On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages

Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, *Senior Member, IEEE* and Bram Adams, *Senior Member, IEEE*

Abstract—Nowadays, developing software would be unthinkable without the use of third-party packages. Although such code reuse helps to achieve rapid continuous delivery of software to end-users, blindly reusing code has its pitfalls. For example, prior work has investigated the rationale for using packages that implement simple functionalities, known as trivial packages (i.e., in terms of the code size and complexity). This prior work showed that although these trivial packages were simple, they were popular and prevalent in the npm ecosystem. This popularity and prevalence of trivial packages peaked our interest in questioning the 'triviality of trivial packages'. To better understand and examine the triviality of trivial packages, we mine a large set of JavaScript projects that use trivial npm packages and evaluate their relative centrality. Specifically, we evaluate the triviality from two complementary points of view: based on project usage and ecosystem usage of these trivial packages. Our result shows that trivial packages are being used in central JavaScript files of a software project. Additionally, by analyzing all external package API calls in these JavaScript files, we found that a high percentage of these API calls are attributed to trivial packages. Therefore, these packages play a significant role in JavaScript files. Furthermore, in the package dependency network, we observed that 16.8% packages are trivial and in some cases removing a trivial package can impact approximately 29% of the ecosystem. Overall, our finding indicates that although smaller in size and complexity, trivial packages are highly depended on packages by JavaScript projects. Additionally, our study shows that although they might be called trivial, nothing about trivial packages is trivial.

Index Terms—Trivial Packages, npm ecosystem, Mining Software Repository.

1 Introduction

The use of third-party packages is becoming increasingly popular since it allows teams to reduce development time and costs and increase productivity [1], [2], [3]. A major enabler for the use of third-party packages (hereafter referred to as packages) is the capability for developers to easily share their code through software packages on dedicated platforms, known as software package managers (e.g. Node Package Manager (npm) and Python Package Index (PyPI)). Entire ecosystems have been created around these package managers, e.g., the Node.js ecosystem is largely supported by npm [4].

Despite the many benefits and wide popularity of using software packages, they also pose some major drawbacks such as increased maintenance costs, an increased risk of exposure to vulnerabilities and even legal issues [5], [6], [7], [8]. One specific incident, the *left-pad* incident [3], [9], triggered a large debate on whether developers should be reusing packages for "trivial tasks"[1]. Since then a number of studies focused on the topic of "trivial packages" and found that indeed, the left-pad incident is not isolated, and that trivial packages account for more than 17% of the 800,000 packages on npm [3], [10]. In addition, these packages tend to be heavily used, with some trivial packages (e.g., `escape-string-regexp`) being downloaded more than eleven million times per week [11].

The fact that these trivial packages, in terms of the code size and complexity, play such a central role made us ask the question **are trivial packages really trivial?** Although we do agree that these packages may be small in size and implement very specific functionality, the fact that they are so prevalent is something that warrants the questioning of their triviality. Therefore, in this paper, we examine triviality of trivial packages based on their *usage*. In particular, we focus on the usage of trivial packages in 1) the projects that use them (**project usage**) and 2) the role they play in the ecosystem they belong to (**ecosystem usage**).

We perform an empirical study by analyzing more than 15,000 JavaScript projects, of which 3,965 depend on trivial packages. To examine **project usage**, we use static analysis to determine the centrality of the files that use trivial packages and analyze how widely the trivial packages are used in these files. To examine **ecosystem usage**, we leverage network analysis to examine the role of trivial packages in the ecosystems dependency network. Our study is formalized through three Research Questions (RQs):

- **Project usage. RQ1: Are trivial packages used in central parts of JavaScript projects?** Since these packages are small in size and complexity, one may expect that they are used in unimportant parts of software projects. Thus, to better understand how projects use trivial packages, we examine their role in the source code files of the projects

- *Md Atique Reza Chowdhury and Emad Shihab are with the Data-driven Analysis of Software (DAS) Lab at the Department of Computer Science and Software Engineering, Concordia University, Montréal, Canada. E-mail: m_wdhu, eshihab@encs.concordia.ca*
- *Rabe Abdalkareem is with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queens University, Canada E-mail: abdrabe@gmail.com*
- *Bram Adams is with the Lab on Maintenance, Construction, and Intelligence of Software (MCIS), Département de Génie Informatique et Génie Logiciel, École Polytechnique de Montrèal, Montréal, Canada. E-mail: bram.adams@polymtl.ca*

Manuscript received April 19, 2005; revised August 26, 2015.

1. The left-pad incident refers to a 11-line package that implements simple string manipulation. This package was used by Babel, a package that is used by most major website, including Facebook, Netflix, and Airbnb.

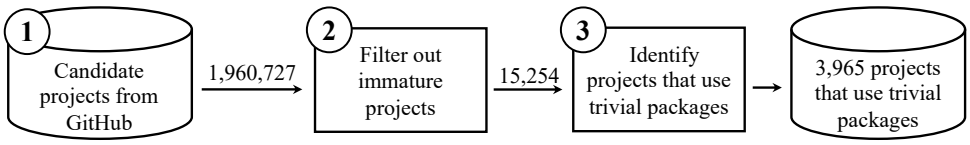


Figure 1: An overview of our data collection approach.

that depend on them. Using the file-level dependency graph, we find that files that depend on trivial packages are very central in their respective projects. This finding indicates that trivial packages may not be so trivial after all, since they are used in central parts of the projects that depend on them.

- **Project usage. RQ2: How widely used are trivial packages in JavaScript projects?** In addition to knowing if the trivial packages are used in central parts of the projects, we want to investigate how heavily a trivial package's APIs are used within a JavaScript file to determine these package's importance within the trivial dependent JavaScript files (i.e., are they only used in one central part or throughout the projects). Again, we use static source code analysis to determine the percentage of API calls that are made to trivial packages. Also, we measure the entropy of the package to determine how widespread its use is. We find that trivial packages are at least as widely used as non-trivial packages, indicating that they may not be so trivial.
- **Ecosystem usage. RQ3: Do trivial packages play a central role at the ecosystem level?** To complement our analysis in RQs 1 and 2, which focus on project-level usage, we examine the centrality of trivial packages within the ecosystem. We study the package dependency network for both direct and transitive dependencies of the studied projects. We find that trivial packages are more central to the ecosystem than non-trivial packages. Furthermore, we find that removing certain trivial packages from the ecosystem may impact up to 29% of the packages in the ecosystem. Our result shows that npm trivial packages are central building blocks in the ecosystem, and hence, their role is not so trivial.

Our study makes the following contributions:

- To the best of our knowledge this is the first in-depth study that examine the centrality and role of trivial packages to projects using them and to the ecosystem they belong to.
- The findings of this paper are based on an extensive analysis, which includes a large dataset of JavaScript projects that depended on trivial packages and the use of state-of-the-art technique that include dependency network analysis.
- To encourage replication and further studies on the use of trivial packages, we have disclosed our dataset and source code for our analysis in our replication package [12].

Paper organization: Section 2 presents our study design and approach. We describe our results in section 3. We discuss the results and implications of our study in Section 4 and related works in Section 5. Threats to validity is shown in Section 6. Finally, Section 7 concludes our paper.

Table 1: Filtering steps of the studied JavaScript projects.

Filtering Step	# Projects
JavaScript projects in GitHub	7,863,361
npm projects in GitHub	2,289,130
JavaScript projects that are not npm packages	1,960,787
Filtering out immature and/or inactive projects	15,254

2 Case Study Design

To investigate the role of trivial packages in software projects, we study a large dataset of JavaScript software projects that depend on at least one npm trivial package. Figure 1 shows an overview of our general approach. We describe each step in our approach below.

2.1 Dataset of Candidate Projects

Since our analysis focuses on understanding the role of trivial packages in software projects that use them, we need to study a diverse and sufficiently large number of JavaScript projects that depend on trivial packages.

To acquire our dataset, we resort to the public GHTorrent dataset [13], [14] to extract information about all the JavaScript projects hosted on GitHub. We extract the data pertaining to 7,863,361 JavaScript projects that are hosted on GitHub, as of 15th March 2019. We then filter out projects that do not use npm as their package management system. As a result, we found 2,289,130 projects that use npm as their package management system (i.e., projects have `package.json` file, which is the configuration file for npm projects). Moreover, since some npm packages use GitHub as their code repository [15], we exclude these npm packages from our list by crosschecking our list of URLs and GitHub URLs of all the npm packages. It is important to note that we exclude npm package repositories from our dataset so we do not analyze them as standalone JavaScript projects. We identify 328,343 npm packages in our list of candidate projects and we filter these packages out.

2.2 Pruning List of Projects

As recommended in prior work [3], [16], we perform extra steps to eliminate immature projects from our candidate dataset. To do so, we adopt similar filtering criteria that were used in prior work [3], [16]. We choose to select projects that

Table 2: Summary of the number of developers, commits, watchers, and stars for 15,254 JavaScript projects.

