

# Fine-Grained Network Analysis for Modern Software Ecosystems\*

PAOLO BOLDI, Dipartimento di Informatica, Università degli Studi di Milano, Italy

GEORGIOS GOUSIOS, Department of Software Technology, Delft University of Technology, The Netherlands

Modern software development is increasingly dependent on components, libraries and frameworks coming from third-party vendors or open-source suppliers and made available through a number of platforms (or *forges*). This way of writing software puts an emphasis on reuse and on composition, commoditizing the services which modern applications require. On the other hand, bugs and vulnerabilities in a single library living in one such ecosystem can affect, directly or by transitivity, a huge number of other libraries and applications. Currently, only product-level information on library dependencies is used to contain this kind of danger, but this knowledge often reveals itself too imprecise to lead to effective (and possibly automated) handling policies. We will discuss how fine-grained function-level dependencies can greatly improve reliability and reduce the impact of vulnerabilities on the whole software ecosystem.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories; Software development process management**.

Additional Key Words and Phrases: software reuse, security breaches, network analysis

## ACM Reference Format:

Paolo Boldi and Georgios Gousios. 2020. Fine-Grained Network Analysis for Modern Software Ecosystems. 1, 1 (December 2020), 15 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

## 1 INTRODUCTION

Software engineers reuse code to reduce development and maintenance costs. A popular form of software reuse is the employment of Open-Source Software (OSS) libraries, hosted on centralized code repositories, such as MAVEN<sup>1</sup> or NPM.<sup>2</sup> In such settings, developers specify *dependencies* to external library releases in a textual file, that is then committed to the repository of the *client program*. Automated programs, usually referred to as *package managers*, resolve the dependency descriptions and connect to the central repositories to download the specific library releases that are required to build the client program. Critically, dependencies can have dependencies of their own (*transitive dependencies*), and thus package managers need to resolve the transitive closure of the dependency graph in order to build client applications. Dependency versions are usually declared in a hierarchical versioning format, which is, in

\*This is an extended version of the paper *How Network Analysis Can Improve the Reliability of Modern Software Ecosystems* presented by the first author in 2019 at the IEEE International Conference on Cognitive Machine Intelligence (CogMI). Sections 3, 5, 6 are new and the remaining parts have been largely reworked.

<sup>1</sup><https://search.maven.org/>

<sup>2</sup><https://www.npmjs.com/>

Authors' addresses: Paolo Boldi, [paolo.boldi@unimi.it](mailto:paolo.boldi@unimi.it), Dipartimento di Informatica, Università degli Studi di Milano, Milan, Italy; Georgios Gousios, [g.gousios@tudelft.nl](mailto:g.gousios@tudelft.nl), Department of Software Technology, Delft University of Technology, Delft, The Netherlands.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2020 Association for Computing Machinery.

Manuscript submitted to ACM

Manuscript submitted to ACM

1

most, but not all cases, a derivative of *semantic versioning* [15]. The co-evolving network of dependencies and end-user applications is referred to as an *ecosystem*.

The convenience of declarative specification for package reuse has led to the massive adoption of package managers and package repositories. While the startup costs seem low for projects, reuse does not come for free. The software engineering literature has thoroughly documented the hidden maintenance costs around dependency reuse. From the perspective of a library user, it is hard to keep track of dependency updates especially for transitive dependencies [10], and assess their impact on the client code base [2]; the semantic versioning API evolution provisions are rarely respected in practice by library maintainers [16]; and entrusting precious data on code that the package manager automatically downloads is often not a wise choice [10]. On the other hand, from the perspective of the library maintainer, it is difficult to evolve APIs, for example by removing methods [18], without breaking clients [2], while incentives for professional maintenance of library code are lacking [13]. In addition, as libraries may also depend on other libraries, the library maintainers face the same issues as library users face. Finally, the very nature of dependency networks and the uncoordinated nature of their evolution burdens library users; recent studies have shown that the average Javascript program has an estimated mean of 80 transitive dependencies [20] (up from 54 in 2017 [8]), while 50% of the dependency sets change every 6 months in the Rust ecosystem [6].

As a result, in recent years, we have witnessed several spectacular failures of package management networks, with severe implications on client programs, end users and the further adoption of OSS:

- (1) A dispute over a library name in the *npm* ecosystem led to the removal of a library called `leftpad`, which consisted of just 11 lines of code. The package removal led to the collapse of thousands of libraries which directly or transitively depended on `leftpad`, and hence a major disruption for client programs.
- (2) 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 repository (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.

In this work, we argue that the main reason for such failures is that current tools work at the wrong level of abstraction: while developers release and keep track of *packages*, actual reuse happens at the *code* level. What we propose is to use *fine-grained network analysis*, as a more precise instrument for the analysis of package ecosystems. We can obtain precise, code-based reuse information by statically or dynamically analyzing the code at the function/method call level. Static analysis comprises a set of well-established techniques in the field of program analysis. While it can derive sound(y) [11] code reuse relationships, its precision is obstructed by the dynamic nature of modern programming languages. On the other hand, dynamic analysis can derive fully precise call relationships by instrumenting programs while they are running, but its soundness is limited by factors such as the availability of extensive test suites or representative workloads. Nonetheless, over-imposing call relationships on top of a package dependency network can lead to more precise analyses, effectively augmenting the soundness properties of the package dependency network with precision.

## 2 FORGES, PRODUCT AND DEPENDENCIES

At the dawn of the computer era, software development was mainly a solitary heroic activity of single men facing the complexity of problems with their bare hands in the darkness of their caves; but those days have gone: modern software development relies more and more on existing third-party libraries, giving programmers the freedom to focus on the core of what they have to do, delegating tedious or routine chores to reliable, specialized libraries they can download from the Internet.

