✓ Artifact evaluated by FSE

Call Graph Construction for Java Libraries

Michael Reif Michael Eichberg Ben Hermann Johannes Lerch Mira Mezini
Technische Universität Darmstadt
Darmstadt, Germany
{lastname}@cs.tu-darmstadt.de

ABSTRACT

Today, every application uses software libraries. Yet, while a lot of research exists w.r.t. analyzing applications, research that targets *the analysis of libraries independent of any application* is scarce. This is unfortunate, because, for developers of libraries, such as the Java Development Kit (JDK), it is crucial to ensure that the library behaves as intended regardless of how it is used. To fill this gap, we discuss the construction of call graphs for libraries that abstract over all potential library usages. Call graphs are particularly relevant as they are a precursor of many advanced analyses, such as inter-procedural data-flow analyses.

We show that the current practice of using call graph algorithms designed for applications to analyze libraries leads to call graphs that, at the same time, lack relevant call edges and contain unnecessary edges. This motivates the need for call graph construction algorithms dedicated to libraries. Unlike algorithms for applications, call graph construction algorithms for libraries must take into consideration the goals of subsequent analyses. Specifically, we show that it is essential to distinguish between the scenario of an analysis for potential exploitable vulnerabilities from the scenario of an analysis for general software quality attributes, e.g., dead methods or unused fields. 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. Overall, we observed that the proposed call graph algorithms reduce the number of call edges up to 30% when compared to existing approaches.

CCS Concepts

•**Theory of computation** → *Program analysis;*

Keywords

Call Graph Construction, Libraries, Java

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950312

1. INTRODUCTION

Call graphs are used as a major building block by more advanced analyses. A frequent use of call graphs is, e.g., to combine them with control flow graphs (CFG) to form an interprocedural control flow graph (ICFG) [17, 18, 26]. These in turn build the foundation for complex algorithms, such as solvers for data flow problems (e.g., [20, 24]), flow-sensitive points-to algorithms (e.g., [10]) or security-related analyses (e.g., [4, 14]). Call graphs can also be used directly to identify dead methods (i.e., methods that are never called). On the other side, the use of libraries is ubiquitous in software development. Yet, a systematic discussion of (a) the construction of call graphs for libraries and (b) call graph algorithms targeting specific needs of libraries is missing.

Currently, the gold standard for constructing call graphs for libraries is to use a standard algorithm, such as Class Hierarchy Analysis (CHA) [9], Rapid Type Analysis (RTA) [5], or Variable-Type Analysis (VTA) [30] 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 can be 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. As we discuss in this paper, ignoring the first property leads to the construction of call graphs that miss important edges, while ignoring the second property leads to call graphs with many spurious edges. 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. In the following, we use the term *open-package assumption* (OPA) to refer to this assumption. Call graphs build based on OPA 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 by users of it. In Java, e.g., a library's classes, fields 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 a well established best practice, *Do not add code to the namespace of a 3rd*

party library. This practice is already mandated by the first versions of the Java Language Specification [12][1]. Since then, libraries are generally developed based on this assumption[2], 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.* If we did, the algorithm would be either unsound or imprecise depending on the scenario in which it is used. As a result, we propose and evaluate two call graph algorithms for libraries, one for each of the usage scenarios described. Both algorithms: LibCHA$_{\text{OPA}}$ and LibCHA$_{\text{CPA}}$, build upon the classical CHA algorithm. The first algorithm (LibCHA$_{\text{OPA}}$) is sound under the open-package assumption. It makes very conservative worst-case assumptions and can be used to identify security issues such as those that have led to trusted method chaining attacks [15]. 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. In case of a dead methods analysis, it in particular leads to false negatives. For this kind of analysis, the library developer wants to treat the library as closed. The second algorithm (LibCHA$_{\text{CPA}}$) is sound for this purpose.