Measurement	Min.	Median	Mean	Max.
Developers	2	5	6.74	69
Commits	100	271	669	97,504
Watchers	1	6	23.99	2,451
Stars	1	9	303.73	48,765

Table 3: The distribution of "Line of Code" and "Cyclomatic Complexity" of all packages (trivial and not-trivial) used in our studied JavaScript projects.

Type of packages	Line of Code				Cyclomatic Complexity			
	Min.	Median	Mean	Max.	Min.	Median	Mean	Max.
Trivial	7	19	19.35	34	1	5	5.324	10
Non-Trivial	36	200	2044.7	853,967	2	50	699.58	313,291

are non-forks, have more than 100 commits by more than one contributor and have a community interest in them (i.e., projects that have at least one star and a watcher on GitHub). Finally, we select the projects that have at least one external npm package dependency. These filtering steps allow us to extract a list of 15,254 JavaScript projects that are the client of npm packages (step 2 Figure 1). Table 1, shows the steps and number of projects after each step in the dataset acquisition process. Table 2 shows the summary statistics for different metrics of the selected JavaScript projects in our candidate dataset. As the table shows, our dataset contains a good distribution of projects in terms of developers, commits, watchers and stars.

2.3 Identifying JavaScript Projects that Use Trivial Packages

Since the goal of this study is to understand the role of trivial packages in JavaScript projects, we need to identify projects that depend on trivial npm packages in the selected candidate projects. To do so, we start by cloning the selected 15,254 projects. Then, we analyze them following a four-step approach (step 3 in Figure 1) to identify projects that use trivial packages.

First, we extract each project's dependency information by examining the package.json file, which is the configuration file for npm projects. The package.json, among other configurations, specifies the list of packages that the project depends on. We extract the package name and its associated version for each runtime dependency for each project in our 15,254 projects candidate dataset. We only consider runtime dependencies since they are required to install and run the projects.

Once we have the list of dependencies for each project in our candidate dataset, we download these packages using the package name and related version information. We download the dependent packages by using the npm-pack command [17]. The npm-pack command consults with the npm registry [18] and resolves the semantic version and downloads the appropriate 'tar' file that contains the source code of the package for each dependency-version pair.

Third, once we have the 'tar' file for each npm package, we analyze them to identify trivial packages. To do so, we extract the 'tar' file and analyze if the package is trivial or not by leveraging the definition proposed by Abdalkareem *et al.* [3], which categorize a package as trivial if its number of JavaScript "Line of code (LOC)" ≤ 35 and "Cyclomatic Complexity" ≤ 10. We analyze all the packages using the Understand tool [19]. Understand is a static analysis tool that provides, amongst other metrics, Line of Code (LOC) and Cyclomatic complexity measures for the packages. Table 3 shows the distribution summary of Line of Code LOC and Cyclomatic Complexity CC measurements of all analyzed trivial and non-trivial packages used in our studied JavaScript projects.

Table 4: The distribution of the number of npm packages that are used in all the JavaScript projects in our dataset.

Type of packages	Min.	Median	Mean	Max.
Trivial	1	2	2.34	31
Non-trivial	1	16	19.69	106

Forth, we identified projects that are trivial package dependent (i.e. projects that use at least one trivial package). To do so, we first used the depchecker [20] tool to extract the npm packages that are actually used in JavaScript files. The depchecker tool analyzes the dependencies in a project to identify how each dependency is used (i.e. identifies unused dependencies in the JavaScript source code and dependencies that are missing from package.json). Then, for each file in the studied JavaScript projects, we extract the number of actual dependent packages and how many of these dependent packages are trivial based on the definition proposed by Abdalkareem *et al.* [3] (as described in the previous step). If a file depends on one or more trivial packages, we consider that file as a trivial dependent file, otherwise, we consider it as a non-trivial dependent file. In the same way, if a project has at least one trivial dependent file then we identify that project as a trivial dependent project.

According to this approach, in our candidate dataset, among the 15,254 JavaScript projects that we analyze, 26% (3,965) of the projects are trivial dependent. Since we want to analyze the role of trivial packages in JavaScript projects, we conduct our analysis on these 3,965 JavaScript projects dataset that use at least one trivial npm package. Table 4 shows the distribution of trivial and non-trivial packages in the projects in our dataset.

3 Case Study Result

This section presents the results to our three RQs. For each RQ, we provide a motivation, describe the approach used and present our results.

3.1 RQ1: Are trivial packages used in central parts of JavaScript projects?

Motivation: Previous work showed that trivial npm packages are widespread, and has arguably some negative impact on software projects [3]. However, since these packages are small in size and complexity, one may expect that they are used in unimportant parts of software projects. To understand how projects use trivial packages, we examine their role in the source code files of the dependent projects. For example, if a trivial package is used in isolated part

Table 5: The distribution of number of files in the studied JavaScript projects in our dataset. The table shows the distribution of number of all, trivial, and non-trivial dependent files in our dataset.

File Type	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
All	10	16	27	49.3	49	1,592
Trivial	1	1	2	4.3	4	161
Non-Trivial	1	14	25	45.7	45	1,592

(i.e., file) in a project then its impact on that project can be neglected. Answering this question will help us understand the relative centrality of trivial packages in the software projects that use them.

Approach: To examine a trivial package's centrality in a JavaScript project, we identify the files that use trivial packages since they provide a direct link between trivial packages and their centrality in a project.

We examine the centrality of trivial dependent files by analyzing the file-level dependency graph among the files of a project and measure the centrality score [21] of trivial dependent and non-trivial dependent files. To identify the JavaScript files that are more central in a software project, we apply network analysis on the file-level dependency graph of each project and measure the centrality score. The centrality score of a node in a network reflects how central that node is in the network [22], [23], [24]. In scientific literature, network analysis is a popular measure in social sciences, which studies networks between humans (actors) and their interactions (ties). In our context, the JavaScript files are the actors and their inter-dependencies are the ties. For each JavaScript file within a project, we extract information on which other files the concerned file depends on (out-degree) and by which other files the concerned file is being dependent upon (in-degree). Then, we calculate the degree centrality score [21] for each file of a project in our dataset. The degree centrality score is a measure of the number of in-degree and out-degree for a JavaScript file within a project. This degree centrality score is normalized by dividing by $n-1$, the maximum possible degree in a graph that has n total nodes in that graph. The degree centrality of a node V_i is given by:

$$Degree\ Centrality\ (V_i) = \frac{|N(V_i)|}{n-1} \tag{1}$$

Where the $|N(V_i)|$ is the number of nodes (files in our case) that are connected to the node V_i (i.e., file under examination). The degree centrality score has a value that ranges between 0 and 1, where 1 means that the node is in the center of the network (i.e., connected to all other nodes) and zero indicates that the node is isolated.

To calculate the degree centrality of trivial and non-trivial dependent files in each project in our dataset, we start by generating a file-level dependency graph representation of files in every JavaScript project in our dataset. We use the madge tool to generate the file-level dependency graphs [25]. The madge tool is a static source code analysis. It first parses the source code of files and generates the abstract-syntax-tree (AST) of a given JavaScript file in a project. It then checks the AST to determine, which module (i.e, JavaScript file) is being called in another file. Then, the madge tool constructs a file-level dependency graph of a project by traversing the generated AST. It starts from an entry point (i.e, JavaScript file) and detects all import statements in the AST. All files that are recursively accessible from the projects entry point are marked as dependent on files. The output of the madge tool is a file-level dependency graph that shows each file in a software project and a list of files it depends on. We configure the madge tool to analyze every JavaScript project in our dataset and generate a file-level dependency graph representation. Next, we mark the trivial dependent files in the generated file-level dependency graph.

After that we run the networkx tool [26] on the generated file-level dependency graph, to calculate the centrality score of every file in the generated file-level dependency graph as explained earlier. The networkx tool is a well-known tool for analyzing and visualizing social network data. Finally, to put our results in perspective, we compare and contrast the degree centrality score for trivial and non-trivial dependent files.

In addition, to get an in-depth understanding of the JavaScript file's relative centrality within a software project, we rank the files based on their degree centrality score, e.g., JavaScript file with highest degree centrality score is ranked 1 and the rank increases with decreasing degree centrality values. In case if two JavaScript files have similar degrees of centrality scores, they will have the same ranking. Since the trivial dependent projects in our dataset vary in the number of JavaScript files, we segment the projects into four groups (based on the quartile they fall in) namely small, small-mid, mid-large, and large projects based on the distribution of the number of JavaScript files in the projects. From the distribution of the number of files in the studied projects, shown in the first row of Table 5, we group projects having #files $<$ 1st Qu. into small projects; 1st Qu. $\leq$ #files $<$ median into small-mid projects; median $\leq$ #files $<$ 3rd Qu. into mid-large projects, and #files $\geq$ 3rd Qu. into large projects. In addition, to put our results in perspective, we again compare the distribution of degree centrality rank for trivial dependent and non-trivial dependent files in each group of projects.