In a way, this is just the industrial revolution arriving at the harbor of software production. In the words of Immanuel Kant: “All trades, arts, and handiworks have gained by division of labour, namely, when, instead of one man doing everything, each confines himself to a certain kind of work distinct from others in the treatment it requires, so as to be able to perform it with greater facility and in the greatest perfection. Where the different kinds of work are not distinguished and divided, where everyone is a jack-of-all-trades, there manufactures remain still in the greatest barbarism.” [7]

In free and open-source software, people share their work in the form of libraries, hosted on centralized code repositories, such as Maven<sup>3</sup>, *npm*, RubyGems, but also GitHub<sup>4</sup> or SourceForge<sup>5</sup>; some of these repositories are language-specific, whereas others are not. We will broadly refer to such repositories as *forges*. Forges use different approaches in organizing the libraries or projects they host, and in fact they refer to such libraries in a variety of ways: for example, *npm* uses a flat organization of “packages”, and the same does RubyGems with what they call “gems”, whereas Maven organizes what they call “artifacts” into groups. To avoid ambiguity, we shall prefer the abstract name *product* to refer to all of them: a product is a coherent piece of software that can be used alone or as a library to develop something else. We let  $\mathcal{P}$  be the set of all products<sup>6</sup>, and we shall typically refer to a product  $p \in \mathcal{P}$  with a name, such as `org.apache.maven.plugins` or `org.slf4j`.

Every product in every forge exists typically in a number of *revisions*: a new revision of a product is published to correct bugs or vulnerabilities found in previous releases, or to introduce new functionalities. Every revision is identified in some way (e.g., by a version number or by a hashcode); the granularity (hence, the frequency) of releases changes from one repository to another (for example, in GitHub releases may actually be identified with commits and are extremely frequent). We use the term *version* to refer to the identifier (whatever it is) that specifies what revision of a specific product we are talking about (e.g. `1.0.1-RC1`). We let  $\mathcal{R}$  be the set of all revisions, and we shall typically refer to a revision  $r \in \mathcal{R}$  with a name combining the product name and the version, such as `org.apache.maven.plugins-34` or `org.slf4j-2.0.0-alpha1`. We also let

$$\text{product} : \mathcal{R} \rightarrow \mathcal{P}$$

be the function that returns the product corresponding to a given revision. For instance,

$$\text{product}(\text{org.slf4j-2.0.0-alpha1}) = \text{org.slf4j}$$

Any given revision (i.e., version of a product on a forge) is available in the form of a number of *artifacts* (e.g., jar files, textual documents, zipped archives, etc.). Some of them contain the actual source code of the library, whereas others contain metadata of various kind (makefiles, installation instructions, documentation, etc.). For the purposes of the

<sup>3</sup><https://search.maven.org/>

<sup>4</sup><https://github.com/>

<sup>5</sup><https://sourceforge.net/>

<sup>6</sup>This set continuously evolves (typically, expands) in time; this fact will be ignored in this paper, as if we are taking a single snapshot of the state of things at a certain moment in time.

current paper, we only consider the source code and the so-called *dependency specification*. The latter metadata contains a description of the other products that are needed for this product to be compiled and/or executed: the syntax used in dependency specification depends on the forge. The dependency specification of revision  $r \in \mathcal{R}$  is denoted by  $\text{dep}(r)$ .

A number of tools called *package managers* are available that allow developers to specify which products their code depends on. For maximum flexibility<sup>7</sup>, the dependency specification is expressed by a CNF logical formula [12]; more precisely, the dependency specification is a set (interpreted as a logical conjunction) of *dependency clauses*, where each clause is itself a set (interpreted as a logical disjunction) of *dependencies*. A dependency is interpreted as a set of revisions of a product.

Here is an example of how a dependency is defined in Maven:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>34</version>
</plugin>
```

and here is another one for Ivy:

