Software Ecosystem Call Graph for Dependency Management

Joseph Hejderup
Delft University of Technology
The Netherlands
j.i.hejderup@tudelft.nl

Arie van Deursen
Delft University of Technology
The Netherlands
arie.vandeursen@tudelft.nl

Georgios Gousios
Delft University of Technology
The Netherlands
g.gousios@tudelft.nl

ABSTRACT

A popular form of software reuse is the use of open source software libraries hosted on centralized code repositories, such as Maven or *npm*. Developers only need to declare dependencies to external libraries, and automated tools make them available to the workspace of the project. Recent incidents, such as the Equifax data breach and the *leftpad* package removal, demonstrate the difficulty in assessing the severity, impact and spread of bugs in dependency networks. While dependency checkers are being adapted as a counter measure, they only provide indicative information. To remedy this situation, we propose a fine-grained dependency network that goes beyond packages and into call graphs. The result is a versioned ecosystem-level call graph. In this paper, we outline the process to construct the proposed graph and present a preliminary evaluation of a security issue from a core package to an affected client application.

ACM Reference Format:
Joseph Hejderup, Arie van Deursen, and Georgios Gousios. 2018. Software Ecosystem Call Graph for Dependency Management. In *Proceedings of 40th International Conference on Software Engineering: New Ideas and Emerging Results Track, Gothenburg, Sweden, May 27-June 3 2018 (ICSE-NIER'18),* 4 pages.
https://doi.org/10.1145/3183399.3183417

1 INTRODUCTION

Software engineers reuse code to reduce development and maintenance costs. A popular form of software reuse is the use of open-source software (OSS) libraries, hosted on centralized code repositories, such as `Maven`[1] or `npm`.[2] In such settings, developers specify *dependencies* to external library versions in a textual file, that is then committed to the repository of the *client program*. Automated programs, typically *package managers*, resolve the dependency descriptions and connect to the central repositories to download the specific library versions that are required to build the client program. Library names and versions often follow de-facto conventions, such as *semantic versioning*.

Several implications may arise from the fact that programs and libraries can have dependencies on other libraries, and that those dependencies are not resolved in a well-defined manner. Increasingly, libraries are being used as building blocks for creating other libraries, leading to highly interconnected *ecosystems* [4, 7]. The interconnections form a graph, in which the nodes are versioned libraries and the edges are dependencies on the libraries. The structural properties of those graphs can significantly affect the *functionality* of thousands of end-user projects [6]. Moreover, including arbitrary code from an online repository induces *trust* and *security* implications; how can developers ensure that the imported code contains no security holes? How can they know when a security issue discovered in a *transitive* dependency requires an update? Dependency networks also present challenges to library maintainers: how can they assess the direct or transitive impact of their changes?

In the recent years, we have witnessed dependency network failures with severe implications on client programs:

- A dispute over a library name in the `npm` ecosystem led to the removal of a library called *leftpad*. The package removal further lead to the collapse of thousands of libraries which directly depended on *leftpad*, and hence a major disruption for client programs. After the *leftpad* incident, a study [6] estimated that there exist libraries whose removal can affect more that 30% of the core components of the network.
- A company named *Equifax* leaked over 100.000 credit card records due to a dependency that was not updated. The compromised systems included a vulnerable version of the Apache Struts library, whose update was postponed as the Equifax security team erroneously underestimated the impact of the bug on their codebase.[3]
- Malicious developers uploaded to the Python package manager (PyPI) libraries whose name was deliberately misspelled, being almost identical to the original libraries (e.g., `urllib` instead of `urllib3`). The intention was to steal information from client applications of developers who had accidentally mistyped the library name in the dependency file.

Recent research has been focused on the analysis of the evolution of code repositories and how libraries are growing together in a shared environment [3, 4, 7]. To study ecosystems, developers typically build dependency graphs, in which nodes represent either libraries or library versions. To represent a dependency (i.e. creating an edge), researchers emulate the version resolution algorithm of the original package manager. While this model is useful for initial evaluations of dependency networks, it can only provide partial information due to the following limitations:

(1) The dependency relationship in the network is on a version-basis (e.g library A $v1.2.3$ depends on library B $v2.3.4$). Reasoning about how a library can influence other connected libraries on a

[1] https://search.maven.org/
[2] https://www.npmjs.com/

[3] https://blogs.apache.org/foundation/entry/apache-struts-statement-on-equifax

ICSE-NIER'18, May 27-June 3 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5662-6/18/05…$15.00
https://doi.org/10.1145/3183399.3183417

version-basis such as bug propagation is limited. (2) A dependency on a library does not necessarily mean that the code in that library is actually used. Providing developers with *actionable* information such as security alerts on dependencies requires further analysis of dependency relationships. (3) Dependency networks do not enable developers (or researchers) to perform change impact analysis beyond a single library; this leads to lost opportunities of evaluating problems at the ecosystem level.