Results: Table 6 shows the summary distribution of the degree centrality score for trivial and non-trivial dependent files in our dataset. Here, we observe that overall the degree centrality values for trivial dependent files are higher than that of non-trivial dependent files. The table shows that the median/mean degree centrality values are 0.022/0.061 and 0.003/0.021 for trivial and non-trivial dependent files, respectively. To test if the difference is statistically significant between the two result sets, we applied the non-parametric Wilcoxon rank-sum test [27]. We determine if the difference is statistically significant at the customary level of 0.01. We also estimated the magnitude of the difference between

Table 6: The distribution of the degree centrality of trivial and non-trivial dependent JavaScript files.

File Type	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Trivial	0.00	0.003	0.022	0.061	0.070	1
Non-Trivial	0.00	0.000	0.003	0.021	0.019	1

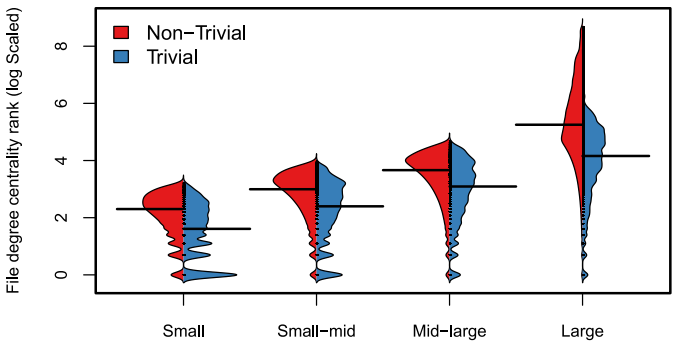


Figure 2: Distribution of degree centrality rank of trivial dependent and non-trivial dependent files in different project groups based on number of files.

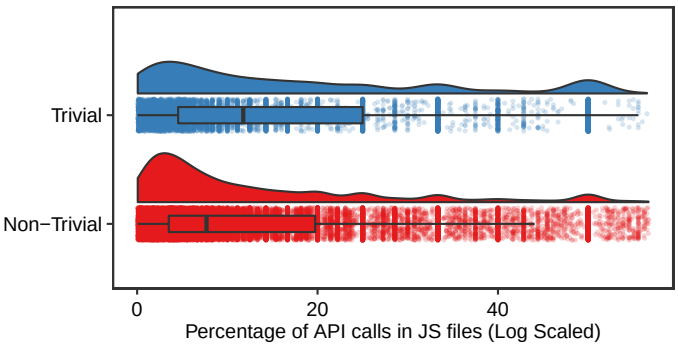


Figure 3: The distribution of percentage of API calls for trivial and non-trivial packages in JavaScript files.

datasets using the Cliff's Delta [28] (or d). Cliff's Delta is a non-parametric effect size measure for ordinal data. We consider the effect size values: negligible for $|d| < 0.147$, small for $0.147 \leqslant |d| < 0.330$, medium for $0.330 \leqslant |d| < 0.474$ and large for $|d| \geqslant 0.474$. We found that the results are statistically significant (p-value $<$ 2.2e-16) with medium effect size ($d = 0.3471$).

In addition, Figure 2 shows a beanplot distribution of the degree centrality rank of trivial dependent and non-trivial dependent files for the four groups of projects. From Figure 2, we observe that for each group of projects, the trivial dependent files have a lower degree centrality rank than that of non-trivial dependent files, which indicate that trivial packages are used in central part of the projects. Also, the results for each segment is significant (p-value $<$ 2.2e-16). We also measured the effect size and observed -0.3853 (medium), -0.2397 (small), -0.3355 (medium) and -0.5040 (large) cliff's delta value for small, small-mid, mid-large and large projects respectively. Overall, these results highlight that trivial packages are used in files that are more central in the studied JavaScript projects.

To investigate why trivial packages are used in more central parts of the studied projects, we perform a manual investigation. To do so, we randomly selected five projects. Then, for each trivial dependent file in the selected project, we extracted the trivial packages. After that, we perform a manual process to understand the type of functionalities that are provided by the trivial packages in these projects. For each trivial package, the first two authors examine the source code and the readme file of these packages.

Based on this manual process, we observe that the majority of the examined trivial packages in the selected five projects provide mainly utility functionality. For example, they provide functionalities such as string manipulation, stream operation, and file operations. We also found that the examined trivial packages provide functionalities regarding different types of data structure manipulation; numerical, geometric, and logical functionalities.

Our findings indicate that trivial packages are used in central parts of software projects compared to non-trivial packages. In our dataset, trivial dependent files have on median 0.022 degree centrality value while it is 0.001 for non-trivial dependent files. This difference is statistically significant. These findings show that trivial packages are used in central parts of the projects that depend on them. Hence, they may not be so trivial after all.

3.2 RQ2: How widely used are trivial packages in JavaScript projects?

Motivation: Thus far, we saw that trivial packages are used in central parts of the projects that depend on them. Next, we want to examine the diffusion of a used package across the projects. In another word, we want to examine whether trivial packages are used only in central parts of the projects or their usage is dispersed across different parts of the projects. For example, prior work showed that if the Application Programming Interfaces (API) of a package Pkg_A are invoked less than APIs' of another package Pkg_B in a software project then this is a clear indication that Pkg_B is more important than Pkg_A in that specific project [29]. Thus, low usage of trivial package APIs' in a JavaScript file suggests that, even if these packages are used in more central files, these package's importance within that file is low. Therefor, we investigate how heavily a trivial package's APIs are used within a JavaScript file to determine these package's importance within the trivial dependent JavaScript files.

Approach: To determine how important trivial packages are within a project, we again perform a two-way complementary analysis. First, we measure the percentage of each package's project programming interface calls in a file that depends on an external package in our dataset. Then, we examine how widespread the use of a package is in each project. Specifically, we use static code analysis and calculate the following two measures:

Percentage of trivial package API calls in a trivial dependent file: Although, based on our definition, a trivial dependent file has at least one trivial package dependency, in fact, it can have any number of non-trivial package dependencies. In our dataset, the median number of trivial and non-trivial packages in trivial dependent files are 1 and 3, respectively.

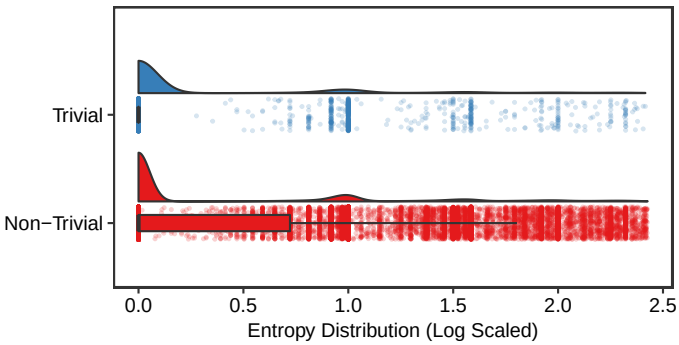


Figure 4: The distribution of trivial and non-trivial package API entropy.

Therefore, since these files have a lower number of trivial package dependencies, we want to understand what percentage of total API calls in a trivial dependent file are associated with trivial packages.

To do so, we use a static source code analysis tool to extract and measure all the occurrences of external package API calls in JavaScript files. For each of the source files, we extract the API call of the external packages using the Understand tool [19]. We use the Understand tool [19], which is a source code analysis tool that, among other things, extracts API calls in JavaScript files and has been extensively used in other research work [30], [31], [32].

The Understand tool performs a complete lexical parse of the source code, similar to what a compiler would do. Using that parse information, it creates a list of "entities" in the source code. An entity is any semantic object that the tool can capture information on - such as a class, method, or variable. Then it creates a database of how all of those entities are used and how they interact with each other. We call those uses and interactions "references". The Understand API then uses a custom string syntax to search the generated database to find combinations of references and entities. For example, give a list of all functions in the project, or find everywhere that a function in list A is called by a function in list B. Then, we calculate the percentage of a packages API calls within a JavaScript file by counting all the API calls in that file.

External package entropy: We again use the extracted information about the API calls of external packages to compute the entropy of the packages. In our study, the entropy of a package shows how widely the package is used in a project. The higher the entropy of a package (i.e., API usage spread across files.), the more difficult it gets to uproot the package from the project. Similar to prior work [33], [34], we define the entropy of an external package as the distribution of API calls of that package across files. For example, in a JavaScript project, the package Pkg_x's APIs are called 10 times in file $F1$, 15 times in file $F2$, and twice in file $F3$, we calculate the entropy of the package Pkg_x as $(-\frac{10}{27}\log_2\frac{10}{27} - \frac{15}{27}\log_2\frac{15}{27} - \frac{2}{27}\log_2\frac{2}{27})$, which equal to 1.28. It is important to note that the higher the entropy value the more widespread is the usage of the package is in a JavaScript project and if a package is used only in a single file then its entropy is zero.

Result: Figure 3 shows the distribution of percentage of API calls for trivial packages and non-trivial packages within the trivial dependent files. Here, we observe that median value of percentage of API calls for trivial packages within trivial dependent files is higher than that of non-trivial packages with median of 11.76% and 7.69% calls, respectively. We also examine whether the result is statistically significant and we also calculate the effect size. We found that the results are statistically significant (p-value $<$ 2.2e-16) and effect size is small (Cliff's delta estimate $=$ 0.25). This API call analysis of trivial dependent files shows that trivial packages play important role in these files.