```
<dependency org="org.slf4j"
  name="slf4j-api" rev="[1.7,)" />
```

In these examples, you see that dependencies can point to a *specific* revision (in the first case, we are asking for version 34 of `org.apache.maven.plugins`) or to a set of revisions (in the second case, anything starting from version 1.7 of the `org.slf4j` library will fit).

### 3 PACKAGE MANAGERS AND DEPENDENCY RESOLUTION

Not only *package managers* allow to specify which products a piece of code depends on, but also they automatize the process of downloading recursively the dependencies of a given project and using them to build it. In order to describe how package managers work let us first provide some definitions.

The *global source dependency graph* is a directed graph whose node set is  $\mathcal{R}$  and with an arc from  $r$  to  $r'$  whenever  $r'$  satisfies at least one of the dependencies in one of the clauses of  $r$ . The out-neighbors of a revision  $r$  in the global source dependency graph are called the (direct) dependants of  $r$ .

The *source dependency graph* of revision  $r_0 \in \mathcal{R}$  is the smallest subgraph of the global source dependency graph that includes  $r_0$  and all the revisions that are reachable from  $r_0$  in the global source dependency graph. We show an example of source dependency graph in Figure 1.

A given set of revisions  $R \subseteq \mathcal{R}$  is said to satisfy a given dependency if it includes at least one element satisfying the dependency;  $R$  satisfies a given dependency clause if it satisfies at least one element of the clause;  $R$  satisfies a given dependency specification if it satisfies all of its dependency clauses.

We say that  $R$  is *dependency-closed* iff it satisfies  $\text{dep}(r)$  for all  $r \in R$ . It is easy to see that the set of nodes of the source dependency graph of  $r_0$  is dependency-closed, but not necessarily the smallest dependency-closed set of revisions including  $r_0$ . Moreover, it includes several revisions of the same product, which is in general undesirable.

A package manager, given a revision  $r_0 \in \mathcal{R}$ , finds a subset  $R \subseteq \mathcal{R}$  that satisfies the following properties:

<sup>7</sup>Not all package managers allow for this level of flexibility. In most cases, only one-element clauses are accepted.

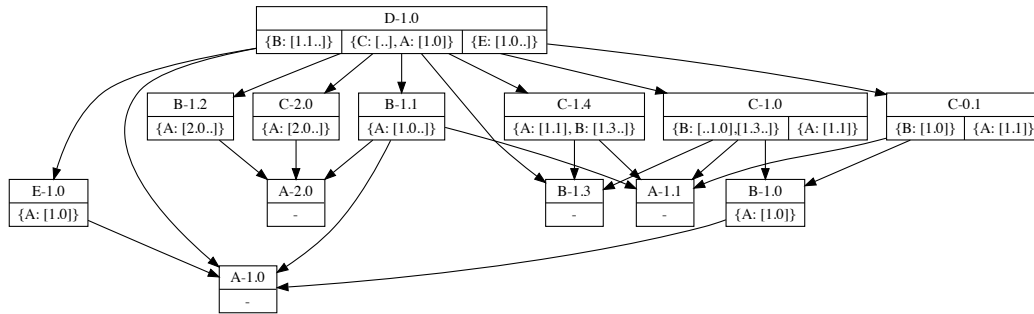


Fig. 1. An example: the source dependency graph of revision D-1.0. In this example, we have only 13 revisions of five products (A, B, C, D, E). Each node is a revision: in the upper part we write the name of the revision (product and version); in the lower part its dependency specification: each rectangle is a dependency clause, represented as a set of dependencies. For example, revision C-1.0 depends on revision 1.1 of A and on any revision of B with version  $\leq 1.0$  or  $\geq 1.3$ . As another example, C-1.4 requires either revision 1.1 of A or any revision of B with version  $\geq 1.3$ . Finally, D-1.0 depends on revision  $\geq 1.1$  of B, on any revision of E with version  $\geq 1.0$  and then either on any revision of C or revision 1.0 of A.

- $r_0 \in R$
- $R$  is dependency-closed
- it does not contain different revisions of the same product; that is, if  $r, r' \in R$  and  $\text{product}(r) = \text{product}(r')$  then  $r = r'$
- no proper subset of  $R$  satisfies the above three conditions.

In other words, the package manager should cherry-pick a subgraph of the source dependency graph of  $r_0$  so that no more than one revision of the same product is chosen but at the same time the resulting set is dependency-closed<sup>8</sup>. This subgraph is called *resolved dependency graph* of  $r_0$ . Figure 2 shows a possible resolved dependency graph for the example of Figure 1.

Resolution is a process that can produce different (incomparable) sets of revisions: for example, Figure 3 is an alternative resolved dependency graph for the same revision. Not only, different package managers use different resolution strategies and may thus end up with different resolved dependency graphs, but even the same package manager may produce different resolutions depending on some contextual information (e.g., a timestamp).

In the following, we assume that we have a single package manager and let  $\text{resolved}(r_0, c) \subseteq \mathcal{R}$  be the (node set of the) resolved dependency graph of  $r_0$  obtained by the package manager in the context  $c$ .

#### 4 THE PRICE OF REUSE

Public software forges, package managers and the resulting ecosystems made the dream of code reuse a reality, but this reality comes at a price. These ecosystems are extremely fragile: according to [8], in 2017 JavaScript products used to have an average of 54.6 products they depended upon (directly or transitively), with a steady growth of more than 60% every year; there are products in RubyGems that are in the transitive dependency of more than 400 000 other products (meaning that if you remove one them, about 40% of all the products in the ecosystem will cease to work).