To summarize, we make the following contributions:

- A motivation for call graph construction algorithms dedicated to libraries and a thorough discussion of the design space for such algorithms (Section 2).
- Two concrete call graph algorithms for libraries based on adaptations of the classical CHA algorithm (Section 3): One that can be used to identify security issues (LibCHA$_{\text{OPA}}$) and one that targets general software quality issues (LibCHA$_{\text{CPA}}$).
- A comprehensive empirical evaluation, which shows that call graphs computed by the classical CHA algorithm and those computed by LibCHA$_{\text{OPA}}$ and LibCHA$_{\text{CPA}}$ are significantly different. The evaluation supports our claims that (a) classical call graph construction algorithms (specifically CHA) do not serve the needs of either security or general quality related analyses of libraries and (b) we need two types of algorithms to address the respective needs.
- A case study (Section 5) that shows that using LibCHA$_{\text{CPA}}$ as the foundation of a dead methods analysis enables us to find ≈ 6 times more dead methods compared to a solution based on the classical CHA algorithm.

We discuss related work in Section 6 and conclude the paper in Section 7.

2. WHY LIBRARY CALL GRAPH ALGORITHMS?

In this section, we motivate the algorithms presented in this paper. We start by characterizing a library's private implementation versus its public interface. Then, we motivate the need to consider *all* possible extensions of the library by means of inheritance to construct a sound library call graph[3]. Finally, we explain that different library usage scenarios may require different kinds of analyses, which in turn need different call graphs.

[1]The part describing packages in which a developer is expected to put her code.

[2]99 of the top 100 most popular Java libraries on Maven central (http://mvnrepository.com/popular), as of Dec. 2015, adhere to this best practice; i.e., no library contributes code to other libraries and in the last case (Google Web Toolkit (GWT)) it seems as if a 3rd-party library was copied over to GWT and shipped as an integral part of it.

```
1  package l {
2    interface J { public void mj() }
3    public interface K { public void mk() }
4    class A { public void mk(){} /*API*/ }
5    class B extends A implements J,K {
6      public void mj(){} /*Impl.*/
7    }
8    class C {
9      public void mc(){} /*API*/
10     public void md(){} /*Impl.*/
11   }
12   public class D extends C {
13     public void md(){} /*API*/
14   }
15   class E implements K { public void mk()  }
16   class F { public void mj()  }
17   public class Factory {
18     public K createBK(){return new B();}
19     public Object create(){new E(); return new B();}
20   }
21 }
```

Listing 1: A Simple Library

2.1 A Library's Private Implementation

Conceptually, a library's private implementation consists of all code that a library user cannot directly use. Under the open-package assumption (OPA), a library's private implementation consists of all methods and fields with private visibility. Under the closed-package assumption (CPA), the library's private implementation additionally includes (a) every code element (class, method or field) that has at most package visibility and (b) all protected and public fields-/methods of a package visible class, unless they are indirectly exposed to the library's user. The latter happens, e.g., if the package visible class inherits from a public class or interface and overrides or implements a method declared by the super-type, or it has a subclass that is public (or implements a public interface), which inherits the respective method. In other words, a field/method of a package visible class does not belong to the private implementation, if a user can potentially directly access the field/method.

To illustrate CPA, consider the code in Listing 1. The types `A`, `B`, `C`, and `J` belong to the library-private implementation. The class `B` implements the public interface `K`, which defines the public method `mk` (Line 3). Hence, a method with declared return type K could actually return an object of type `B` (Line 18), enabling the user to call the method `mk` defined by `A` (Line 4). Therefore, `<A>.mk` belongs to the public API. In case of the public methods defined by `C` only the method `mc` (Line 9) belongs to the public API. This method is inherited by `D` and is not overridden. Hence, a user who calls `mc` on an instance of `D` actually invokes `<C>.mc`. The method `<C>.md` (Line 10) is overridden by `D` (Line 12) and therefore belongs to the private implementation.

Our approach is conservative in classifying elements as part of the library implementation in the sense that it may classify code as belonging to the public API, although a user of the library cannot actually use it. For example, the method `<E>.mk` (Line 15) would be identified as belonging to the library's public interface, because the class implements the public interface `K`. Even if `E` is never returned to a user: `E` does not escape the scope of the library. Hence, a user will never be able to invoke `<E>.mk`. However, by being conservative we ensure that we will not miss a call edge.