In the second part of analyzing this research question, we investigate the distribution of API calls of a trivial package across the project by computing its entropy. Figure 4 shows a bean-plot distribution of entropy scores for trivial and non-trivial packages. We observe that trivial and non-trivial packages have similar entropy score distribution with median entropy score equal to zero for both types of packages. Most of the packages (68.067%) in our dataset have zero entropy scores, which suggests that these packages are used in only a single JavaScript file in the studied JavaScript projects. This result is statically significant with p-value $<$ 1.789e-05 but the effect size is negligible (Cliff's delta estimate: -0.1119). The entropy score distribution of trivial and non-trivial packages indicates that trivial and non-trivial packages tend to be used in different ways, but these two types of packages are essential in software projects.

> ***A higher percentage of total API calls of JavaScript files are associated with trivial packages (11.76% and 7.69% for trivial and non-trivial packages) and thus these packages are important within these files. Moreover, entropy distribution of trivial and non-trivial packages shows both types of packages are important in software projects. Our results indicate that trivial packages are consider to be as widely used as non-trivial package in the projects that depend on them.***

3.3 RQ3: Do trivial packages play a central role at the ecosystem level?

Motivation: In previous research questions, we found that trivial packages are important components for the JavaScript projects that directly depend on them. However, npm packages, trivial or non-trivial, do not exist in isolation, they interconnect with other packages and they form what is known as the npm ecosystem. We believe that examining how central trivial packages are with the software ecosystem that they belong to will provide us with a general understanding of their importance. Thus, in this question, we seek to understand the centrality of a trivial package in the dependency network of npm ecosystem, which consists of all direct and indirect dependencies of the studied projects.

Approach: To examine the centrality of trivial packages from the npm ecosystem perspective, we extract all the dependencies (direct and indirect) for each JavaScript project in our dataset and construct its dependency network graph. To extract this package dependency graph, initially, we install and clone the projects' dependencies by using

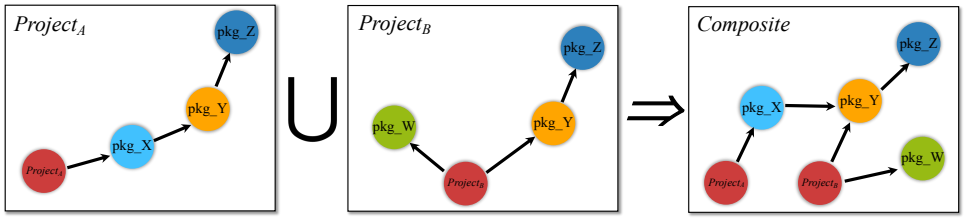


Figure 5: Composite Dependency Network.

the `npm install` command, which install the package version specified in package.json file. By doing so, all the direct and indirect dependencies of every project in our dataset are saved locally in the project's home directory in a folder named "node_modules". Then, we use the `npm-ls` [35] command to list installed package and their inter-dependencies in *json* format. Subsequently, we merge all the dependency network graphs of all the projects in our dataset and compile a composite dependency network at a given point in time. Figure 5 depicts an example of the process of merging the dependency network graphs of two JavaScript projects ($Project_A$ and $Project_B$). In our illustrating example, $Project_A$ is directly dependent on pkg_X which in turn depends on pkg_Y whereas pkg_Y depends on pkg_Z. $Project_B$ has two direct dependencies and one transitive dependency. Here, in the composite dependency network, dependency hierarchy is preserved while accommodating all the dependencies of both projects. We recursively apply this merging process on all the dependency network of all the projects in our dataset. As a result of this merging process, we get a composite package dependency network that consists of 32,319 connected packages. Then we analyze the source code of each package in the constructed dependency network and identify trivial and non-trivial packages. We find that 16.8% of 32,319 packages in the constructed dependency network are trivial packages. After that, we use the composite packages dependency network to examine the centrality of trivial packages in two complementary measures. First, we measure the centrality of trivial packages within this dependency network using the PageRank algorithm [36]. Second, we study the centrality of the trivial packages by measuring the *Technical Bus Factor (TBF)* of these packages. Similar to the idea of *social bus factor*, which measures the effect of removal of a developer from a project, the *TBF* measures the effect of the removal of a package from a dependency network [37]. In the following subsection, we describe how we measure these values for every package in our constructed graph.

PageRank of External Packages: PageRank score [36] of a node (packages in our case) indicates the centrality of the node in a network. The more dependent on a node in a network the higher is its PageRank score. PageRank has a value range between 0 and 1. We calculated the PageRank score of every package (trivial and non-trivial) in our composite package dependency network. To do so, we use the well-known network analysis tool called *networkx* tool [26]. Then, to put our results in perspective, we compare the PageRank score of trivial and non-trivial packages.

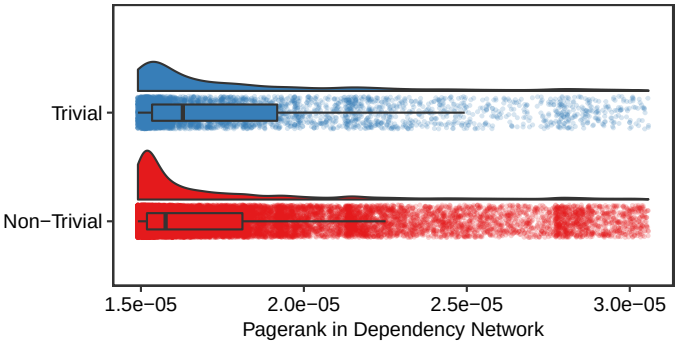


Figure 6: The distribution of PageRank values for trivial and non-trivial packages.

Technical Bus Factor (TBF): To understand the effect of removing one trivial package from the package dependency network, we calculate *TBF*, which simulates the removal of a package from our constructed composite network. We then evaluate how many other packages, directly or indirectly dependent on the removed package, are affected. We calculate what percentage of 32,319 packages, which is the total number of packages in our dependency network, are affected by the removal of one package from the package dependency network. The higher a package's *Technical Bus Factor (TBF)* value; the more central that node is in the package dependency network.

Result: Figure 6 shows PageRank score distribution for trivial and non-trivial packages. We notice that the median PageRank score of trivial packages (1.71e-05) is higher than that of non-trivial packages (1.61e-05). This result is significant (p-value $<$ 2.2e-16) and effect size is small (Cliff's delta estimate: 0.1578). This result shows that many packages are dependent upon trivial packages which makes trivial packages vital nodes in the ecosystem that they belong to.

Table 7 shows the statistical summary of the distribution of *technical bus factor (TBF)* of the trivial and non-trivial packages. From Table 7, we see that removing a trivial package from our composite dependency network has a much larger impact than that of non-trivial package

Table 7: The statistical summary of the distribution of technical bus factor (TBF) for the trivial and non-trivial packages in our composite dependency network.

File Type	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Trivial	0.00	0.0031	0.0155	3.5324	0.1918	36.8174
Non-Trivial	0.00	0.0031	0.0093	1.9480	0.0495	34.9485

Table 8: The top-20 most impactful trivial packages measured by *Technical Bus Factor (TBF)*.

Packages	TBF	Rank	Functionality
inherits	36.82	1	Inherits one constructor's prototype to another constructor.
isarray	35.43	2	Checks if the object in the argument is an array.
process-nextick-args	34.15	10	Amends the functionality of process.nextTick, which defers a callback function until next eventloop, by enabling it to accept arguments.
debuglog	34.13	11	Shows degugging information in stderr.
escape-string-regexp	32.26	14	Escapes special characters.
ansi-regex	32.00	18	Matches ANSI escape codes.
object-assign	31.90	22	Assigns values to objects.
strip-ansi	31.89	24	Removes ANSI escape codes from a string.
indexof	31.14	49	Returns index of an object in an array.
foreach	30.87	59	Iterates over the key value pairs of either an array or a dictionary like object.
pinkie-promise	30.54	63	Returns JavaScript promise object
is-object	30.20	64	Checks if the argument is an object.
get-stdin	30.10	65	Get standard input as a string or buffer.
xtend	30.03	68	Extends an object by appending all of the properties from each object in a list.
has-flag	29.77	70	Checks if function argument has a specific flag.
has-color	29.67	73	Detects whether a terminal supports color.
once	29.65	74	Restricts a function to be called only once.
graceful-readlink	29.02	79	Returns a file's symbolic link.
number-is-nan	28.91	82	Checks whether the value in the argument is undefined and its type is Number

removal. We see that the median *TBF* values for trivial packages is 0.0155 while it is 0.0093 for non-trivial packages. We observe that this result is a statistically significant with p-value $<$ 2.2e-16 and small effect size (Cliff's delta estimate: 0.1525).

