANT DIFFERENCES.

	H_0	p-value	d
Fetch Time	reject	< 0.75	0.008 (S)
Install Time	reject	< 0.49	0.002 (S)
Load Time	reject	< 0.05	0.012 (S)

we were able to successfully install and load 14,126 packages from the 20,000 packages collected. The controlled experiment was conducted on a `z620 hp workstation` with eight core-processors running at `3.60Ghz` and `32GB` of RAM space, running `Ubuntu ver.16.04`, and `NodeJs ver.4.2.6`. Using internet speedtests[21], we recorded the internet speed to be at 151.96 MB per second. The most common reason for some failed runs was because dependencies were either missing or our NodeJs environment was unsuitable had parameters. Overall, the experiment took about 36 hours to execute all 14,126 scenario runs.

4) Findings: Table VII shows the summary statistics for the three developer usage cost metrics for `npm` packages. We observe from the table that took a median of 109ms to fetch, 105ms to install and 242ms to load a library. We find that developers are especially concerned when loading applications with large dependencies (i.e., *'loading around 10.000 files into the NodeJs environment'*)[22]. With a manual validation, we find that the slowest loading npm packages had dependencies to large JavaScript libraries. For instance, the `forkeys-benchmark`[23] package took the longest to load into the NodeJs environment. After reading documentation, we conclude that the reason for taking the longest

time is because its dependency to the JS benchmark library (`Benchmark.js` [4]. Another library that took a significantly long loading time was `gulp-transformers`[24], that has dependencies to the JavaScript `gulp` [5] build tool and the `browserfly` [3] JavaScript library that lets developers `require('modules')` in the browser by bundling up all of dependencies.

Testing our hypothesis that *micro-packages incur developer usage costs of a micro library are no less than the rest of packages in the `npm` ecosystem*, we find that micro-packages have no statistically differences in developer usage costs to the rest of the `npm` packages. Results in Figure 5 and Table VIII confirm acceptance of the alternative hypothesis that micro-packages share identical developer usage costs as other packages in the ecosystem.

In summary for RQ3, we propose and experiment to investigate developer usage costs for over 20,000 `npm` packages. To answer RQ3, *our findings indicate that micro-packages have no statistically differences in developer usage costs to the rest of the `npm` packages.*

---

[21]speedtest at www.beta.speedtest.net

[22]this discussion shows how developers discuss loading times of JS modules at https://groups.google.com/forum/#!topic/nodejs/52gksIpgX4Q

[23]website is at https://github.com/forkeys/forkeys-benchmark

[24]The package functions to unify gulp, browserify and its transforms once and for all at https://github.com/limdauto/gulp-transformers

IV. DISCUSSION

A. Implications and Challenges

Results of our empirical study contributes to the research performed on software ecosystems [13], [19] and their effect on software maintenance. The issue of micro-packages and how an ecosystem can ripple effects to an application has been studied in related work [21], [22]. However, ongoing research on these self-organizing ecosystems are still unclear on how and why ecosystems form, how they achieve their impact, or how they will continue to sustain themselves. Our work is working towards that goal of understanding how to cope with threats to an ecosystem.

The study brings to light many new challenges and avenues for potential future research. JavaScript best practices[25] recommends to modularize (i.e., limit one function to one task) for better code comprehension and reuse. However, a byproduct of package modularization is the influx of excessive dependencies in the ecosystem. As these ecosystems are self-organizing with no strict social structures, we believe future work is needed to understand how ecosystems to cope with increased dependency complexity over time. Future work include revisiting the best practices on how to make packages resilient to ecosystem changes whilst maintaining code modularity. One potential strategy that is employed in other ecosystems is to merge popular and useful libraries into the core platform package. For example, the popular `Joda-time` Java library was migrated into a core part of the Java 8 and onwards JDK (i.e., `java.time (JSR-310)`). In addition, we envision that support is needed to raise the awareness of developers to how sensitive their applications are to the ecosystem.

As future work, we would like to extent the notion of packages. Currently the `require()` function does not differentiate between internal and external modules. Therefore, further work is needed to explore the true notion of the dependency relationships of external modules. We are however confident that our definition of a micro-package is sufficient to detect this phenomena.

B. Threats to Validity

1) Construct: This validity is concerned with threats to our micro-package definitions. In this work, we simply use the number of functions to reason for a definition of a package, however, in reality the refined definition of a micro-package would be the smallest number of functions in the least amount of code. We are confident that our definitions are sufficient to capture excessive dependencies in the ecosystem and discuss how threat is will be treated as future work (i.e., Section IV-A).