In our work, we propose to *extend dependency networks with call graph information*, within and across dependencies, thereby constructing ecosystem-wide *dependency network call graphs*. This takes into account how libraries are interconnected at the source code level.

Our vision is the following: (1) Construct a dependency network at the function-level granularity. (2) Evaluate the dynamics of changes made to libraries in a dependency network from a program analysis perspective. (3) Study and evaluate historical changes in a dependency network.

In the following sections, we outline concepts of our call graph based dependency network and highlight how it can enable a fine-grained impact analysis assessment.

2 CALL GRAPH BASED DEPENDENCY NETWORK

Most package managers for OSS libraries use a variation of semantic versioning to specify dependency versions. Semantic versioning allows developers to specify dependencies, not only as an exact version but also a version range. The resolution of version ranges to exact versions is *time-dependent*; the package manager resolves the latest version available at the package repository at the time the resolution was initiated. This complicates precise retrospective studies of dependency networks and makes their results fragile. As an example, consider a library A, with two versions: $v1.2$ released in Oct 2014 and $v1.3$ released in Oct 2016. A library B depends on version $1.*$ of library A. If we create the dependency graph of the package ecosystem today, we would only resolve the dependency to $A\ v1.3$, missing two years worth of time where the correct dependency would be $A\ v1.2$.

Consequently, we need a fine-grained dependency network. For this, we can exploit the fact that the vast majority of open source libraries included in dependency networks are developed on GitHub. Instead of relying on aggregated metadata from package managers, such as the data provided by the `libraries.io` service, we can analyze the commits on the dependency specification files. Using these files, we can construct dependency networks with more details. Following the example above, if the repository exporting library B has received a `commit` c at any time between Oct 2014 and Oct 2016, then we could resolve A to $v1.2$ for the version created in c. Moreover, relying on GitHub for constructing dependency networks will enable us to include client programs in our analysis, thereby extending the impact of our envisioned analyses.

After obtaining high resolution dependency networks, we need to construct call graphs for each library version that we include in our graph. Creating call graphs can be done either with static analysis, where possible executions are determined from analyzing the source code or through dynamic analysis, where probes are the

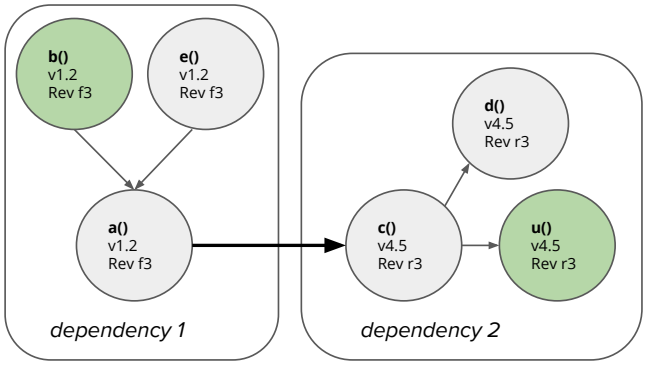

Figure 1: Dependency network call graph

method invocations recorded at runtime. Our only requirement is that the call graph construction will be able to (statically) construct call graphs that extend beyond a single project to the transitive set of dependencies the project specifies. This requirement is akin to the linking process in compiled languages, and is already supported by many tools.

Next, we need to map the call graph on the dependency network. There are two ways to do this: i) include the generated call graph as an attribute to a dependency network node, or ii) decorate the call graph nodes with metadata about the dependency (e.g., the dependency name and version). We choose the second option, as the graph nodes are functions and represents our end goal of being able to perform impact analysis at an ecosystem level.

By following the steps above, we arrive to the definition of our dependency network call graph as follows:

Definition. *A **dependency network call graph** for an ecosystem is a directed and immutable graph $G = (V, C)$ where:*

1. *V is a set of versioned functions. Each $v \in V$ is a 3-tuple $< id, v, c >$, where id is a fully qualified function name, v is the version of the library and c is the commit.*
2. *E is a set of edges that connects functions. Each $(v_1, v_2) \in E$ represents a function call from v_1 to v_2.*

Although the *dependency network* shares some similarities with the work of Hejderup and Kikas et al [5, 6], it differs in that the network is a large interconnected call graph and the versioning is annotated at the function call level. Figure 1 illustrates a simple call graph based dependency network where *dependency 1* depends one *dependency 2*. Inside each dependency block, the nodes represent versioned functions with full function identifier, version and `commit` revision. The edges in the network can be classified into internal and external calls. An external call is made from the `a()` node in dependency 1 to the `c()` node in dependency 2. The process to decide and resolve an external function call into the correct versioned one is not trivial and is explained in the following subsection.