<sup>8</sup>It should be noted that real-world package managers are more complex than this; we refer the interested reader to [1] for details. Nonetheless, we think that the definition we are using contains the core of what a package manager is supposed to achieve.

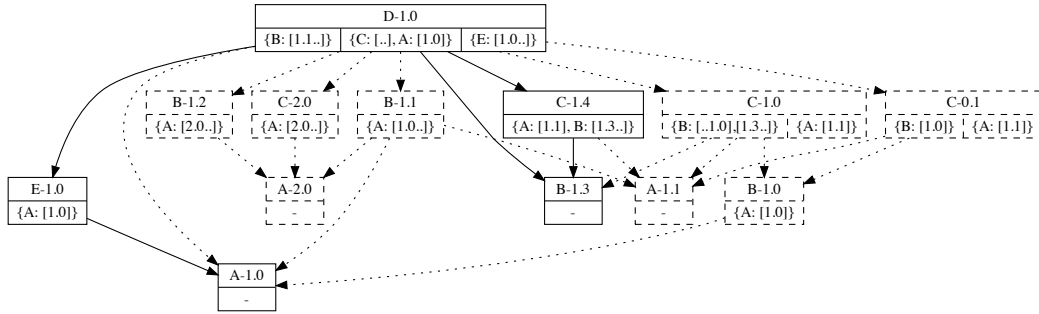


Fig. 2. A resolved dependency graph of  $D-1.0$  obtained from the source dependency graph of Figure 1. The dotted revisions are not included in the resolved dependency graph. As one can easily see, all dependencies are satisfied: for instance,  $D-1.0$  requires a version of  $B$  that is 1.1 or newer (and  $B-1.3$  is such), a version of  $E$  that is 1.0 or newer ( $E-1.0$  is such) and then either a revision of  $C$  or version 1.0 of  $A$  (and the former is provided, because the graph includes  $C-1.4$ ).

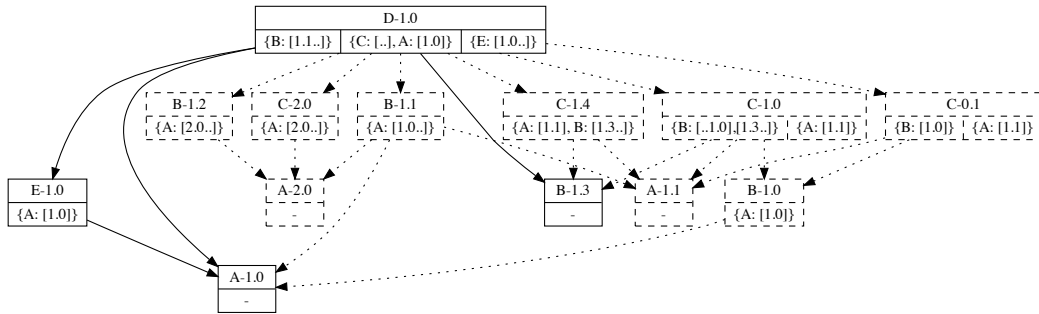


Fig. 3. An alternative resolved dependency graph of  $D-1.0$  obtained from the source dependency graph of Figure 1. Once more, all dependencies are satisfied: for instance  $D-1.0$  requires a version of  $B$  that is 1.1 or newer (and  $B-1.3$  is such) and then either a revision of  $C$  or version 1.0 of  $A$  (and the latter is provided because the graph includes  $A-1.0$ ).

Package *users* gain great value from reusing code, but they need to invest significant resources into shielding themselves from software security, legal compliance and source code incompatibility issues.

According to Snyk's annual 2019 report on the state of open-source security<sup>9</sup>, the number of security issues found in software almost doubles every two years (+44% every year), and about 78% of them are found in indirect dependencies. This observation hints at how complicated software maintenance actually is: when a new vulnerability alert is found, for example, it is essentially impossible to know whether the issue impacts on a specific product. Of course, the dependency graph can be used to know if the impact is possible, but it is not enough to know if the specific piece of code that was broken is ever actually invoked (directly or indirectly) in the product we are looking at, and in the positive case what are the functions that are put at risk and how the problem can be circumvented.

<sup>9</sup><https://bit.ly/SoOSS2019>

Even worse: 69% of the developers are totally unaware of vulnerabilities existing in the products they depend upon, and 81.5% of the systems simply don't update their dependencies [10]. This is probably because on one hand few tools are available to warn those developers in an automated way; it is true, for example, that GitHub has recently launched an automated service notifying repository owners that they depend on packages affected by known security vulnerabilities, but even so, it is like crying wolf: in most cases, vulnerabilities found in dependencies would not affect their software anyway.

As a concrete example, consider in Figure 4 the resolved dependency graph (the same as in Figure 2 but where we dropped the products that were not used in the resolution). This is in fact a fragment of the source dependency graph of Figure 1.

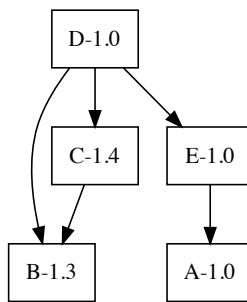


Fig. 4. A fragment of the global source dependency graph, corresponding to the resolution of Figure 2.

Suppose that a security alert is issued about revision B-1.3; then by transitivity two other products involved are potentially infected (as shown in Figure 5). But is this really the case?

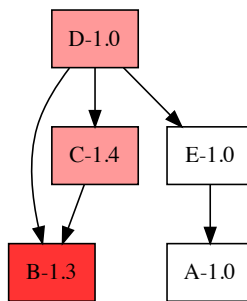


Fig. 5. If a vulnerability is found in B-1.3 (red in the picture), some other revisions in this picture may be at risk (light red in the picture).

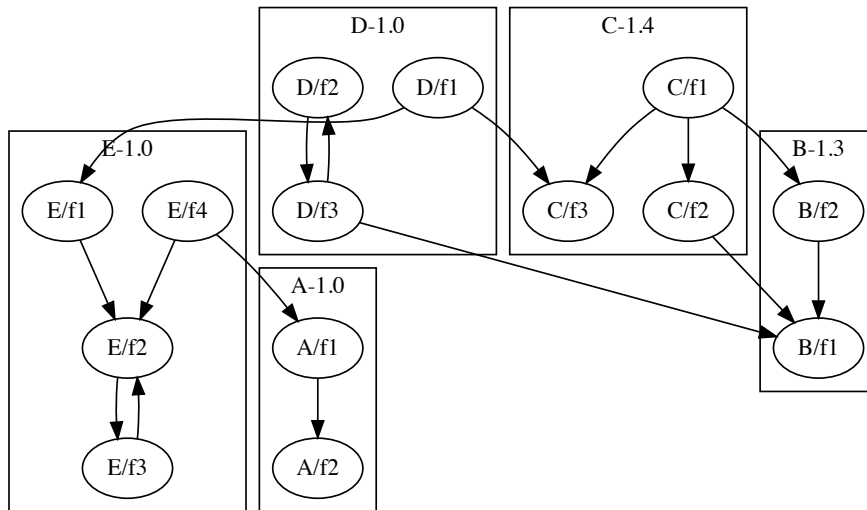


Fig. 6. A function-level view of Figure 4.

From a different viewpoint, also legal and licensing issues are made quite complex by dependencies. The complexity of licensing contracts and their effects on dependencies is often underestimated by software developers. While several companies offer license compliance checking services (e.g., BlackDuck software, WhiteSource, FOSSA), a project’s source code cannot be checked in isolation from its dependency graph, and a project’s dependency graph can extend to more components than what specified in the package manager (e.g., implicit dependencies on system libraries).

On the other hand, package *providers* have no reasonable means of evolving what they offer in a systematic way, because they are not sure of the impact a change in their products, or in their licensing, can have on their clients.

