How Network Analysis Can Improve the Reliability of Modern Software Ecosystems

Paolo Boldi
Dipartimento di Informatica
Università degli Studi di Milano
Milan, Italy
paolo.boldi@unimi.it

***Abstract*—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.**

***Keywords*-Software reuse, security breaches, network analysis.**

I. Two stories

A. The programmer who almost broke the Internet

In 2016, Azer Koçulu was a 28-year-old software developer living in Oakland, California. He had been publishing open-source software for years and uploaded most of his work to *npm*, a popular package manager used by many JavaScript projects to install their dependencies. Like many of us who write code that anyone can use, Koçulu was adhering to the ethics of early programmers at MIT and later distilled as a set of more concrete values put forth by Richard Stallman, the most famous activist of the free software movement: as Stallman wrote in his 1985 manifesto [1], "the fundamental act of friendship among programmers is the sharing of programs".

One of the many open-source JavaScript libraries that Koçulu wrote was called `kik` (it was a library to help programmers set up their project templates). On March 11, Koçulu received an email from Bob Stratton, a patent and trademark agent who was working for kik.com[1], the Ontario-based world-popular messaging app: Stratton was asking Koçulu to rename his software library to avoid accusations of trademark infringement.

Koçulu refused to rename his project, even after being offered 30 000 USD by Mr. Stratton: his principles were more important than money, and he asked Stratton to refrain from contacting him again. Stratton brought his request up to *npm*'s chief executive, Isaac Schuleter, who answered (on March 18): "In this case, we believe that most users who would come across [Koçolu's] `kik` package, would reasonably expect it to be related to kik.com. In this context, transferring ownership of these two package names achieves that goal."

npm siding with Kik was totally unexpected by Koçulu: "I know you for years," he commented, "and would never imagine you siding with corporate patent lawyers threatening open source contributors." In an act of rage, and in the name of his ideals, on March 20 Koçulu decided to remove from *npm all* of his packages. "I dont wanna be a part of *npm* anymore!", he said.

Two days later, on March 22, JavaScript developers all around the world started to receive a strange error message when they tried to run their code:

```
npm ERR! 404 'left-pad' is not in the npm registry
```

Most programmers did not even know what `left-pad` was, but somehow their software couldn't be run without it. In a matter of hours, many of them found the GitHub repository[2] were `left-pad` was maintained. It turned out to be a mere 11-line-function that adjusted a given string to a desired length by padding it with blanks on the left.

```
module.exports = leftpad;
function leftpad (str, len, ch) {
  str = String(str);
  var i = -1;
  if (!ch && ch !== 0) ch = ' ';
  len = len - str.length;
  while (++i < len) {
    str = ch + str;
  }
  return str;
}
```

Some of the most widely used *npm* packages were suddenly broken. Among them, React[3], created and used by Facebook; about one more million of websites were affected. Regard-

Partially funded by the FASTEN EU Project, H2020-ICT-2018-2020 (Information and Communication Technologies)

[1]http://kik.com/

[2]https://github.com/azer/left-pad/issues/4#issue-142787766

[3]https://facebook.github.io/react/

less of how trivial or humble `left-pad` might seem, its disappearance was felt globally, because countless packages and websites depended directly or indirectly on it. Ironically enough, even kik.com ran into the `left-pad` problem—Mike Roberts, who was managing the company's messaging app, said in an interview that the error prevented his colleagues from running software they had been working on. "What the heck" Roberts recalled thinking, "one of our packages is missing?".

npm finally decided to restore the 11 lines of code. According to Laurie Voss (the chief technology officer of *npm*) "Un-un-publishing is an unprecedented action that we're taking given the severity and widespread nature of breakage, and isn't done lightly. This action puts the wider interests of the community of *npm* users at odds with the wishes of one author; we picked the needs of the many."