2.1 Network Construction

The process of constructing the network is shown in Algorithm 1. The initial step is to select commits that include changes to the dependency file of the repository (line 3). A change can be a new release of the library or a change to the list of specified dependencies.

Algorithm 1: Network Construction

```
   Input  : git-based repository
1  G ← ∅;
2  buildNetwork repo
3      C ← filterDependencyCommits(repo);
4      if C ≠ ∅ then
5          foreach c ∈ C do
6              rev ← checkout(c);
7              CG ← constructCG(rev);
8              CG_ann ← annnoteFunctions(CG);
9              if c specifies new library version then
10                 G ← G ∪ getPrevCommitEdges(c, CG_ann);
11             end
12             if c specifies a dependency update then
13                 G ← G ∪ resolveDependencies(c, G, CG_ann);
14             end
15         end
16     end
```

Algorithm 2: Resolving Dependencies

```
   Input  : commit, G, CG_ann
   Output : set of resolved dependency call edges
1  resolveDependencies commit, G, CG_ann
2      E ← ∅;
3      depfile ← getDependencyFile(commit);
4      D ← parse(depfile);
5      if D ≠ ∅ then
6          foreach d ∈ D do
7              G_name ← sliceByName(G, d.name);
8              ver ← resolveVersion(d.constraint, c);
9              G_ver ← sliceByVersion(G_name, ver);
10             CG_d ← sliceByCommit(G_ver, commit);
11             E ← E ∪ {getDependencyCalls(CG_d, CG_ann)};
12         end
13     end
14     return E
```

For each of the selected commits, the source code of the library is checked-out when the `commit` was made and a call graph is constructed from the source code (lines 6-7). The functions (i.e., nodes) of the call graph are annotated with information about the `commit`, name and version of the library and then added to the dependency network (line 8). Depending on the type of the change in the processed `commit`, there are two sub-cases:

(1) If the change is a new release of the library, a copy of the edges representing function calls to external libraries in the previous version is added to the graph.
(2) If the change specifies a new dependency, the dependencies need to be re-evaluated and edges from each function in CG_{ann} to functions in external libraries need to be created.

The process to resolve dependencies is presented in Algorithm 2. The dependency file is obtained and parsed from the `commit` (lines 3-4). For each dependency, the existing dependency network is sliced by the dependency name, then sliced further by the version that is resolved by emulating the resolution process in the original package manager. The remaining step is to slice at the `commit` level. The time stamp of the provided `commit` is extracted and the closest `commit` to the time stamp in G_{ver} is selected. The (transitive) call graph for the processed dependency is created and links between the processed dependency and external dependencies are resolved and returned (line 11). After the edges are created, the process repeats until there are no more dependencies to add in the graph.

2.2 Impact Analysis

Impact analysis helps in the determination of the subset of the dependency network that is affected by a given set of changes or bugs. The identified subset allows developers and library maintainers to evaluate the impact within or across dependencies at the function call level. As an example, library maintainers can assess the potential impact of a set of changes in the network before releasing a new version. Further, developers can localize functions or methods in the program that are implicitly affected by a critical bug in a *transitive* dependency. Finally, the `commit` revision in the set of affected function nodes could be extended with using tools such as `git-diff` or `git-log` to track and identify function additions, removals or renames.

The process of identifying the affected nodes in a dependency network is summarized in Algorithm 3. Given the name and version of a library and the set of changed functions, the initial process is to find the corresponding versioned functions of f in the dependency network (line 3). For each versioned function, a reachability analysis is performed that traverses the dependency network for identifying one or more calls to the set of changed functions (line 5). The result of CG_{reach} contains a subset of the impacted (e.g reachable) function calls to one versioned function. The partial impact set CG_{reach} is added to the result in G (line 7). Finally, after all versioned functions are processed, the impact set is returned (line 10).

Algorithm 3: Impact Analysis

```
   Input  : A set of affected f in name & version of library
   Output : Affected slice of the dependency network
1  G ← ∅;
2  impact (name, version, f)
3      F ← findVersionedFunctions(name, version, f);
4      foreach function f_i ∈ F do
5          G_reach ← reachability(f_i);
6          if CG_reach ≠ ∅ then
7              G ← G ∩ CG_reach;
8          end
9      end
10     return G
```

3 INITIAL EVALUATION

The concepts presented in the previous section are implemented in an early prototype in JavaScript and currently process `npm`-based projects. The dependency resolution mechanism in the prototype is based on `npm`'s `semver`[4] library, and call graphs are extracted

[4] https://github.com/npm/node-semver

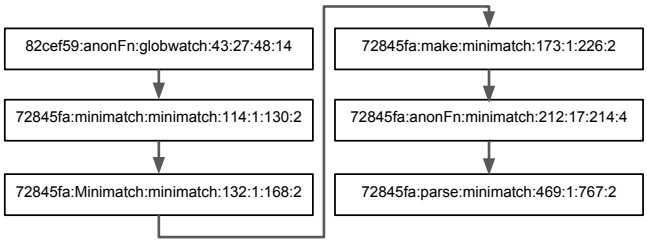

Figure 2: Impacted function calls in `globwatch`

from executing test cases of `npm` packages in `Jalangi`,[5] which is a dynamic analysis framework.