To investigate the characteristics of trivial packages that have the highest *TBF* values in our dependency graph, the first two authors manually examine the top twenty trivial packages. Table 8 shows the names, *TBF* values, their ranks in dependency network based on *TBF* and the description of the functionalities of the top trivial packages. We rank these packages in dependency network based on their *TBF* where package with highest *TBF* is ranked 1 and rank increases with decreasing *TBF*. From Table 8, we see that these trivial packages have *TBF* values ranges between 36.82 and 28.91, which means that trivial packages in the list based on the *TBF* value can affect at least approximately 29% of all packages in the dependency network when any one of these is removed.

Based on our manual examination of these trivial packages, we found that these packages provide popular utility functions, enhancement of JavaScript standard functionalities and cross-platform compatibility features.

First, the examined trivial packages provide some popular utility functions like checking objects e.g. has-flag, has-color, is-object, number-is-nan; string operations e.g. ansi-regex, strip-ansi; and object manipulation e.g. xtend, foreach. The second group of the examined trivial packages is used to enhance existing native functionality of the JavaScript engine. For example, `process-nextick-args` [38] extends the capability of *process-nextick* by enabling this function to accept arguments. Finally, we found some trivial packages provide functionalities that help developers to deal with the cross-platform compatibility. Since JavaScript code can be run on different types and versions of web browsers, these packages provide backward and forward compatibility. For example, *isarray* package [39] is a well-known package and in the dependency network it is ranked 2nd based on it's *TBF*. It provides same functionality like the native *Array.isArray*. *Array.isArray* supports browsers with newer version, e.g. IE9+, Chrome 5+, Firefox 4+, Opera 10.5+ and Safari 5+. However, as this function is not supported in older versions of browsers, *isarray* package is widely used because it supports older browser versions that are not compatible with ECMAScript 5 or later. These types of packages that provide cross-platform compatibility are known as *ponyfills* and *polyfills* [40]. Whereas *polyfills* are prone to unexpected bugs as these pollute the global scope, *ponyfills* is the smarter alternative which exports functionalities as a module without exploiting global scope. 25% of top 20 trivial packages e.g isarray, debuglog, object-assign, pinkie-promise, number-is-nan, are *ponyfills*. Furthermore, 37.97% of all *ponyfill* solutions in npm are trivial packages [40], [41]. From this analysis we see that trivial packages are often the byproduct of compatibility ensuring efforts.

Additionally, this analysis of the top 20 trivial packages revealed that some developers have a proclivity of publishing trivial packages. For example, Sindre Sorhus [42], a famous open-sourcerer, who created Yeoman [43] and Awesome Project [44], collaborated 7 of the top 20 trivial packages. We examined all of his 1,148 packages in npm and surprisingly 55.14% of his published packages are trivial packages.

Trivial packages are vital nodes in the package dependency network (i.e., ecosystem). In fact, our results show that 16.19% of trivial packages and only 9.27% of non-trivial packages have a TBF value grater than 15%. These results indicate that trivial packages play a central role in the npm ecosystem, and in some cases removing a trivial package can at least approximately affect 29% of all packages in the dependency network.

4 Discussion

In this section, we first discuss our findings concerning the centrality of trivial package overtime in software projects. Then, we discussed the implications of our findings.

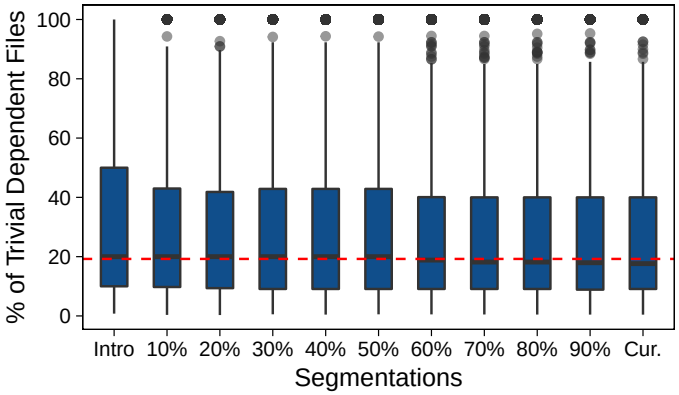


Figure 7: The distribution of the percentage of trivial dependent files in all the studied projects based on TDDT segmentations. Dotted horizontal line present overall median.

4.1 Re-examining the Role of Trivial Packages Overtime

Our results were presented on a specific snapshot of the projects and their dependencies. Hence, in this subsection, we further investigate the validity of our findings over time.

In research questions 1 and 2, we focus on studying the centrality of trivial packages from the usage perspective. To do so, we examine the current snapshot of the studied projects[2]. Now, we want to examine the role of trivial packages in the studied projects overtime. We believe that examining usage of trivial package over time will provide us with a general overview of the usage of trivial packages compare to only examine the current snapshot. Also, an increment in the number of trivial dependent files overtime in a software project suggests these packages' importance and developer's reliance on these packages whereas decrement suggests otherwise.

First, we examine the evolution of the number of trivial dependent files over their development timespan of a project. Second, we analyze the evolution of percentage of trivial package API calls in trivial dependent files over the development timespan of software projects. To identify the development period in which a project has some trivial package dependency, we need to know the commit that introduced the first trivial package in a project. This commit is either the first commit in a software project or before this commit the project was non-trivial dependent. Since all the projects in our dataset use git as their source control system, we iterate each commit starting from the initial commit of a software project to check if the commit is adding any trivial package into a JavaScript file. When we encounter such commit, we break the iteration and mark and register that commit as a trivial introducing commit for that software project.

Trivial dependent projects in our dataset start being trivial dependent from the trivial introductory commit. We consider the development timespan of a project, which ranges from first trivial introductory commit till the latest commit as trivial dependent development timespan (TDDT). We segment this TDDT into 10 equal parts by the means of the total number of commits in this period. For each project, we count the total number of commits in its TDDT and take a snapshot at each 10th percentile commit. Therefore, this segmentation process provides 11 snapshot points for each project, which are at: first trivial introductory commit, 10% commit, 20% commit, 30% commit, 40% commit, 50% commit, 60% commit, 70% commit, 80% commit, 90% commit and latest commit. As module growth is a predicted phenomenon in the software development lifecycle [45], [46], [47], we measure the percentage of trivial dependent files to all files across a project's TDDT not the raw number.

Figure 7 shows box-plots of the percentage of the number of trivial dependent files in all the studied projects in our dataset for the 11 snapshot points in the projects' TDDT. Here, we observe that the percentage of the number of trivial dependent files remain almost constant over time with approximately median percent of trivial dependent files equal to 20%. These results reflect the centrality of and developer's reliance on these trivial packages in software projects.

We further investigate the percentage of trivial packages' API calls in trivial dependent files throughout the concerned project's TDDT. Table 9 shows the percentage of package's API calls distribution in these files for each project across its TDDT. Once again, to put our analysis in perspective, for every TDDT segment, the table shows the percentage of trivial packages (TP) and non-trivial packages's API calls.

From Table 9, we observe that the percentage of trivial package's (TP) API calls is higher than that of non-trivial package's (NTP) API call at each snapshot point in the projects development timespan. For example, at 30%'s TDDT, we see that trivial packages' API calls is higher (with mean=30.5 and median = 16.7) that the percentage of API calls for the non-trivial packages (with mean = 16.3 and median = 9.1). We see similar results at the late of the development lifespan of the studied projects. As the table shows at 90%'s TDDT, we see that with 30.3/16.7 mean/median of API calls for trivial packages is higher than the ones for the non-trivial packages (15.8/8.3).

To examine whether the results are statistically significant, we perform the Wilcoxon rank-sum test and the Cliff's Delta effect size test on the data from each segment. The last two rows of Table 9 shows p-value and the effect size between the percentage of trivial and non-trivial packages' API calls for every TDDT. From Table 9, we see that these results are statistically significant and have small effect sizes in all the snapshot points. For example, at 30% TDDT, we found that the different between the percentage of the API calls for trivial and non-trivial packages are statically significant (p-value = <2.2e-16) and the effect size is small. This analysis shows that the percentage of API calls for trivial packages within trivial dependent files remains higher throughout the development timespan of the concerned software projects.

4.2 Developers' Perspective

Since our analysis has been quantitative in nature, in this section we want to triangulate our findings and understand the developers' perspective of our findings. Thus, we conducted a user study where we sent a summary of our findings and a version of this paper to JavaScript

2. In our study, the current snapshot of a project refers to the date when we collected project in our dataset.

Table 9: The statistical summary of the distribution of external package API call percentage in JavaScript files throughout project's development lifespan. The table shows the distribution for trivial packages (**TP**) and non-trivial packages (**NPT**).

Segments	Intro		10%		20%		30%		40%		50%		60%		70%		80%		90%	
	TP	NTP	TP	NTP	TP	NTP	TP	NTP	TP	NTP	TP	NTP	TP	NTP	TP	NTP	TP	NTP	TP	NTP
Min.	0.1	0.1	0.1	0.1	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Median	16.7	9.1	16.7	9.1	16.7	9.1	16.7	9.1	16.7	9.1	16.7	8.6	16.7	8.7	16.7	8.3	16.7	8.3	16.7	8.3
Mean	30.1	16.4	30.4	16.5	30.0	16.4	30.5	16.3	30.2	16.3	30.7	15.9	30.7	16.0	30.8	15.9	30.8	15.9	30.3	15.8
Max.	100	96.6	100	96.5	100	96.6	100	96.8	100	96.9	100	97.1	100	96.9	100	97.3	100	97.5	100	97.6
p**-value**	<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16	
Cliff's d	0.2434		0.2540		0.2453		0.2626		0.2589		0.2641		0.2635		0.2661		0.2760		0.2543	

developers. The email's main goal was to inquire about the developers' thoughts about our findings (e.g., anything that is surprising, interesting, obvious, etc.).