2.2 Covering Possible Library Extensions

Established algorithms ignore OPA usage scenarios, which is understandable given that extension code does not need to be considered, when analyzing stand-alone applications. Yet, extension code can lead to *direct call dependencies between library methods* that are not apparent from the class hierarchy. For illustration, consider a library that defines two types both providing a method `m`: An interface `I` and a class `C`, whereby `C` is *not a subtype* of `I`. Using a standard call graph algorithm such as CHA, RTA, CFA, etc. a call `<I>.m` would not be resolved against `<C>.m` as `C` is not a subtype of `I`. Yet, a user of the library may later on create a subclass of `C`, say `SubCandI`, that also implements `I`, but does not override `m`. Hence, to produce a sound call graph for the library the call `<I>.m` also must be resolved against `<C>.m`.

Hence, when constructing a library call graph to reason about OPA usage scenarios, we have to do *call-by-signature resolution* for all interface-based calls. The need to do so is exemplified by a real world security bug (CVE- 2010-0840) found in the JDK. To facilitate comprehension of the bug, we will first introduce the basics of the Java Security Model before we will discuss the attack in detail.

The Java Security Model allows to execute untrusted code safely, i.e., even malicious code cannot do any harm to the executing environment. This is required whenever a user may not trust the provided application. The Security Model is a stack-based access control mechanism that guards all sensitive actions by permission checks which verify that all the code on the current call stack is granted access to that sensitive action.

Often attacks exploit forgotten permission checks, but there have also been attacks, in which trusted code is used to call sensitive actions on behalf of the attacker. The trick to make this work is to make sure that the attacker's code is not on the call stack, when permissions are checked. This is possible, if library code accepts a callback that is provided by an attacker. But, the callback cannot be implemented by the attacker, because the implementation would be unprivileged and present on the call stack. Instead, the attacker must find a suitable callback implementation that can be configured to fit his needs.

One such vulnerability is documented in CVE-2010-0840: Attackers exploited that checks consider the permissions associated with the declaring class of the method on the call stack but not its runtime receiver type. Hence, the runtime receiver type may belong to untrusted code. For illustration, consider Figure 1. Both the interface `Entry` and the class `Expression`, belonging to the library, declare the method `getValue()`. `Expression` neither implements `Entry` nor is otherwise semantically related to it. `Expression`

[3]Unless reflection or native methods are used.

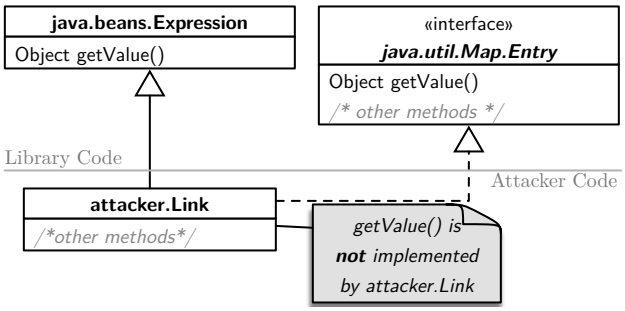

Figure 1: Trusted Method Chaining Attack

encapsulates a reflective call, whose receiver, method, and arguments are specified by arguments to its constructor. The reflective call is performed in `Expression.invoke()`, which is called from within `getValue()`.

Now, consider the following trusted method chaining attack which bypasses the Java Security Model. The attacker must create an `Expression` instance that encapsulates a `System.setSecurityManager(null)` invocation. Yet, he cannot call `invoke` on the `Expression` object himself. Instead, he defines a class `attacker.Link` that extends `java.beans.Expression` and implements `java.util.Map.Entry`, thus linking `Expression` and `Entry`: As a result, `Expression.getValue()` is now an implementation of `Entry.getValue()`. The last step is to find a library method that accepts an `Entry` object and calls `getValue()` on it, such that when this happens the attacker is not on the call stack.

```
1  HashSet<Map.Entry<Object,Object>> set =
2      new HashSet<>();
3  set.add(
4      new Link(System.class,"setSecurityManager",null));
5  JList list = new JList(new Object[] {
6    new HashMap<Object, Object>() {
7      public Set<Map.Entry<Object, Object>> entrySet(){
8        return set;
9    }}});
10 JFrame frame = new JFrame();
11 frame.getContentPane().add(list);
12 frame.setSize(50, 50);
13 frame.setVisible(true);
```

