Call Graph Construction for Java Libraries

Michael Reif, Michael Eichberg, Mira Mezini

Abstract: Today, every application uses software libraries. Yet, the research which targets the analysis of libraries – independent of any application – is scarce. This is unfortunate, because for library developers, e.g., those of the Java Development Kit (JDK), it is crucial that the library behaves as intended regardless of how it is used. In this paper, we discuss the construction of call graphs for libraries that abstract over all potential library usages. Unlike algorithms for applications, call-graph construction algorithms for libraries should distinguish between analyses w.r.t. potential exploitable vulnerabilities and those related to general software quality attributes. This distinction affects the decision about what constitutes the library-private implementation, which therefore, needs special treatment. Thus, building one call graph that satisfies all needs is not sensical.

Keywords: Call Graph Construction; Libraries; Java

1 Summary

Call graphs are a major building block of static analyses. They are, e.g., directly used to identify dead methods or act as foundation for more complex algorithms, such as solvers for data-flow problems, flow-sensitive points-to algorithms, or security-related analyses. Despite the over-all presence of libraries in software development, a systematic discussion of constructing call graphs for libraries and their specific needs is missing.

Currently, the gold standard for constructing library call graphs is to use a standard algorithm [GC01], such as Class Hierarchy Analysis (CHA) or Variable-Type Analysis (VTA), and to consider all non-private methods as entry points. However, this ignores two properties that distinguish libraries from stand-alone applications. First, libraries are not closed worlds – they are extended by their users via inheritance. Second, libraries consist of classes and interfaces that define the public API and those which belong to the library private implementation, i.e., the part of the library that is only used internally and cannot be accessed by the libraries’ users. Ignoring the first property leads to call graphs that miss important edges, ignoring the second property leads to call graphs with many spurious edges. Hence, for security focused analyses everything that could be extended should be treated as extensible; for software quality oriented analyses a fine-grained identification of the library private implementation – which takes generally accepted practices into account –
should be performed. Consequently, we argue that call-graph construction algorithms for libraries must distinguish between two usage scenarios of the library.

In the first scenario, the library is assumed to be open, i.e., all non-private classes, fields, and methods can be accessed; non-final classes can be extended and non-final methods can be overridden. We use the term *open-package assumption* (OPA) to refer to this assumption and corresponding call graphs represent the *unrestricted usage scenarios* of the library. In the second scenario, only the code that belongs to a library’s public API is used or gets extended. In Java, e.g., a library’s classes and methods with package visibility do not belong to the public API. Additionally, all code that can only be reached via code that does not belong to the public API is also considered to belong to the library’s implementation; irrespective of its visibility. We refer to this case as the *closed-package assumption* (CPA).

Under CPA, the public API reflects the usage interface that library designers intend to provide to users. CPA directly reflects the generally accepted practice: *Do not add code to the namespace of a 3rd party library*, which is already mandated by the first versions of the Java Language Specification. Since then, libraries are generally developed based on this assumption, which represents the *intended usage scenarios* of the library.

We argue that *it is not possible to adequately address both scenarios by using the same call-graph algorithm*; any such algorithm would be either unnecessarily unsound or imprecise depending on the usage scenario. As a result, we propose and evaluate two call-graph algorithms for libraries w.r.t. OPA and CPA. Both algorithms: LibCHA<sub>OPA</sub> and LibCHA<sub>CPA</sub>, build upon the CHA algorithm. The first algorithm (LibCHA<sub>OPA</sub>) is sound under the open-package assumption, makes worst-case assumptions, and can be used to identify security (e.g. trusted method chaining attacks [Ko10]). However, the conservative algorithm may produce many spurious call graph edges, under CPA. This may lead to incorrect results — false positives and false negatives — when used for analyzing a library’s implementation w.r.t. general software quality attributes.

We provide the implementation and all related data of our approach here:
Implemented within the OPAL project: https://bitbucket.org/delors/opal

**References**


---

2 The part describing packages in which a developer is expected to put her code.

399 of the top 100 most popular Java libraries on Maven central (http://mvnrepository.com/popular), as of Dec. 2015.