The issues we just highlighted are related to the relatively naive design of package managers: they only resolve dependencies based on package versions. As such, they cannot assess the risk of using dependencies, they cannot notify developers of critical (e.g., security) updates, nor can they assist them to evaluate the ecosystem impact of API evolution tasks (e.g., removing a deprecated method). Even if they were able to implement such functionalities, they could do so only at the bulk level of products, but not at the level of function/method.

## 5 NETWORK ANALYSIS AS A SOFTWARE ENGINEERING TOOL

While package-based dependency networks capture dependencies at the product level, actual software reuse happens at the code level. For example, Figure 6 shows the same scenario depicted in Figure 4, but this time we can see the functions within each revision, along with their calls. What we can observe is that a call from E/f4 to A/f1 does not reach A/f2. However, A/f2 is reachable from C/f3. In this setting, a potential vulnerability in A/f2 would render C potentially



vulnerable as well, whereas it would not affect E. If we were able to identify exact function calling relationships between packages, our whole analysis would become more precise.

Fortunately, the field of program analysis has been working for long to automatically extract calling relationships in source code, in the form of call graphs. Call graphs can be generated either *statically*, by analyzing the source code, or *dynamically*, by instrumenting the code and tracing program executions. Static call graphs are neither complete (e.g., because they miss calls by reflection or by dynamic dispatch), nor sound (some identified execution paths may never materialize in real executions). Dynamic call graphs, on the other hand, are strongly dependent on the test cases used to generate the traces. Despite those shortcomings, call graphs are used in a variety of use cases, notably, change impact analysis [17] and dead code elimination [9].

What we propose is to represent dependency relationships with call graphs. Concretely, every (revision of a) product can be seen as a set of functions, each calling other functions either belonging to the same product or to some of its dependants: it can be abstractly represented as a directed graph with two types of nodes, internal and external. While internal nodes represent an actual function within the same revision, external nodes are somehow less precisely identified — they represent a function in some revision of some other product, but which revision is not known, because it depends on the dependency-resolution process.

The whole dependency-resolution process depends on the choice of a resolution strategy adopted by the package manager, whose behavior is determined on the specific revision of a specific product, we aim at using as a starting point. At the end of this process, we can actually identify external nodes of each single revision involved with internal nodes of other revisions, obtaining the actual global call graph.

## 6 (STITCHED) CALL GRAPHS

Let us formalize the notion of call graph we have outlined. Every revision  $r \in \mathcal{R}$  is associated with a directed graph  $G_r = (V_r, E_r)$ , the *call graph of  $r$* , whose node set  $V_r$  is bipartite  $V_r = V_r^{\text{int}} \cup V_r^{\text{ext}}$  into a set of internal nodes ( $V_r^{\text{int}}$ ) and a set of external nodes ( $V_r^{\text{ext}}$ ); the latter nodes have no outgoing arcs. Hence  $E_r \subseteq V_r^{\text{int}} \times V_r$  (i.e., all arcs start from an internal node); the arcs themselves can be classified as being internal (if they end up into an internal node) or external (if they end up into an external node).

External nodes (i.e., functions that exist in some dependant) carry sufficient metadata to allow one to identify them with some internal node of a dependant regardless of how the resolution process was performed. Formally, we can say that there is a function

$$\sigma_r : V_r^{\text{ext}} \rightarrow 2^{\bigcup V_{r'}^{\text{int}}}$$

where  $2^X$  denotes the set of subsets of  $X$ , and the union ranges over all the out-neighbors  $r'$  of  $r$  in the global source dependency graph. This function serves the purpose of mapping external nodes (i.e., calls to some library function) to internal nodes of dependants.

As an example, Figure 7 shows the call graphs of C-1.0 and its dependants (from Figure 1); external nodes are diamond-shaped. The dotted arrows represent  $\sigma$ . We can see that C-1.0 contains two external calls: C/f1 and C/f2 both call some external function. Remember that, according to Figure 1, C-1.0 depends on A-1.1 and on any version of B excluding those larger than 1.0 and smaller than 1.3: we have only two revisions of B satisfying the constraint. The yellow external nodes is mapped to the two internal nodes B/f1 (of revision B-1.3) and B/f3 (of revision B-1.0); which one will be used depends on whether the resolution process chooses B-1.3 or B-1.0. The blue external node