Listing 2: Set Up for the Attack

This behavior is provided by the user interface thread of AWT/Swing. This thread dispatches events, as illustrated in Listing 2. In Line 5, a `JList` object is created. A `JList` usually uses a `ListModel` to control its contents and representation. As an alternative, an arbitrary array can be used. The content representation is then based on each array element's String representation as returned by `toString()`. We add a custom `HashMap` implementation to `JList` (Line 6), which in its default implementation of `toString()` calls `getKey()` and `getValue()` on its entries accessed by `entrySet()`. The `entrySet` method (Line 7) of the custom `HashMap` returns a set containing the pre-configured `Link` instance (Line 4). Finally, the `JList` is added to a `JFrame` (Line 11) and the latter is made visible. This triggers a *paint* event which is processed by the user interface thread.

The shortened call stack of that processing is shown in Figure 2. The user interface thread transitively calls `toString()` on all its contents, when it *paints* `JList`. We provided

```
java.lang.SecurityManager.checkPermission(RuntimeP...)
java.lang.System.setSecurityManager(SecurityManager)
java.beans.Expression.invoke()
java.beans.Expression.getValue(Object)
java.util.AbstractMap.toString()
...
javax.swing.JList.paint(Graphics)
...
java.awt.EventDispatchThread.run()
```

Figure 2: Call Stack at the Permission Check

the custom `HashMap` implementation as content, which does not override `toString()`, thus `AbstractMap.toString()` is called. `AbstractMap.toString()` iterates over the entry set and calls `getValue()` on each entry. The only element of the entry set is the attacker's `Link` instance. Therefore, it effectively calls `Expression.getValue()`, which, in turn, reflectively invokes `System.setSecurityManager(null)`. Setting a new security manager is dangerous and therefore guarded by a permission check, but — as illustrated in Figure 2 — no method defined by the attacker is on the call stack, therefore access is granted.

To systematically find exploitable callback implementations, a static analysis must check that there exists no attacker callable method that transitively calls sensitive actions without proper sanitization or permission checks. Such static analyses have already been proposed [6, 7, 8, 16, 19]. However, the static analysis has to furthermore consider that calls to callbacks are resolved to *all* possible trusted implementations. State-of-the-art call-graph algorithms will not include a call edge from call sites of `Entry.getValue()` to the method `Expression.getValue()`; though this edge is required to find the attack that we presented here. If this edge is included, data flow analyses looking for unguarded paths to sensitive actions are enabled to identify the vulnerability.

2.3 Closed-Package Usage Scenarios

A call graph algorithm that considers all possible extension scenarios of the library, while being sound for analyses under OPA, is not appropriate for analyzing libraries under CPA. For illustration, consider the JRE 7's class `java.awt.datatransfer.MimeType`, which is package visible. This class belongs to JRE's private implementation, which is clearly suggested by the comment directly above the class:

> *THIS IS *NOT* - REPEAT *NOT* - A PUBLIC CLASS! DataFlavor IS THE PUBLIC INTERFACE, AND THIS IS PROVIDED AS A ***PRIVATE*** (THAT IS AS IN *NOT* PUBLIC) HELPER CLASS!*

This class defines the `public` method `match(String)`, which is not (no longer) used by code within the JDK and the class is also not exposed to the client by any means. Hence, this method belongs to the JRE's private implementation and does not have any intended users anymore; i.e., it is a dead method. JDK developers would certainly want to detect and remove such methods from the library, e.g., to improve code comprehension, to avoid useless maintenance, or to shrink the overall size of the library code. Yet, in a call graph that is constructed to cover all extension scenarios — as described in the previous subsection — the method would be treated as an entry point. Hence, an analysis searching for dead methods on top of such a call graph would not report it.

When a developer analyzes a library w.r.t. general software quality attributes, such as the presence of dead methods or dead code, he wants to treat the library as *closed*, i.e., he deliberately does not want to consider code that (eventually) extends, accesses or calls library-private code, which makes it possible to construct a more precise call graph w.r.t. the intended usage of the library. For example, a developer of a library that uses the namespace prefix $x.y.z$ to analyze the implementation of the library itself will not take into consideration what may happen if a user of the library puts code in the package $x.y.z$.