We received five responses (P1 to P5), and we carefully read the responses. We observe that all the surveyed developers found our results to be interesting. For example, developers P1 stated that *"I did not expect that the trivial packages play a key role in the central parts of the projects and the npm ecosystem."* and P4 stated *"This is kind of crazy to me (good kind of crazy), generally as software engineers, we will depend on a package because it accomplishes something difficult to do and we don't want to put in the effort and maintain it ourselves."* Interestingly, one developer P2 pointed out the importance of our study and said that *"I am glad that the study raises awareness for the importance of these trivial packages, since they might sometimes be overlooked from a quality assurance standpoint because they seem small and uncomplicated."*

Some of the developers in our study also pointed out some root causes of these trivial packages. For example, developer P2 stated, *"this study shows solid proof on how not having a proper standard library in a language like JS can force users to adopt alternative strategies (depending on many trivial packages) that might cause other issues in the long term."* Another developer P4 highlighted the direct implication of this study to ecosystem maintainers and project developers and stated *"You brought up a good point about how ecosystem maintainers should specify that a dependency is trivial. Of course this is good advice now that you have done this study and presented your findings; otherwise software engineers don't think this way. When I am evaluating whether or not to include a dependency, the fact that it's trivial is not something I consider. I do consider things like the popularity in the community (this gives a measure of how proven it is), and how active of a project is it (is it a one shot release? is it actively maintained?)."*

4.3 Implications of Our Findings

Our study has a number of implications for trivial package developers, for developers of the projects that depend on trivial packages, and for the npm ecosystem.

Implications for developers of trivial packages: In our study, we found that trivial packages play a central role in both, the JavaScript projects that depend on them and for the npm ecosystem. Since prior work showed that trivial packages also introduce some negative side effects (e.g., they may lack proper tests and may introduce significant dependency overhead [3]), we recommend that trivial package developers put more effort to keep their trivial packages well-maintained and up-to-date. Although developers may think that a trivial package need not much maintenance after its creation, what our study shows is quite the contrary. Additionally, developers should be careful when publishing these trivial packages since our findings show that they can be heavily dependent on and can cause added complexity to the ecosystem.

Implications for developers of the projects that depend on trivial packages: First of all, our main implication for the JavaScript developers who depend on trivial packages is to not overlook the importance of such packages. Our study shows that, in many cases, they are critical to the developers' projects and to many of the other projects that depend on them. Hence, proper updating practices should be followed, even for these trivial packages. Also, developers should carefully examine such packages before depending on them since they may prove problematic if one depends on a poorly developed or maintained trivial package.

Moreover, our results show that JavaScript projects depend on trivial packages throughout the entire lifespan of the projects, which means that they are not just used early on in the project. This finding has two main implications. First, developers should apply a systematic approach when selecting an external package that they want to add and make sure to consider whether the package is trivial. Second, developers should consider some code enhancement techniques to limit the use of trivial packages. For example, developers should use refactoring and migration techniques to reduce dependencies (even dependencies on trivial packages) since they pose potential points of failure for their JavaScript project. The case can be made that eliminating a dependency on a trivial package may be easier to do than on non-trivial packages.

Implications for the npm ecosystem: Our results show trivial packages play a central role in the npm ecosystems and in some cases removing one trivial package can affect approximately 29% of the packages in the ecosystems. We believe that ecosystem maintainers should treat trivial packages with more care and perhaps provide some guidelines for their inclusion in the ecosystem. For example, limits may be put on the amount of dependencies that one trivial package may have. One way to address this issue is to introduce a JavaScript standard library that encompasses many of these trivial packages. The need for such a richer standard library is clearly evident from our finding that many of the trivial packages provide important and essential functionality that would generally be provided in a standard library of other languages.

In addition, developers should give as much attention to trivial packages as they do to larger and more complex packages. Also, ecosystem maintainers should provide a mechanism to warn developers about these trivial packages. For example, developers should be shown the size and

complexity of the trivial packages during the process when developers are looking at a npm package to use.

5 Related Work

In this section, we discuss work related to our study, which is mainly related to software ecosystems. The software projects that belong to the same software ecosystem has been a research interest lately. Several studies examine software ecosystems to understand their characteristics and evolution (e.g., [48], [49], [50], [51], [52], [53]).

Recently, Abdalkareem *et al.* [3] studied an emerging code reuse practice in the form of small packages, trivial packages, in the npm ecosystem. Abdalkareem *et al.* [3] studied various aspects regarding trivial packages. They first defined the size and complexity of these packages and we adopt this definition in our study. Whereas their study was conducted upon understanding why developers use trivial packages, our study examines the importance of these trivial packages in software projects.

Several other studies examined direct and transitive dependencies of software projects. Wittern *et al.* [54] examined packages in npm ecosystem and they observed that 32.5% of the packages have 6 or more dependencies. Moreover, 27.5% of the packages in npm are core packages as they are largely dependent on. Fard *et al.* [55], in their study, evaluated changeability in npm projects and they showed that the average number of dependencies in these projects is 6 and the number is always in the growing trend. Kikas *et al.* analyzed the dependency network structure and evolution of the JavaScript, Ruby, and Rust ecosystems and showed that the number of transitive dependencies is 10 times higher than the number of direct dependencies and this scenario is growing exponentially [56]. Recently, Zimmermann *et al.* [57] systematically examine dependencies between packages, the maintainers responsible for packages in the npm while focusing on security issues. Their results show that individual packages could impact large parts of the npm ecosystem. They also reported that a very small number of developers are responsible for a large number of npm packages. In our study, we also see that direct dependencies are only the tip of the iceberg, whereas indirect dependencies make up the largest portion of a package dependency network. In our study, we have used the idea of the Technical Bus Factor where we measure the percentage of the ecosystems that may impact by removing one trivial package. In our analyzed dataset of software projects, we found 10,507 distinct packages as direct dependencies to these projects whereas the package dependency network, which has direct and transitive dependencies of these software projects, has 32,319 packages.

Researchers have also investigated developers' rationale behind selecting package for his/her software project. Surprisingly, Haenni *et al.* found that developers generally do not apply any logical reasoning while selecting the packages, they just use them to accomplish their task. Abdalkareem *et al.* [3] found that developers have biased perception about trivial packages, developers think that these packages are well tested. Moreover, after including third-party packages, developers are often too reluctant to updates their dependencies, which improves functionalities and fixes security issues or bugs, of these packages. Kula *et al.* [58] observed that 81.5% of their studied projects have outdated dependencies, although these projects heavily depend on external packages. Their interviewing of developers reveals that they are often unaware of the security vulnerabilities of underlying dependencies and therefore they perceive updating dependencies not a necessity but additional work. The study of Wittern *et al.* [54] shows us that the package version number is not a good predictor of a package's maturity. Therefore, to assist developers in updating dependencies, evaluating four software packaging ecosystems (Cargo, npm, Packagist and Rubygems), Decan *et al.* [59] proposed an evaluation based on the "wisdom of the crowds" principle to select appropriate semantic versioning constraints for their dependencies. These types of ecosystem-wide studies help to clarify various general misconceptions and mitigate bad practices in ecosystems.

Other studies examine the API usage of external packages. Mileva *et al.* [60] studied API usage patterns of external libraries to examine the popularity of external package APIs'. They used this popularity metric to determine if a package is successful or not. In addition, Holmes *et al.* [29] quantitatively analyzed how APIs are used. They consider the frequency of API use as popularity and importance of that API. Similar to these studies, we determine the importance of an external package by analyzing the percentage of its API calls in the files that depend upon those packages.

Overall, our study examines software projects that depend on at least one trivial package from the npm ecosystem. So, our study is focused on the characteristics of software projects that adhere to the same environment. This categorization helps us understand the ecosystem better and helps adhere to good practices and mitigate bad practices ecosystem-wide.

6 Threats to validity

In this section, we discuss the threats of validity related to our study.

6.1 Construct validity

Construct validity considers the relationship between theory and observation, in case the measured variables do not measure the actual factors.

In our study, we used several in-house and state-of-the-art tools and techniques. First, to identify the actually used packages in the JavaScript projects in our dataset, we used the depchecker [20] tool to extract file-level dependencies. Our results may be limited by the accuracy of this tool. To validate the accuracy of the depchecker tool. We run the tool on randomly selected five JavaScript projects from our dataset. Then, the first two authors manually examine the results of the depchecker tool thorough manually examining the output of depchecker tool. We find that in all the examined projects the tool correctly identified and reported the actually used packages.