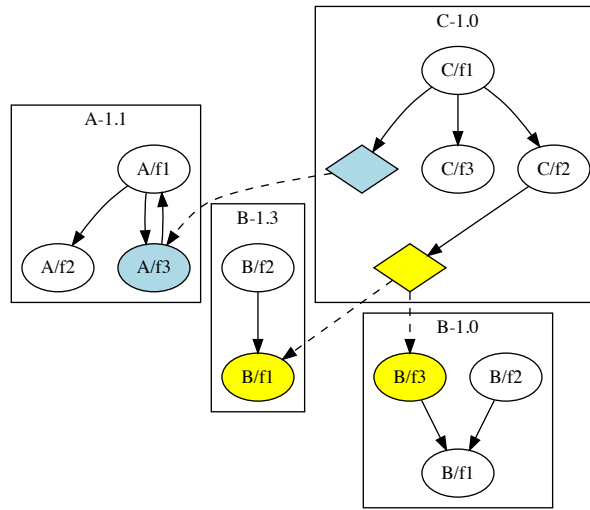


Fig. 7. This picture shows the call graphs of  $C-1.0$  and its dependants (from Figure 1). External nodes are diamond-shaped, and the dotted arrows represent  $\sigma$ .

corresponds to a call to a function of product A, and here only A-1.1 matches, with A/f3 corresponding to the external call.

Using this function, we can define the notion of *stitched call graph* of revision  $r_0$  and a context  $c$ , written  $G(r_0, c)$ : consider the graph obtained as a union of all  $G_r$  with  $r \in \text{resolved}(r_0, c)$ , and quotient its node set with respect to the smallest equivalence relation<sup>10</sup>  $x \sim y$  such that  $x \sim y$  whenever  $y \in \sigma_r(x)$  for some  $r \in \text{resolved}(r_0, c)$ . This graph contains only internal nodes (because each external node is mapped to one or more internal nodes by  $\sigma_r$ ), and it is the function-level equivalent of  $\text{resolved}(r_0, c)$ .

We call this graph “stitched” because it is obtained by stitching together the call graphs of a resolved dependency graph identifying every external node with a set of internal nodes of dependants. Figure 8 shows how the stitched call graph construction works.

## 7 OPPORTUNITIES & CHALLENGES

We present examples on how the availability of stitched call graphs can improve the reliability of software development practices, and improve the robustness of the whole ecosystem:

- Every time a bug or a security breach is found in a library, developers will be able to precisely analyze whether their applications are calling into vulnerable code and decide whether dependency updates are necessary; the ecosystem itself will be able to notify the developers of vulnerable applications in real-time, after a security issue has been disclosed. This type of functionality would have been beneficial in preventing the Equifax breach. By

<sup>10</sup>Given a graph  $G = (V, E)$  and an equivalence relation  $\approx$  between its nodes, the quotient graph  $G/\approx$  has the set of equivalence classes  $V/\approx$  as node set and an arc from  $[x]$  to  $[y]$  if and only if there is some  $x' \approx x$  and some  $y' \approx y$  such that  $(x', y') \in E$ .

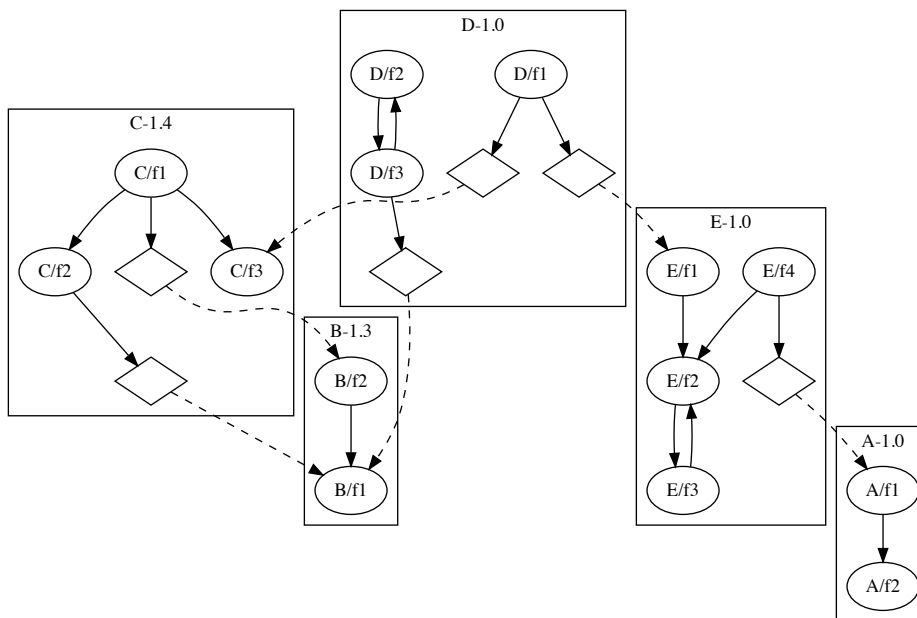


Fig. 8. This picture shows the call graphs of the revisions involved in Figure 2: within each graph, external nodes are diamond-shaped. The dotted arcs correspond to the map  $\sigma$  (the function will map each external nodes to possibly many internal nodes in many dependants, but here we are looking at a resolved graph). The graph obtained after stitching is that of Figure 6.

analyzing security alerts at a function level, you can get much more precise information and know exactly which parts of your code need to be fixed: Figure 9 shows that in our example revision D-1.0 is not at all impacted by a bug found in function B/f2. Similarly, only a portion of the functions of C-1.4 are impacted.

- Using the call graph, one can precisely identify the ecosystem-wide impact (direct and transitive) that any API change can have. This kind of change impact analysis can thus become a first-class tool for software developers (much like a debugger or a profiler is). Developers will be able to get quantitative answers to questions such as “How many packages are affected if I remove a certain method/interface?” and will be able to make decisions and proactively notify downstream packages for breaking changes when an upstream API has changed. The availability of this functionality would have prevented the left-pad incident, for example.
- The fact that licensing is usually only evaluated at the library level and not at the function level introduces (at least conceptually) a rigidity that is not desirable for organizations releasing open source software components. For example, a library may include subroutines with different licensing contracts. Using the call graph we can check that function-level licenses of our own software are consistent with one another, and that they are consistent with the licenses attached to the libraries our software depends on.

More generally, the call graph contains a big deal of information about software that could not be obtained otherwise. In particular, the notion of centrality [4] applied to the call graph can determine which parts of the software ecosystem

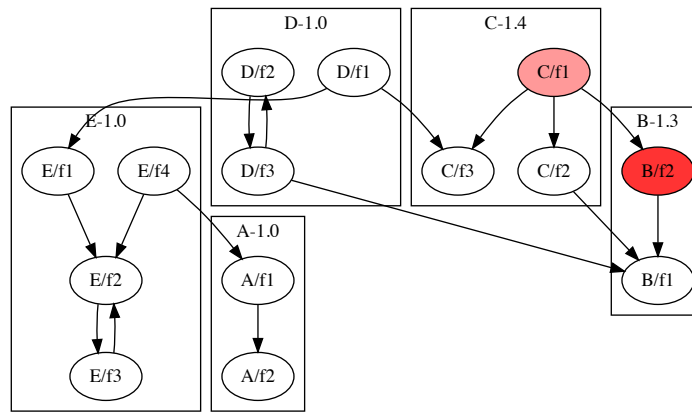


Fig. 9. The function-level analysis shows that if the vulnerability in B-1.3 is because of function B/f2, then only C/f1 is at risk. In particular, none of the functions within library D-1.0 is involved.

are more relevant; this information can be used to target critical or risky functions, or to better concentrate maintenance efforts of large software repositories. This path can be thought of as moving one step beyond the traditional approach to profiling, using network analysis methods as the weapon of choice.