Private code is pervasive in many Java libraries. For example, the Oracle JDK 8[4] defines 8,330 ($\approx 40\%$) package visible classes. Additionally the public classes contain further 11,786 ($\approx 9\%$) package visible methods and 6,668 ($\approx 20\%$) package visible fields.

3. THE CALL-GRAPH ALGORITHMS

The proposed call graph algorithms for libraries are build on top of the Java bytecode framework OPAL[5] and are defined w.r.t. the JVM's semantics. Implementing them as bytecode analyses has two advantages. First, it makes them useable for security related analyses, because attackers can always directly craft bytecode. Second, we can also analyze libraries written in other languages such as Scala or Groovy. Both algorithms require that the bytecode of the library $\mathcal{L}$, for which we want to construct the call graph, and that of any library $\mathcal{L}_{Dep}$ used by $\mathcal{L}$ are available and can be analyzed; this includes in particular the JDK.

Both algorithms share the following main steps. First, they determine the set of entry points under their respective assumption (OPA or CPA). Second, they set each entry point method to be reachable. Third, a fixpoint computation is performed that computes the call graph. The fixpoint is computed when all methods that are marked as reachable are analyzed, which is an iterative process. For each method call found in a reachable method, both algorithms determine the set of potential call targets based on the class hierarchy; i.e., calls to methods may resolve to any subtype of the receivers static type. This resolution of call targets is the same as done by class hierarchy analysis (CHA) [9].

If the receiver type is an interface, both algorithms additionally perform call-by-signature resolution to identify those methods defined by classes that have a matching signature (name, parameter types and return type), but where the class does not inherit from the interface type. In case of OPA, all these methods defined by non-final classes are potential targets. In case of CPA, the interface and also the declaring class visibilities are further evaluated to determine if the target method is a potential target. Each method that is a potential call target is then marked as reachable.

In the following, we elaborate on the two steps of the call graph algorithms that lead to the different call graphs: (1) the computation of entry points and (2) the computation of the call targets in case of call-by-signature resolution.

[4]The classes found in the `rt.jar` of the Mac Version of Oracle JDK 8 updated 66

[5]http://opal-project.de

```
1  def isEntryPoint(declType, method):Boolean =
2    maybeCalledByTheJVM(method) ||
3    method.isStaticInitializer ||
4    (!method.isPrivate &&
5        (method.isStatic || declType.isInstantiable))
```

Listing 3: Entry Point Predicate in case of OPA

3.1 Entry Point Computation

Roughly speaking, a method is an entry point if it can be called (a) by the JVM (e.g., `finalize`) or (b) directly by a user of the library. The differences between the two algorithms are described next. LibCHA$_{\text{OPA}}$ determines if a method is an entry point by the predicate shown in Listing 3.

The first test (Line 2) identifies those methods that may be called directly by the JVM. For example, the `finalize` method is called by the JVM. Another example are serialization related methods, e.g., `readObject`, in `Serializable` classes. These methods are implicitly called by the JVM during the (de-)serialization process. These methods are often `private` and would not be considered as entry points otherwise. The second test (Line 3) checks whether the method is a static initializer. The last test (Line 4) is true if a method is non-private and static or if the non-private instance method's decaring class is instantiable. In this scenario, a class is instantiable if the class has a non-private constructor or has a factory method that potentially creates and returns instances of the class. A factory method is every static method with a return type that is a supertype (reflexive) of the class type and which calls a private constructor.

LibCHA$_{\text{CPA}}$ determines whether a method is an entry point by the logic depicted in Listing 4.

```
1  def isEntryPoint(declType,method):Boolean =
2    maybeCalledByTheJVM(method) ||
3    (method.isSaticInitializer && declType.isAccessible) ||
4    (method.isClientCallable &&
5        ( method.isStatic || declType.isInstantiable))
6
7  def isClientCallable(declType,method):Boolean =
8    (method.isPublic || method.isProtected) &&
9    (declType.isPublic ||
10       declType.subclasses.exists{ subC =>
11           subC.isPublic && subC.inherits(m)})
```

Listing 4: Entry Point Predicate in case of CPA

The first test `mayBeCalledByTheJVM` (Line 2) is the same as in case of LibCHA$_{\text{OPA}}$. The second test (Line 3) is extended and now also tests if the static initializer's declaring class (`declType`) is accessible. The latter is the case if the class or a subclass of it can be referenced from client code. In general, a class is referenced whenever the name of the class can appear in the code without violating visibility constraints. Hence, all public classes and also all package private classes that have a public subclass are immediately accessible.