To calculate the degree centrality of trivial and non-trivial dependent files in each project in our dataset, we generate a file-level dependency graph representation of files in every project in our dataset. To do so, we use the madge tool to generate file-level dependency graphs [25]. Hence, our analysis is limited by the accuracy of the madge tool. To validate the accuracy of the madge tool, we performed

an experiment to examine the precision of the madge tool. To do so, we selected five JavaScript projects from our dataset, and then we ran the madge tool on them. Then the first two authors manually examine the generated file-level dependency graph for every project and trace the generated links from the file-level dependency graph to the projects' source code. Table 10 shows the distribution of the number of files in the manually examined JavaScript projects. Our results show that all the examined links in the generated file-level dependency graphs exist in the source code of the examined JavaScript projects with a precision of 100% since we only have the cases that the madge tool detects.

Table 10: The distribution of the number of files in the five manually examined projects.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
15.0	16.0	23.0	27.6	38.0	46.0

In addition, to examine whether trivial packages are used in central parts of JavaScript projects, we calculated the degree centrality score for each file of a project in our dataset. However, degree centrality is not the only method to measure centralities. Thus, using other methods of calculating centralities (e.g., betweenness centrality [61]) may provide different results.

To answer the second research question and to extract and measure all the occurrences of external package API calls in JavaScript files, we use the Understand tool [19] that is a commonly used source analysis tool in both industry and research [30], [31], [32]. Hence, we are limited by the accuracy of the Understand tool. To validate the accuracy of the Understand tool, we performed two complementary experiments. We first performed an experiment to identify the type of API calls that the Understand tool can detect. To do so, we collect a list of possible ways of making an API call in the JavaScript programming language. We collected ten different types of API calls in JavaScript of which only eight can be detected via static source code analysis (Appendix A shows the list of API call examples in JavaScript). After that, we run the Understand tool on all the collected API calls in our examples. Then, we manually examine the result of the Understand tool to see whether the Understand tool detects all of the collected API calls or not. We determine that it is sufficient to run the Understand tool on each type of API call once, because our goal is to validate the ability of the tool to detect different API call types in Javascript. Since detecting an API call once means that it can be detected all the time, we find it sufficient to perform this detection on one instance of each API call type.

Our result shows that the Understand tool is able to detect all eight of the target API calls. The only two expectations that the tool does not detect are API calls that are associated with the function eval and constructor. In JavaScrip, the functions eval and constructor have a string argument, which is parsed and executed at runtime. These two cases can only be detected at runtime and cannot be detected via static source code analysis.

Second, we performed an experiment to evaluate the accuracy of the Understand tool in detecting API calls. To do so, we randomly selected five JavaScript projects from our dataset and ran the Understand tool. Then, for each JavaScript file in the selected projects, we extracted the API calls. After that the first two authors manually examine the API calls in the source code of each file. Our results show that in all the cases that we examine the Understand tool is able to detect the API calls. It is also important to notice that in all the examined projects, we did not find any use of API call through a string argument (e.g., eval function).

We also use the networkx [26] tool to generate the dependency graph of files of every JavaScript project. Again, our graph dependency network analysis may influence the accuracy of the generated graph. To alleviate these issues, we manually examine the generated call graphs for five projects in our dataset and found that these graphs represent the dependency structure between files in these projects.

To answer our second research question, we only captured the direct usage of external packages in our static code analysis. For example, a package "X" is imported (e.g require statement) and assigned it to a variable "a" and later "a" is assigned to another variable "b". We only tracked the external package usage with variable "a" and did not track "b". We decide to examine the direct usage of these packages for two main reasons. First, this type of transitive assignment of a variable is very rare in JavaScript code as other work shows [62]. This is why we believe that this shortcoming does not significantly impact our findings. Second, if we miss some of the usages of external packages, we missed both trivial and non-trivial packages. As we contrast trivial and non-trivial package usage, this effect will not affect the result of the comparison.

In our analysis, we resort to using the npm-pack command [17] to resolve the semantic version and download the appropriate 'tar' file that contains the source code of the package for each dependency-version pair at the time of our analysis. Thus, dependencies could be different if the analysis is done at a later time.

In our selection process of the JavaScript projects that are hosted on GitHub, we filtered out npm packages that may also exist on GitHub [15]. To do so, we relied on the metadata provided by the GHTorrent dataset [14] to cross-check the list of URLs. Thus, our selection of JavaScript projects heavily depends on the correctness of the projects URLs listed in the GHTorrent dataset. To answer RQ3, we constructed a dependency network graph of all the direct and indirect dependencies packages used in our studied projects. This dependency network graph may not represent the entire packages in the npm ecosystems.

6.2 External validity

In this subsection, we discuss the generalizability of our findings. Our dataset only consists of JavaScript projects, which use npm as their package manager, hence our findings may not hold for projects written in other programming languages or use different package manager. However, npm mainly supports JavaScript projects and it is one of the largest and most rapidly growing software ecosystems [63]. In addition, our dataset that is used in our study present only open source project hosted on GitHub that may do not reflect proprietary projects. Also, our initial dataset size is 15,254 JavaScript projects that use the npm package manager, which may not represent the whole population of JavaScript projects.

7 Conclusion

Code reuse in the form of small/trivial packages became prevalent in software development [3], [64]. We observe that these trivial packages, being small in size and complexity, provide various functionalities ranging from string manipulation to security. Thus it is important to understand whether these packages are trivially used or their usage in software projects transcends their triviality. In this paper, we empirically examine trivial packages relative importance and their use cases from two point of views; from the projects usage and ecosystem usage. to do so, we analyze a large dataset of open-source JavaScript projects that depend on at least on trivial package.

We observe that trivial packages are used in important part of the examined software projects compared to non-trivial packages. Our results show that trivial dependent files have on median 0.022 degree centrality value while it is 0.001 for non-trivial dependent files. We also, found that trivial packages have a higher percentage of total API calls of JavaScript files (11.76% and 7.69% for trivial and non-trivial packages). As for the ecosystem usage, we examine the relative importance of trivial packages in the ecosystem they belong to where we analyze the dependency graph of the direct and transitive dependencies of software projects in our dataset. We observe that trivial packages are highly dependent upon packages in the npm ecosystem, which makes trivial packages salient in the ecosystem. In some case removing one trivial package from the npm ecosystem could affect up to 29% of the whole npm ecosystem.

We believe that there are several possible directions for future work based on our findings. First, we would like to develop an advanced technique to detect and evaluate the quality of trivial package in an ecosystem since our results reveal that trivial packages play a key role in the ecosystem. Second, we want to devise an automatic approach to identify trivial package so developers can be aware that the packages that they use are trivial. Finally, since our study examines only the importance of JavaScript packages, we would like to investigate the notion of triviality in other/more software ecosystems.

References

[1] S. Wagner and E. Murphy-Hill, *Factors That Influence Productivity: A Checklist*, Berkeley, CA, 2019", publisher="Apress, pp. 69–84.

[2] E. Murphy-Hill, C. Jaspan, C. Sadowski, D. Shepherd, M. Phillips, C. Winter, A. Knight, E. Smith, and M. Jorde, "What predicts software developers' productivity?" *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[3] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "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*, ser. ESEC/FSE 2017. ACM, 2017, pp. 385–395.

[4] R. Cox, "Surviving software dependencies," *Commun. ACM*, vol. 62, no. 9, pp. 36–43, Aug. 2019.

[5] W. C. Lim, "Effects of reuse on quality, productivity, and economics," *IEEE Software*, vol. 11, pp. 23–30, 1994.

[6] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 559–563.

[7] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, May 2018, pp. 181–191.

[8] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe, "Where does this code come from and where does it go? - integrated code history tracker for open source systems -," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 331–341. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337263

[9] F. Macdomald, "A programmer almost broke the internet last week by deleting 11 lines of code - sciencealert," http://www.sciencealert.com/how-a-programmer-almost-broke-the-internet-by-deleting-11-lines-of-code, March 2016, (accessed on 06/03/2016).

[10] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "On the impact of micro-packages: An empirical study of the npm javascript ecosystem," 2017.

[11] npm search, "escape-string-regexp - npm," https://www.npmjs.com/package/escape-string-regexp, October 2019, (accessed on 10/02/2019).

[12] M. A. R. Chowdhury, R. Abdalkareem, E. Shihab, and B. Adams, "On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages," Dec. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.4019236

[13] "The ghtorrent project," http://ghtorrent.org/, (Accessed on 02/18/2019).

[14] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. ACM, 2014, p. 384387.

[15] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the diversity of software package popularity metrics: An empirical study of npm," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER 2019, 2019, pp. 589–593.

[16] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. ACM, 2014, pp. 92–101.

[17] "npm-pack — npm documentation," https://docs.npmjs.com/cli/pack.html, (Accessed on 02/17/2019).

[18] "npm-registry — npm documentation," https://docs.npmjs.com/misc/registry, (Accessed on 02/17/2019).

[19] "Understand™static code analysis tool," https://scitools.com/, (Accessed on 02/17/2019).