Despite the obvious advantages, analyzing whole ecosystems at the function level is not trivial. We foresee the following challenges:

*Scale.* The unified call graph Hejderup et al. [6] built for Rust contained 6 million nodes and 16 million edges. However, Rust as an ecosystem featuring 250k revisions is *an order of magnitude* smaller than that of Java (2.5M) or Python (1.2M). Even assuming a prudent estimate of about 100 functions per revision, we are talking of graphs with about  $10^{11}$  nodes, and thousands of billions of arcs!

These graphs are bound to challenge the current state-of-art in graph processing systems, especially considering that those graphs change frequently and that they materialize dynamically based on the dependency-resolution strategy. In addition, such graphs will need to be queried (e.g., traversed and sliced) in real time by clients.

This challenge calls for new, aggressive, dynamic compression techniques, specially tailored around the structure and topology of call graphs, that can go beyond the state-of-art in graph compression [5]. For deeper analysis and ranking it might be necessary to store in compact form some metadata about the actual calls. For example, the users might contribute profiling data making it possible to decorate arcs of the graph with the estimated number of times a particular function is called at a specific location in the code: such information would be invaluable in the ranking process, but it would require further storage and new as-yet unknown compression techniques.

*Call graph soundness.* Soundness is an important property of static analyses. Unsound analyses lead to reporting false positives to developers (e.g., a piece of code is labeled as having a bug, but in reality, it does not), which in turn makes developers not trust the analyses. Unfortunately, creating sound static analyses is known to be an undecidable problem

(Rice’s theorem). For the proposed system to be successful, the right balance between soundness and usefulness should be reached.

The problem of unsoundness translates to either under- or over-approximation of a program’s behavior. In practical terms, and since nodes (functions) can always be fully recovered, the ecosystem call graph may contain more or less edges than what we could obtain from a program execution trace. To mitigate this problem, we can employ two strategies: crowdsourcing the call graph augmentation and machine learning on graphs. In the first case, users of the system can be asked to instrument their test runs or actual deployments with dynamic call graph extraction tools. A centralized location keeping track of the call graphs can compare the uploaded traces and create edges in case they are missing; full trust can be given to such traces, as dynamic call graphs are *de facto* precise. Finally, network analysis can play an important role here. Specifically, one could envisage methods that exploit the structural and evolutionary properties of the graph to employ algorithms, such as friend recommendation, in order to augment or prune the ecosystem call graph.

*Bringing value to developers.* What we have just described is only the backbone behind a set of tools and services that should integrate with the final developer’s programming environment (e.g., in the form of plugins for the programmers’ favorite IDE) as well as with continuous integration tools. To make the call graphs available and to enable additional analyses (e.g., change impact analysis, security/bug propagation etc), the FASTEN project <sup>11</sup> develops a continuously updated service that will act as a hub for ecosystem-scale analysis for the Java, Python, C and Rust programming communities.

The ongoing FASTEN EU Project proposes an innovative approach to solve the above challenges; FASTEN design is based on the streaming toolchain sketched in Figure 10: data coming from different sources are streamed to a centralized server that extracts call graphs, stores them in compressed form and prepares efficient data structures (indices) to query them efficiently. Front-end tools are provided for developers to interrogate the call-graph database.

The preliminary results, especially for what concerns call-graph compression and analysis, are very promising. The first nontrivial problem related to call graphs is about their compressibility: call graphs are a relatively new object, and it is unclear whether standard compression techniques [5] can be fruitfully applied. It turns out that call graphs can be indeed compressed very well, for instance using variants of LLP [3], a technique initially developed to extend webgraph compression techniques to more general types of graphs (e.g., social networks).