Each method that does not satisfy one of the first two tests needs to be callable by a library's user (Line 4) and either must be `static` or be defined by a type that is instantiable. A method is callable (Line 7) if the given method has public or protected visibility (Line 8) and the declaring class of the method is either public (Line 9) or has a public subclass (Line 10), which does inherit the method, i.e., the method is not overridden on the path from the declaring class to the public subclass. In this case, a class is instantiable if and only if it is instantiable as in case of LibCHA$_{\text{OPA}}$ and is accessible as dicussed in the previous paragraph.

3.2 Call-By-Signature for Libraries

The algorithm to compute the edges that must be added to the call graph due to call-by-signature resolution (CBS resolution) in case of interface-based calls is depicted in Listing 5.

```
1  def cbsTargets(declIntf, mSig):Set[Method]=
2    project.findConcreteMethods(mSig).filter{m =>
3      m.isPublic &&
4      !m.definingClass.isEffectivelyFinal &&
5      !(m.definingClass <: declIntf)
6      /*in case of CPA:*/
7      &&
8      ( m.definingClass.isPublic ||
9        m.definingClass.subclasses.exists{subC =>
10         subc.isPublic &&
11         !(subC <: declIntf) && subC.inherits(m)} )
12   }
```

Listing 5: Computing the CBS Targets

Given an interface as well as the signature of a method m_{sig} defined by the respective interface, the algorithm returns a set of all methods that are *only* resolved by signature, i.e., a method with the given signature that is defined in a class that implements the interface will not be returned.

The first step, which is shared by LibCHA$_{\text{OPA}}$ and LibCHA$_{\text{CPA}}$, is to identify all call targets by finding all non-abstract, public instance methods that have the same method signature as $\texttt{m}_{sig}$ (Lines 2–13). For each such method – in the following referred to as `m` – we then check whether `m`'s defining class `C` is effectively final, i.e., whether `C` is declared `final` or if `C` only defines private constructors which makes it impossible to inherit from it. In both cases, `C` cannot be subclassed and hence, `m` cannot become a call-by-signature call target.

In case of LibCHA$_{\text{CPA}}$, we additionally check (Lines 7–12) if either `m`'s defining class ist public or if `C` has a public subclass (Line 10) which does not implement the given interface `I` and which does not override `m` (Line 11).

3.3 Summary

As argued in Section 2, depending on the goal of the analysis, we must choose the respective call graph algorithm. The two different algorithms that we presented in this section, LibCHA$_{\text{OPA}}$ and LibCHA$_{\text{CPA}}$, serve this need.

As shown in Table 1, when we want to analyze a library (be it our own or a 3rd party library) w.r.t. security issues then we must make the most conservative assumptions and this requires that we analyze the library using the open-package assumption (OPA). Additionally, we must use call-by-signature resolution related to all interface method calls to ensure that the call graph is sound. We must make these conservative assumptions when we analyze a third party library, e.g., the JRE, because it is conceivable that another library `A` that we also want to use tries to attack `JRE`. To handle these cases we use LibCHA$_{\text{OPA}}$.

Table 1: The Design Space for Library Call Graph Algorithms

Analysis Context			Closed Library Assumption	CBS
Library	Security Issues ($\text{LibCHA}_{\text{OPA}}$)	in *our* library	no (Someone will try to break our library.)	yes
		in 3rd party libraries	no (Other libraries may trie to break it.)	yes
	Software Quality ($\text{LibCHA}_{\text{CPA}}$)	in *our* library	yes (We don't care about misuses of our library.)	yes
		in 3rd party libraries	yes (We are using the 3rd party library as intended.)	yes
Application	both security and general issues		(implicitly)	no

When we want to analyze a library (be it our own or a 3rd party library) w.r.t. general software quality issues then we shall create the call graph based on the assumption that the library is used as intended by its developers. Hence, we can treat the library as closed and analyze it under the closed-package assumption (CPA) and use $\text{LibCHA}_{\text{CPA}}$. As with $\text{LibCHA}_{\text{OPA}}$, we must consider call-by-signature calls but now only those that have a relation to the public API. For example, the package visible class `F` in Listing 1 defines a method `mj` (Line 16) that is also defined by the interface `J`. Under OPA every call to `<J>.mj` must be resolved against `<F>.mj`. Under CPA, `F` belongs to the library private implementation, hence, the user cannot create a subclass of `F` that also implements the interface `J` and therefore a (library internal) call to `<J>.mj` is not resolved against `<F>.mj`.