[20] J. Li and D. Lukic, "depcheck - npm," https://www.npmjs.com/package/depcheck, (Accessed on 02/17/2019).

[21] S. P. Borgatti, "Centrality and network flow," *Social networks*, vol. 27, no. 1, pp. 55–71, 2005.

[22] S. White and P. Smyth, "Algorithms for estimating relative importance in networks," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03. ACM, 2003, pp. 266–275.

[23] F. Cadini, E. Zio, and C.-A. Petrescu, "Using centrality measures to rank the importance of the components of a complex network infrastructure," in *Proceedings of the Critical Information Infrastructure Security*. Springer Berlin Heidelberg, 2009, pp. 155–167.

[24] X. Qi, E. Fuller, Q. Wu, Y. Wu, and C.-Q. Zhang, "Laplacian centrality: A new centrality measure for weighted networks," *Information Sciences*, vol. 194, pp. 240 – 253, 2012.

[25] "Madge- developer tool for generating a visual graph of your module dependencies," https://www.npmjs.com/package/madge, (Accessed on 02/17/2019).

[26] "Networkx - network graph analysis," https://networkx.github.io/, (Accessed on 02/17/2019).

[27] "wilcox.test function — r documentation," https://www.rdocumentation.org/packages/stats/versions/3.5.1/topics/wilcox.test, (Accessed on 02/17/2019).

[28] "cliff.delta function — r documentation," https://www.rdocumentation.org/packages/effsize/versions/0.6.4/topics/cliff.delta, (Accessed on 02/17/2019).

[29] R. Holmes and R. J. Walker, "Informing eclipse api production and consumption," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '07. ACM, 2007, pp. 70–74.

[30] M. T. Rahman, P. C. Rigby, and E. Shihab, "The modular and feature toggle architectures of google chrome," *Empirical Software Engineering*, vol. 24, no. 2, p. 826853, Apr. 2019.

[31] M. Castelluccio, L. An, and F. Khomh, "An empirical study of patch uplift in rapid release development pipelines," *Empirical Software Engineering*, vol. 24, no. 5, pp. 3008–3044, 2019.

[32] M. Ahasanuzzaman, S. Hassan, and A. E. Hassan, "Studying ad library integration strategies of top free-to-download apps," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[33] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 78–88.

[34] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, June 2013.

[35] "npm-ls | npm documentation," https://docs.npmjs.com/cli/ls.html, (Accessed on 04/21/2019).

[36] "Pagerank - wikipedia," https://en.wikipedia.org/wiki/PageRank, (Accessed on 02/17/2019).

[37] T. Mens, "An ecosystemic and socio-technical view on software maintenance and evolution," in *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, vol. 00. IEEE, 2016, pp. 1–8.

[38] "process-nextick-args - npm," https://www.npmjs.com/package/process-nextick-args, (Accessed on 07/24/2019).

[39] "isarray - npm," https://www.npmjs.com/package/isarray, (Accessed on 07/24/2019).

[40] "sindresorhus/ponyfill: like polyfill but with pony pureness," https://github.com/sindresorhus/ponyfill, (Accessed on 07/05/2019).

[41] "npms," https://npms.io/search?q=keywords%3Aponyfill, (Accessed on 07/05/2019).

[42] "npm," https://www.npmjs.com/~sindresorhus, (Accessed on 07/05/2019).

[43] "The web's scaffolding tool for modern webapps — yeoman," https://yeoman.io/, (Accessed on 07/05/2019).

[44] "sindresorhus/awesome: awesome lists about all kinds of interesting topics," https://github.com/sindresorhus/awesome#readme, (Accessed on 07/05/2019).

[45] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[46] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *2009 IEEE International Conference on Software Maintenance*, ser. ICSME 09, Sep. 2009, pp. 51–60.

[47] Godfrey and Qiang Tu, "Evolution in open source software: a case study," in *Proceedings 2000 International Conference on Software Maintenance*, ser. ICSME 2000. IEEE, Oct 2000, pp. 131–142.

[48] R. Bloemen, C. Amrit, S. Kuhlmann, and G. Ordóñez Matamoros, "Gentoo package dependencies over time," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. ACM, 2014, pp. 404–407.

[49] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. IEEE Computer Society, 2013, pp. 280–289.

[50] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When github meets cran: An analysis of inter-repository package dependency problems," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE Computer Society, 2016, pp. 493–504.

[51] K. Manikas, "Revisiting software ecosystems research: A longitudinal literature study," *Journal of Systems and Software*.

[52] D. M. German, B. Adams, and A. E. Hassan, "The evolution of the r software ecosystem," in *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 243–252.

[53] J. "Kabbedijk and S. Jansen, ""steering insight: An exploration of the ruby software ecosystem"," in *"Software Business"*. "Springer Berlin Heidelberg", "2011", pp. "44–55".

[54] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 351–361.

[55] A. M. Fard and A. Mesbah, "Javascript: The (un)covered parts," *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 230–240, 2017.

[56] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. IEEE Press, 2017, pp. 102–112.

[57] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC19. USA: USENIX Association, 2019, p. 9951010.

[58] R. G. Kula, D. M. Germán, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? - an empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5

[59] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[60] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining api popularity," in *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*, ser. TAIC PART'10. Springer-Verlag, 2010, pp. 173–180.

[61] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, pp. 35–41, 1977.

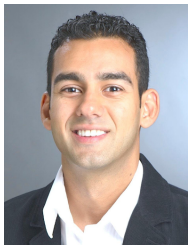
[62] A. Feldthaus, M. Schfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE)*. ACM, 2013, pp. 752–761.

[63] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Softw. Engg.*, vol. 24, no. 1, pp. 381–416, 2019.

[64] R. Abdalkareem, "Reasons and drawbacks of using trivial npm packages: The developers' perspective," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. ACM, 2017, pp. 1062–1064.

Md Atique Reza Chowdhury is a M.Sc. in Department of Computer Science and Software Engineering at Concordia University, Montreal. His research interests include mining software repositories, and analysis of software ecosystems.

Rabe Abdalkareem is a postdoctoral fellow in the Software Analysis and Intelligence Lab (SAIL) at Queens University, Canada. He received his Ph.D. in Computer Science and Software Engineering from Concordia University, Montreal, Canada. His research investigates how the adoption of crowdsourced knowledge affects software development and maintenance. Abdalkareem received his masters in applied Computer Science from Concordia University. His work has been published at premier venues such as FSE, MSR, ICSME and MobileSoft, as well as in major journals such as TSE, IEEE Software, EMSE and IST. Contact him at abdrabe@gmail.com; https://rabeabdalkareem.github.io/.



Emad Shihab is an associate professor in the Department of Computer Science and Software Engineering at Concordia University. He received his PhD from Queens University. Dr. Shihabs research interests are in Software Quality Assurance, Mining Software Repositories, Technical Debt, Mobile Applications and Software Architecture. He worked as a software research intern at Research In Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. Dr. Shihab is a member of the IEEE and ACM. More information can be found at http://das.encs.concordia.ca.

Bram Adams is an associate professor at Polytechnique Montreal, where he heads the Lab on Maintenance, Construction, and Intelligence of Software. His research interests include release engineering in general, as well as software integration, software build systems, and infrastructure as code. Adams obtained his PhD in computer science engineering from Ghent University. He is a steering committee member of the International Workshop on Release Engineering (RELENG) and program co-chair of SCAM 2013, SANER 2015, ICSME 2016 and MSR 2019.

APPENDIX A
LIST OF TEN DIFFERENT API CALLS IN JAVASCRIPT

In this appendix, we present a list of ten possible ways of making an API call in the JavaScript programming language. We collected ten different types of API calls in JavaScript, of which only eight can be detected via the Understand tool.

A.1 Function declaration (Detected)

```
// function_declaration.js

function hello() {
console.log('hello')
}

hello()
```

A.2 Function expression (Detected)

```
// function_expression.js
var hello = function() {
console.log('hello')
}

hello()
```

A.3 Function as constructor (Not detected)

```
// function_constructor.js
var hello = new Function("console.log('hello')")

hello()
```

A.4 Function in eval (Not detected)

```
// eval.js

eval("function hello (){ console.log('hello') }")

hello()
```

A.5 Object method (Detected)

```
// object_method.js

var greeter = {
hello: function () {
console.log('hello')
}
}

greeter.hello()
```

A.6 Function method (Detected)

```
// function_method.js

function greeter() {
this.hello = function () {
console.log('hello')
}
}

new greeter().hello()
```

A.7 As a constructor (Detected)

```
// constructor.js

function hello() {
console.log('hello')
}

new hello()
```

A.8 Via call() (Detected)

```
// function_call.js

function greeter() {
this.hello()
}

var x = new function(){
this.hello = function(){
console.log('hello')
}
}

greeter.call(x)
```

A.9 Via apply() (Detected)

```
// function_apply.js

function greeter() {
this.hello()
}

var x = new function(){
this.hello = function(){
console.log('hello')
}
}
greeter.apply(x)
```

A.10 Self-Invoking functions

```
// self_invoking.js

(function() {
console.log('hello')
})()
```