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 countermeasure, they only provide indicative information. To remedy this situation, we propose a fine-grained dependency network that goes beyond the versions of 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. 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 led 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 than 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.
- 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-by-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
version-centric 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 be-
ond 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 net-
work 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 seman-
tic 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 depen-
dency 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 li-
braries 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 ana-
lyze 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 net-
works 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 ecosys-
tem is a directed and immutable graph G = (V, E) where:

(1) V is a set of versioned functions. Each v ∈ 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 (v1, v2) ∈ E
represents a function call from v1 to v2.

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
on dependency 2. Inside each dependency block, the nodes repre-
sent 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 de-
cide 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.
## Network Construction

<table>
<thead>
<tr>
<th>Algorithm 1: Network Construction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: git-based repository</td>
</tr>
<tr>
<td><strong>Output</strong>: processed dependency graph</td>
</tr>
<tr>
<td>1. ( G \leftarrow \emptyset )</td>
</tr>
<tr>
<td>2. ( \text{buildNetwork} \text{repo} )</td>
</tr>
<tr>
<td>3. ( C \leftarrow \text{filterDependencyCommits(repo)} )</td>
</tr>
<tr>
<td>4. if ( C \neq \emptyset ) then</td>
</tr>
<tr>
<td>5. foreach ( e \in C ) do</td>
</tr>
<tr>
<td>6. ( \text{rev} \leftarrow \text{checkout}(e) )</td>
</tr>
<tr>
<td>7. ( CG \leftarrow \text{constructCG}(	ext{rev}) )</td>
</tr>
<tr>
<td>8. ( CG_\text{ann} \leftarrow \text{annotateFunctions}(CG) )</td>
</tr>
<tr>
<td>9. if ( e ) specifies new library version then</td>
</tr>
<tr>
<td>10. ( G \leftarrow G \cup \text{getPrevCommitEdges}(e, CG_\text{ann}) )</td>
</tr>
<tr>
<td>11. end</td>
</tr>
<tr>
<td>12. if ( e ) specifies a dependency update then</td>
</tr>
<tr>
<td>13. ( G \leftarrow G \cup \text{resolveDependencies}(e, G, CG_\text{ann}) )</td>
</tr>
<tr>
<td>14. end</td>
</tr>
<tr>
<td>15. end</td>
</tr>
</tbody>
</table>

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 \( CG_\text{ann} \) 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 \( G_{\text{ver}} \) 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.

## 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 \texttt{git-diff} or \texttt{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 \( CG_{\text{reach}} \) contains a subset of the impacted (e.g reachable) function calls to one versioned function. The partial impact set \( CG_{\text{reach}} \) is added to the result in \( G \) (line 7). Finally, all versioned functions are processed, the impact set is returned (line 10).

## 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 \texttt{semver}

---

Algorithm 2: Resolving Dependencies

<table>
<thead>
<tr>
<th>Algorithm 2: Resolving Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: commit, ( CG_\text{ann} )</td>
</tr>
<tr>
<td><strong>Output</strong>: set of resolved dependency call edges</td>
</tr>
<tr>
<td>( \text{resolveDependencies} ) commit, to commit</td>
</tr>
<tr>
<td>1. ( E \leftarrow \emptyset )</td>
</tr>
<tr>
<td>2. ( \text{depfile} \leftarrow \text{getDependencyFile(commit)} )</td>
</tr>
<tr>
<td>3. ( D \leftarrow \text{parse(depfile)} )</td>
</tr>
<tr>
<td>4. if ( D \neq \emptyset ) then</td>
</tr>
<tr>
<td>5. foreach ( d \in D ) do</td>
</tr>
<tr>
<td>6. ( G_{\text{name}} \leftarrow \text{sliceByVersion}(G, d.name) )</td>
</tr>
<tr>
<td>7. ( G_{\text{ver}} \leftarrow \text{resolveVersion}(d, \text{constraint}, c) )</td>
</tr>
<tr>
<td>8. ( G_{\text{ver}} \leftarrow \text{sliceByVersion}(G_{\text{name}}, \text{ver}) )</td>
</tr>
<tr>
<td>9. ( CG_{\text{d}} \leftarrow \text{sliceByVersion}(G_{\text{ver}}, \text{commit}) )</td>
</tr>
<tr>
<td>10. ( E \leftarrow E \cup \text{getDependencyCalls}(CG_{\text{d}}, CG_{\text{ann}}) )</td>
</tr>
<tr>
<td>11. end</td>
</tr>
<tr>
<td>12. end</td>
</tr>
<tr>
<td>13. return ( E )</td>
</tr>
</tbody>
</table>

Algorithm 3: Impact Analysis

<table>
<thead>
<tr>
<th>Algorithm 3: Impact Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: A set of affected ( f ) in name &amp; version of library</td>
</tr>
<tr>
<td><strong>Output</strong>: Affected slice of the dependency network</td>
</tr>
<tr>
<td>( G \leftarrow \emptyset )</td>
</tr>
<tr>
<td>( \text{impact} ) (name, version, ( f ))</td>
</tr>
<tr>
<td>3. ( F \leftarrow \text{findVersionedFunctions}(\text{name}, \text{version}, f) )</td>
</tr>
<tr>
<td>4. foreach function ( f_i \in F ) do</td>
</tr>
<tr>
<td>5. ( G_{\text{reach}} \leftarrow \text{reachability}(f_i) )</td>
</tr>
<tr>
<td>6. if ( CG_{\text{reach}} \neq \emptyset ) then</td>
</tr>
<tr>
<td>7. ( G \leftarrow G \cap CG_{\text{reach}} )</td>
</tr>
<tr>
<td>8. end</td>
</tr>
<tr>
<td>9. end</td>
</tr>
<tr>
<td>10. return ( G )</td>
</tr>
</tbody>
</table>

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 \texttt{semver}

\footnote{https://github.com/npm/node-semver} library, and call graphs are extracted.
Figure 2: Impacted function calls in globwatch

from executing test cases of npm packages in Jalangi,\textsuperscript{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.\textsuperscript{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.\textsuperscript{7}

We selected an arbitrary package called globwatch (v0.0.1) from the results and checked-out the commit 82cef59 from airportyh/globwatch\textsuperscript{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.

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