The initial evaluation concentrates on testing the concepts in a small controlled setting before attempting processing at an ecosystem-scale. The first use case evaluates the impact of a security bug in the `npm` ecosystem. Towards this end, we use security advisories from the *Node Security Platform* and evaluate the impact of a bug discovered in the `isaacs/Minimatch` library.[6] The bug is localized in the `parse` function and we obtained the affected commits tagged with a version (e.g, all <=3.0.1) from the repository. Using reversed dependency resolution in place of a call graph network in Algorithm 3, led to the discovery of over 36.000 `npm` packages that directly or indirectly resolve to a vulnerable version of `Minimatch`; our results are publicly available.[7]

We selected an arbitrary package called `globwatch` (v0.0.1) from the results and checked-out the `commit 82cef59` from `airportyh/globwatch`[8]. This `npm` module continuously keeps watching for file changes via a user-defined *glob pattern*. We obtained the call graph by executing the test cases in `Jalangi` and later traversed the call graph to find call-paths that implicitly use the `parse` function in `isaacs/Minimatch` (v0.2.14/`72845fa`). The impacted set resulted in two distinct call-paths, of which, the output of one call path is presented in Figure 2 where the call is executed from top to bottom. Each node in the figure contains the `commit sha` and also the line and the column information of the function in the source code. The line and column number information is retained to precisely identify anonymous function (e.g `anonFn`) bodies in JavaScript.

4 DISCUSSION & CHALLENGES

The sheer size of code repositories and the frequent release of libraries poses many challenges to the construction and maintenance of a *versioned call graph* based dependency network. The concepts presented in Section 2 imply a use of program analysis techniques to construct, infer and traverse call graphs. Performing such an analysis at ecosystem-scale introduces several problems: (1) Obtaining a sound or accurate call graph can be computationally expensive, thus making the construction of the network time consuming. (2) Having an imprecise call graph could potentially lead to false negatives in the impact analysis. (3) The use of `commit` time for resolving version ranges in Algorithm 2 could be unreliable for repositories with improper time configuration. We intent to mitigate this by using the build time of a `commit` from `TravisCI`-connected repositories [1].

To make the technique practical for developers and library maintainers, it is necessary to process ecosystems events such as changes made to a library and their dependencies in *real-time*. Building a real-time pipeline and adapting program analysis techniques to process on an event-basis calls for modifying current tools to work on an *incremental basis*.

5 RELATED WORK

In several studies [3–7], dependency networks have been used to study the dynamics of interconnected libraries in software ecosystems. However, there is a lack of research on *dependency management*, and yet, it is among some of the most common activities a developer needs to handle. To the best of our knowledge, there is one qualitative study by Bogart et al [2] that reasons about the cost of changes between library maintainers and their clients. However, none of these studies focus on techniques for a fine-grained and actionable dependency management for developers and library maintainers.

6 SUMMARY

In this paper, we present a technique to construct and analyze dependency relationships in a software ecosystem at the function-level granularity. The technique combines historical dependency data from version-controlled repositories with call graph construction to build a fine-grained representation of a dependency network. This representation can extend program analysis to diagnose problems at an ecosystem level, such as the spread of a security bug to affected clients or libraries by inspecting their interconnected function call relationship. We believe that our approach points towards actionable dependency management, where dependencies and their changes are evaluated at the source code level.

7 ACKNOWLEDGEMENT

The work is part of the Codefeedr project, which is financed by NWO with award number 628.008.001.

REFERENCES

[1] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 447–450.

[2] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 109–120.

[3] Eleni Constantinou and Tom Mens. 2017. An empirical comparison of developer retention in the RubyGems and npm software ecosystems. *Innovations in Systems and Software Engineering* 13, 2-3 (2017), 101–115.

[4] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2–12.

[5] Joseph Hejderup. 2015. *In dependencies we trust: How vulnerable are dependencies in software modules?* Master's thesis. Delft University of technology.

[6] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 102–112.

[7] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 351–361.

[5] https://github.com/Samsung/jalangi2

[6] https://nodesecurity.io/advisories/118, `isaacs/Minimatch` is the Github identifier of the `minimatch` package

[7] https://archive.org/details/MinimatchNode.csv

[8] https://github.com/airportyh/globwatch