B. The missing patch that costed 4 billion dollars

Equifax is a consumer credit reporting agency founded in 1899, one of the three largest agencies of this kind (along with Experian and TransUnion); they collect information on over 800 million consumers and almost 100 million businesses of all countries. In September 2017, Bloomberg News reported that Equifax had been the victim of a *major breach of its computer systems* in March 2017. Equifax announced that the event potentially impacted approximately 145.5 million US consumers, and up to 44 million British residents as well as tens of thousand of people in the rest of the world.

Information accessed by the hacker (or hackers) in the breach included first names, family names, SSN's, birth dates, addresses and, in some cases, driver's license numbers. Credit card numbers for approximately 209,000 US consumers were also accessed.

It is estimated that this massive data breach costed Equifax an unprecedented amount of more than 4 billion US dollars. Equifax disclosed in 2017 that the reason behind the attack was a failure to patch one of its Internet servers against a pervasive software flaw: the attackers entered the Equifax system in mid-May through a web-application vulnerability that had a patch available since March.

The vulnerabilty exploited by the attackers was in the Apache Struts web-application software. The Apache Software Foundation released a statement saying that it was sorry that attackers exploited a bug in its software to breach Equifax; nonetheless, according to Ren Gielen (vice-president of Apache Struts) "Most breaches we become aware of are caused by failure to update software components that are known to be vulnerable for months or even years".

In fact, a study on software security [2] revealed that 80% of the code in todays' applications comes from libraries and about one fourth of such libraries have known vulnerabilities.

II. Repositories, libraries, dependencies

Back in the old days, software development was 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.

This is just the industrial revolution arriving at the harbour 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." [3]

In free and open-source software, people share their work in the form of libraries, hosted on centralized code repositories, such as Maven[4], *npm* or RubyGems, or so-called forges, such as GitHub[5] or SourceForge[6]; some of these repositories are language-specific, whereas others are not. Libraries are distributed on these repositories in the form of suitable artifacts (e.g., jar files in the case of typical Java libraries), and usually come in a number of releases: a new release of a library is published to correct bugs or vulnerabilities found in previous releases, or to introduce new functionalities. Every release is identified by some identifier (e.g., a version number). The granularity (hence, the frequency) of releases changes from one repository to another (for example, in GitHub releases actually coincide with commits and are extremely frequent).

From the viewpoint of developers, a number of tools called *package managers* are available that allow them to specify which libraries their code depends from, and that can automatize the process of downloading recursively the dependencies of a given project and using them to build it. Here is an example of dependency specification in Maven:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.0</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 be specified with respect to a *specific* release (in the first case, we are asking for version `3.8.0` of the `org.apache.maven.plugins` library) or to a set of releases (in the second case, anything starting from version `1.7` of the `org.slf4j` library will fit). In the second case,

[4]https://search.maven.org/
[5]https://github.com/
[6]https://sourceforge.net/

the decision of which release(s) to use is left to the package manager, and different package managers may adopt different strategies.

In Figure 1 you can see a bunch of (releases of) libraries with their dependencies. Here both library A and library B depend directly on library C, and transitively on library D.

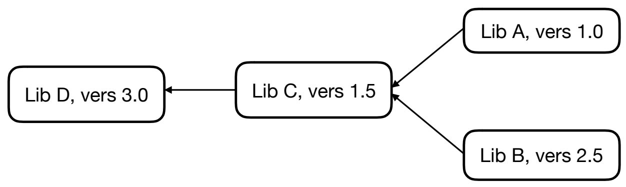

Fig. 1. Various releases of libraries, with their dependencies.

III. 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 [4], in 2017 JavaScript libraries used to have an average of 54.6 libraries they depended upon (directly or transitively), with a steady growth of more than 60% every year; there are libraries in RubyGems that are in the transitive dependency of more than 400 000 other libraries (meaning that if you remove one them, about 40% of all the libraries in the ecosystem will cease to work).

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[7], 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 sofware 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 library. 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 library 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.

Even worse: 69% of the developers are totally unaware of vulnerabilities existing in the libraries they depend upon, and 81.5% of the systems simply don't update their dependencies [5]. 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, suppose that a security alert is issued about Library D (version 3.0) of Figure 1; then by transitivity all libraries in Figure 2 are potentially infected. But is this really the case?