2) Internal: This validity is related to the accuracy of the data collected and tools used in the experiments. One of the major threat is the accuracy of data that is contained in the `package.json` metafile configuration file for RQ2 and RQ3. There exists cases when a dependency is not listed in the metafile (i.e., `require()` call to unlisted dependency module). We understand this situation can rarely exist, as the

---

[25]https://www.w3.org/wiki/JavaScript_best_practices

---

vast majority of quality library would correctly list all dependencies. In future work, we plan to use more direct source-code dynamic analysis to mitigate this threat. Another potential threat can be related to the use of `esprima` [1] for the generation of code-metrics in RQ1 and `require-profiler` [9] to measure the developer usage cost metrics in RQ3. To our best knowledge, both these tools are heavily used in the npm ecosystem and are compliant to ECMAScript standards. `Esprima` runs on many popular web browsers and other ECMAScript platforms such as Rhino, Nashorn, and NodeJs

3) External: The external validity refers to the generalization of our results. We only collected a sample of the true size of JavaScript projects, that are open to the public. Although we believe our large dataset of over 169,000 libraries is a sufficient representation of npm JavaScript current practices, expanding to different ecosystems is a potential avenue for future work.

V. RELATED WORK

1) Software ecosystems: Recently, there has been an increase in research that formulate software systems as belonging to an ecosystem. Lungu best termed ecosystems as a *"collection of software projects which are developed and evolved together in the same environment"* [19]. There has been other definitions from a business perspective [15], [16] and also from a technological perspective [20]. In this work, we consider the ecosystem of libraries that belong to the same technological platform, in this case JavaScript. Related, Wittern *et al.* [31] studied the topology of popular npmJavaScript libraries, finding that npm packages are largely dependent on a core set of libraries. This is consistent with our result that package reuse is apparent in the ecosystem.

Related, Mens *et al* [22] perform ecological studies of open source software ecosystems related to other software ecosystems. Haenni *et al.* [14] performed a survey to identify the information that developers lack to make decisions about the selection, adoption and co-evolution of upstream and downstream projects in a software ecosystem. Bavota *et al.* [11] has also looked into evolution issues of software ecosystems, particularly in the Apache software ecosystem. Similar work was performed by German *et al.* [13] for the R software ecosystem. Robbes [26] studied the API deprecations gravitating around the Squeak and Pharo software ecosystems. They found that dependency changes such as API deprecations can indeed have a very large impact on the ecosystem. This further illustrates how much impact an disruption to the software ecosystem can be for both library users and developers.

2) Library Reuse and Maintenance: Serebrenik [28] and Mens [21] discussed the many challenges that face software ecosystem, especially in terms on evolution updates and stability of dependencies. In summary, most work on library reuse has been on software library breakages, library updating and migration to stable versions. For instance, Bogart *et al.* [12] studied how developers reacted to stability of dependencies in their ecosystem. Other existing work are related to the traditional java and C programming language. For instance,

Teyton *et al.* [29] studied library migration for Java systems. In this work they studied library migrations of java open source libraries from a set of client with a focus on library migration patterns. Another work was by Xia *et al.* [32], that studied the reuse of out-dated project written in the c-based programming language. Recently, there has been large-scale empirical studies conducted on library migrations and evolution. Raemaekers *et al.* [24] performed several empirical studies on the Maven repositories about the relation between usage popularity and system properties such as size, stability and encapsulation. They also studied the relationship between semantic versioning and breakages [25]. Other related empirical studies were conducted by Jezek *et al.* [17] and Joel *et al.* [18]. They studied in-depth how libraries that reside in the Maven Central super repository evolve. In this work, we are concerned with the potential possibilities of library maintenance involved for micro-packages. So far this issue has only been reported by the JavaScript `npm` community, however, smaller designs may not necessarily mean less library support.

VI. CONCLUSION

In this paper, we explore the role of a micro-package in the `npm` JavaScript ecosystem. Results find that micro-packages form a significant portion of the `npm` ecosystem. Apart from the ease of readability and comprehension, we show that some micro-packages having long dependency chains incur just as much usage costs as the rest of the packages.

We envision that this work motivates the need to further investigate the notion of micro-packages and how to increase package resilience to changes in the ecosystem. The study concludes that it is important for developer to be aware of ecosystem changes that might affect their applications.