A second important preliminary observation is that, despite their size, call graphs exhibit relatively short call chains, allowing for efficient resolution of reachability and co-reachability queries [19], which are the fundamental ingredient of most change-impact studies.

These early findings suggest that the challenges we envision can be won, leading to a new, efficient, safer and more productive software-writing environment.

## 8 CONCLUSIONS

Since the introduction of software modularization [14], software engineers have long been trying to make software reusable. Technologies such as object-oriented programming, components, commercial off-the-shelf libraries and aspects have all touted the reuse horn, to various degrees of success. During the last 15 years, package managers and the open-source software movement have succeeded in making the dream of software reuse a tangible reality. However, this

<sup>11</sup><https://www.fasten-project.eu>

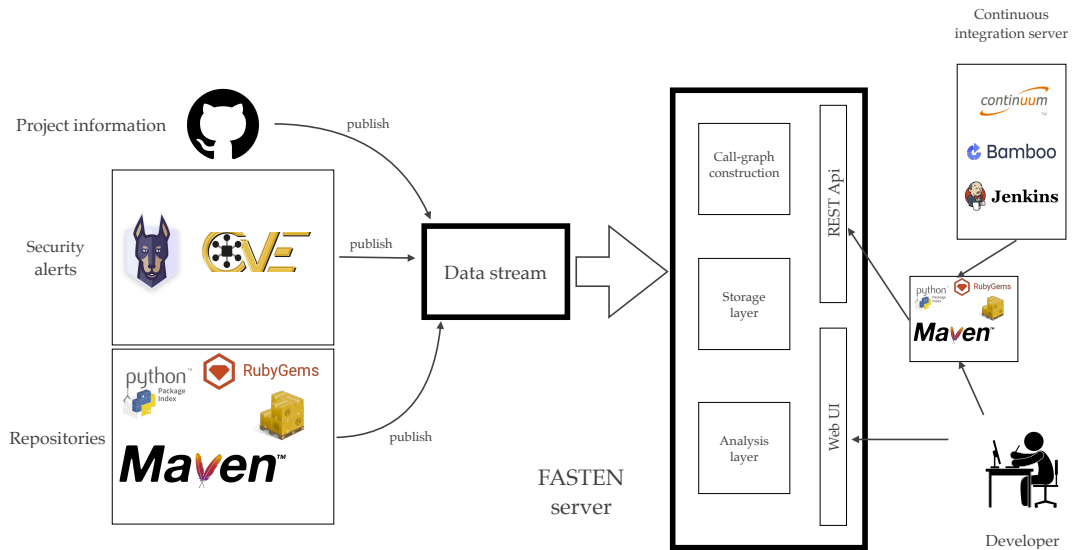


Fig. 10. The FASTEN EU Project toolchain.

reality is not without problems. The current state of practice and the frequent, spectacular failures of modern software ecosystems point towards the limits of what comprises the first generation of package management technology.

In this vision paper, we presented a new design for package management systems that can, in a large degree, overcome the fallacies of current ones and pave the way to new, exciting reuse possibilities. What we need to do is change our unit of reuse from the package to the function/method and embrace network analysis as first-class citizen in future software engineering tools.

#### ACKNOWLEDGMENT

We would like to express our gratitude to Sebastiano Vigna and Stefano Zacchiroli for their precious help. This work has been partially funded by the FASTEN EU Project, H2020-ICT-2018-2020 (Information and Communication Technologies).

#### REFERENCES

- [1] P. Abate, R. Di Cosmo, G. Gousios, and S. Zacchiroli. 2020. Dependency Solving Is Still Hard, but We Are Getting Better at It. In *The 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (London, Ontario, Canada). 547–551. <https://doi.org/10.1109/SANER48275.2020.9054837>

- [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] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM, 587–596. <https://doi.org/10.1145/1963405.1963488>
- [4] Paolo Boldi and Sebastiano Vigna. 2014. Axioms for Centrality. *Internet Math.* 10, 3-4 (2014), 222–262.
- [5] Paolo Boldi and Sebastiano Vigna. 2019. (Web/Social) Graph Compression. In *Encyclopedia of Big Data Technologies.*, Sherif Sakr and Albert Y. Zomaya (Eds.). Springer. [https://doi.org/10.1007/978-3-319-63962-8\\_54-1](https://doi.org/10.1007/978-3-319-63962-8_54-1)
- [6] Joseph Hejderup, Moritz Beller, and Georgios Gousios. 2018. *Building a Unified Call Graph at Ecosystem Level*. Technical Report TUD-SERG-2018-002. Delft University of Technology. 20 pages. <http://gousios.org/pubs/ucg.pdf> Online: <http://gousios.org/pub/ucg.pdf>.
- [7] Immanuel Kant. 2002 [1785]. *Groundwork for the Metaphysics of Morals*. Oxford University Press.
- [8] 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, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. 102–112. <https://doi.org/10.1109/MSR.2017.55>
- [9] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial dead code elimination. *ACM SIGPLAN Notices* 29, 6 (1994), 147–158.
- [10] Raula Gaikovina Kula, Daniel M. Germán, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. Do Developers Update Their Library Dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration. *CoRR* abs/1709.04621 (2017). arXiv:1709.04621 <http://arxiv.org/abs/1709.04621>
- [11] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46.
- [12] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. 2006. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 199–208.
- [13] Cassandra Overney, Jens Meinicke, Christian Kästner, and Bogdan Vasilescu. 2020. How to Not Get Rich: An Empirical Study of Donations in Open Source. In *Proceedings of the 2020 42th International Conference on Software Engineering*.
- [14] David L Parnas. 1972. On the criteria to be used in decomposing systems into modules. In *Pioneers and Their Contributions to Software Engineering*. Springer, 479–498.
- [15] Tom Preston-Werner. [n.d.]. Semantic Versioning 2.0.0. <https://semver.org>
- [16] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158.
- [17] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. 2004. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 432–448.
- [18] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2018. On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK. *Empirical Software Engineering* 23, 4 (2018), 2158–2197.
- [19] Jeffrey Xu Yu and Jiefeng Cheng. 2010. *Graph Reachability Queries: A Survey*. Springer US, Boston, MA, 181–215.
- [20] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 995–1010.