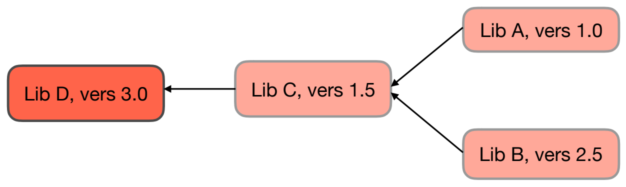

Fig. 2. If a vulnerability is found in Library D, all other libraries in this picture may be at risk.

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., Black-Duck 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 an systematic way, because they are not sure of the impact a change in their libraries, 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 *libraries*, but not at the level of function/method.

The idea of moving the analysis of software ecosystems to the granularity level of functions[8] is at the core of the recent EU Project FASTEN [6]. Specifically, FASTEN aims at building a fine-grained call graph: ideally, for each library release in an ecosystem, FASTEN will build a call graph.[9] Links to developer-specified dependencies will then be resolved at the function-call level. The result is a versioned, ecosystem-level

[7]https://bit.ly/SoOSS2019

[8]Here, by function we abstractly refer to an atomic unit of code, or subroutine, that performs a specific task. Depending on the programming language, it can be called a function, or a method, or routine, etc.

[9]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 simpler to build but they are neither complete (e.g., because they miss calls by reflection or by dynamic dispatch) nor sound (some 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.

call graph, that not only solves the issues identified above, but also both opens the door to advanced applications and challenges the current state of the art in graph storage and processing.

Figure 3 shows the same scenario depicted in Figure 1, but this time we can see the functions within each library, and their calls (represented as dotted arrows).

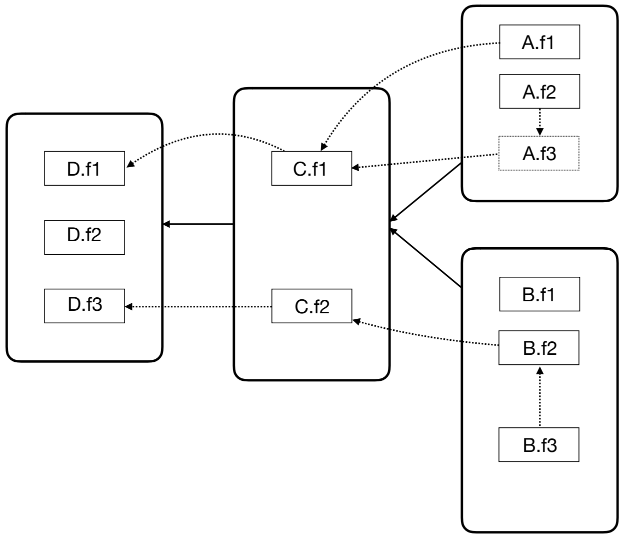

Fig. 3. A function-level view of Figure 1.

IV. Network analysis as a software-development tool

Concretely, every (release of a) library can be seen as a set of functions, each calling other functions either belonging to the same library or to some other library (specified as a dependency): 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 library, external nodes are somehow less precisely identified — they represent a function in some release of some other library, but which release 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 behaviour is determined on the specific release of a specific library we aim at using as a starting point. At the end of this process, we can actually identify external nodes of each single library release involved with internal nodes of other libraries releases, obtaining the actual global call graph.

Let me offer some examples on how the availability of such a graph 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, all 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 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 4 shows that in our example Library A is not at all impacted by a bug found in function D.f3, and only some of the functions of Library B are impacted, but not all of them.
- Using the call graph, one can precisely identify the ecosystem-wide impact (direct and transitive) that any API change can have. The change-impact analysis can 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 take decisions and proactively notify downstream packages for breaking changes when an upstream API has changed. The availability of this kind of tool 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 we might want to remove. Think of a library where different subroutines may have 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 from.

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 [7] applied to the call graph can determine which parts of the software ecosystem 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 though of as moving one step beyond the traditional approach to profiling, using network analysis methods as its main weapon.