REFERENCES

[1] Ecmascript (javascript) parsing infrastructure for multipurpose analysis,accessed 2017-01-11. http://esprima.org/.

[2] Gephi - an open graph visualization platform, accessed 2017-01-11. https://gephi.org/.

[3] A javascript library browserify that lets you require('modules') in the browser by bundling up all of your dependencies, accessed 2017-01-11. http://browserify.org/.

[4] A javscript benchmarking library that supports high-resolution timers and returns statistically significant results.,accessed 2017-01-11. https://benchmarkjs.com/.

[5] A javscript build toolkit,accessed 2017-01-11. http://gulpjs.com/.

[6] Neo4j graph database,accessed 2017-01-11. https://neo4j.com/.

[7] Npm package manager for javascript,accessed 2017-01-11. https://www.npmjs.com/.

[8] Official blog on npm breakage of left-pad,accessed 2017-01-11. http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm.

[9] Simple tool to figure out the tree of require() calls and determine how time its taking to load,accessed 2017-01-11. https://github.com/vandium-io/require-profiler.

[10] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA, 1999.

[11] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. How the apache community upgrades dependencies: An evolutionary study. *Empirical Softw. Engg.*, 20(5):1275–1317, Oct. 2015.

[12] C. Bogart, C. Kästner, and J. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Proceedings of the ASE Workshop on Software Support for Collaborative and Global Software Engineering (SCGSE)*, 11 2015.

[13] D. M. German, B. Adams, and A. E. Hassan. The evolution of the r software ecosystem. *Proc. of European Conf. on Soft. Main. and Reeng. (CSMR2013)*, pages 243–252, 2013.

[14] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proc. of Int. Work. on Soft. Eco. Arch. (WEA13)*, pages 1–5, 2013.

[15] S. Jansen, S. Brinkkemper, M. A. Cusumano, S. Jansen, S. Brinkkemper, and M. A. Cusumano. *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar Publishing, Incorporated, 2013.

[16] S. Jansen, A. Finkelstein, and S. Brinkkemper. A sense of community: A research agenda for software ecosystems (pp. In *187–190). Presented at the 31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009, IEEE*, 2009.

[17] K. Jezek, J. Dietrich, and P. Brada. How Java APIs break - An empirical study. *Information and Software Technology*, pages 129–146, 2015.

[18] M. v. E. Joel Cox, Eric Bouwers and J. Visser. Measuring dependency freshness in software systems. 2:109–118, May 2015.

[19] M. Lungu. Towards reverse engineering software ecosystems. In *Intl. Conf. on Soft. Maint. and Evo. (ICSME)*, 2008.

[20] K. Manikas and K. M. Hansen. Software ecosystems - a systematic literature review. *J. Syst. Softw.*, 86(5):1294–1306, May 2013.

[21] T. Mens. An ecosystemic and socio-technical view on software maintenance and evolution. In *2016 IEEE International Conference on Software Maintenance and Evolution (Invited Paper)*, ICSME'16, 2016.

[22] T. Mens, M. Claes, and P. Grosjean. Ecos: Ecological studies of open source software ecosystems. In *Soft. Main. Reeng. and Rev. Eng. (CSMR-WCRE)*, pages 403–406, Feb 2014.

[23] D. L. Parnas. Software pioneers. chapter The Secret History of Information Hiding, pages 399–409. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[24] S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *Proc. of Intl. Comf. Soft. Main. (ICSM)*, pages 378–387, Sept 2012.

[25] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the maven repository. pages 215–224, Sept 2014.

[26] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to api deprecation?: The case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 56:1–56:11, New York, NY, USA, 2012. ACM.

[27] J. Romano, J. Kromrey, J. Coraggio, and J. Skowronek. *Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys?* 2006.

[28] A. Serebrenik and T. Mens. Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15, pages 40:1–40:6, New York, NY, USA, 2015. ACM.

[29] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc. A study of library migrations in java. *Journal of Software: Evolution and Process*, 26(11), 2014.

[30] S. Wasserman and K. Faust. *Social network analysis: Methods and applications*.

[31] E. Wittern, P. Suter, and S. Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 351–361, New York, NY, USA, 2016. ACM.

[32] P. Xia, M. Matsushita, N. Yoshida, and K. Inoue. Studying reuse of out-dated third-party code in open source projects. In *Japan society for Software Science and Technology*, Computer Software, Vol.30, No.4, pp.98-104, 2013.