V. Solutions & challenges

So far so good. But how feasible is all this? It is difficult to have a precise evaluation of the sizes involved in this ambitious project. As an example, the unified call graph Hejderup et al. [8] built for Rust contained 6 million nodes and 16 million edges. But Rust as an ecosystem is *an order of magnitude smaller* than that of Java or Python. As another example, the Software Heritage Archive [9] collects about 80M projects with about 1B of revisions. 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 kinds of graphs 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.

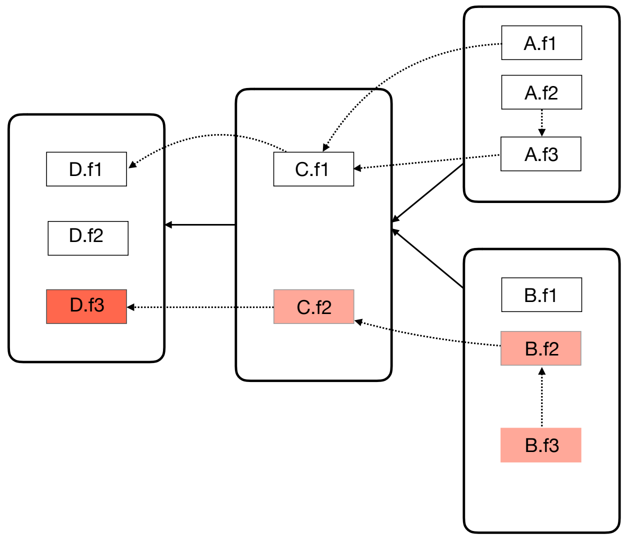

Fig. 4. The function-level view shows that if the vulnerability in Library *D* is because of function D.f3, then only C.f2, B.f2 and B.f3 are at risk. In particular, none of the functions within Library A is involved.

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 [10]. 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.

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' favourite IDE) as well as with continuous integration tools.

Although the path we described is still long and winding, the pot of gold is so precious that it totally justifies the efforts: a new world for developers where they can instantly be alerted when one of their dependencies is outdated or declared vulnerable, which functions are impacted by this, whether they are violating any copyrights, etc.—A world of total integration between software producers and the ecosystem where they live.

Acknowledgement

I want to express my gratitude to Georgios Gousios, Sebastiano Vigna and Stefano Zacchiroli for their precious help.

References

[1] R. Stallman, "The GNU manifesto," *Dr. Dobb's Journal of Software Tools*, vol. 10, no. 3, pp. 30–??, Mar. 1985. [Online]. Available: \texttt{https://www.gnu.org/gnu/manifesto.en.html}

[2] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," *Asp. Secur. Inc*, pp. 1–26, 2012.

[3] I. Kant, *Groundwork for the Metaphysics of Morals*. Oxford University Press, 2002 [1785].

[4] 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, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 102–112. [Online]. Available: https://doi.org/10.1109/MSR.2017.55

[5] 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," *CoRR*, vol. abs/1709.04621, 2017. [Online]. Available: http://arxiv.org/abs/1709.04621

[6] "FASTEN: Fine-grained analysis of software ecosystems as networks," 2019–2021, H2020-ICT-2018-2020 (Information and Communication Technologies).

[7] P. Boldi and S. Vigna, "Axioms for centrality," *Internet Math.*, vol. 10, no. 3-4, pp. 222–262, 2014.

[8] J. Hejderup, M. Beller, and G. Gousios, "Building a unified call graph at ecosystem level," Delft University of Techology, Tech. Rep. TUD-SERG-2018-002, Apr 2018, online: http://gousios.org/pub/ucg.pdf. [Online]. Available: http://gousios.org/pubs/ucg.pdf

[9] J.-F. Abramatic, R. Di Cosmo, and S. Zacchiroli, "Building the universal archive of source code," *Communications of the ACM*, vol. 61, no. 10, pp. 29–31, October 2018.

[10] P. Boldi and S. Vigna, "(web/social) graph compression," in *Encyclopedia of Big Data Technologies.*, S. Sakr and A. Y. Zomaya, Eds. Springer, 2019.