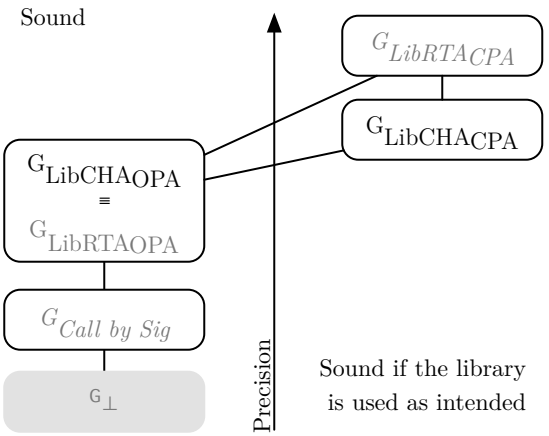

Figure 3: Precision of Call Graph Algorithms

The relation between the proposed/discussed call graph algorithms is depicted in Figure 3 and is inspired by the representation used in previous work [13]. It represents the relative precision of the algorithms compared to each other. As indicated in the figure, a call graph that is constructed using call-by-signature resolution for all types of methods calls would also be sound for libraries. As in case of applications, these call graphs are so huge that they are hardly useable [13].

The first meaningful call graph algorithm is $\text{LibCHA}_{\text{OPA}}$ which is sound but rather imprecise because the identification of the library's private implementation is most conservative. Only code that is technically not usable by any client, because it is private, is considered as belonging to the private implementation. This property makes the respective call graphs suited for security focused analyses. $\text{LibCHA}_{\text{CPA}}$, on the other hand identifies a library's private implementation based on generally agreed best practices and a thorough analysis of the visibility of the library's classes and methods. Hence, the resulting graphs are unsound if the library is not used as intended, but they much better approximate (and in non-security related cases still over approximate) all potential real runtime call graphs. Therefore these call graphs are better suited for general software quality analyses.

Theses ideas generalize to other call graph algorithms, i.e., it is conceivable to adapt other established call graph algorithms such that they can be used to analyze libraries. For RTA, e.g., we would also consider calls based on their signatures. Additionally, it is necessary to treat all classes that could be instantiated as instantiable. In case of OPA, it would result in a call graph that is identical to the one created by $\text{LibCHA}_{\text{OPA}}$. In case of CPA, it would be possible to define one scope per package and to associate all package private classes that are confined to their package with their respective scope. This could lead to a reduction in the number of call edges when compared to $\text{LibCHA}_{\text{CPA}}$.

4. EMPIRICAL STUDY

4.1 Setup

The empirical study is designed to help answer the following questions:

RQ1 Do we need call graph algorithms specialized for libraries?

RQ2 Is it necessary to distinguish between the open- and closed-package assumptions?

RQ3 Does a precise computation of the entry point set lead to a more precise call graph?

RQ4 What are the performance characteristics of the proposed algorithms?

We used the three algorithms to construct respective call graphs for a large set of libraries[6]: the 100 most used *distinct*[7] Java related libraries from Maven Central Repository [8]. The set is representative for a wide range of libraries. It contains very small (e.g., JUnit) to very large (e.g., Scala Library) libraries; libraries developed primarily in an industrial context (e.g., Guava) or in an open-source setting (e.g., Apache Commons); libraries from very different domains: testing (e.g., Hamcrest, Mockito), databases (e.g., HSQLDB), bytecode engineering (e.g., cglib), runtime environments (e.g.,

[6]The script to run the experiments can be downloaded from: http://www.st.informatik.tu-darmstadt.de/artifacts/DLC/

[7]In case of libraries that appeared multiple times in the list, we just downloaded the most current version.

[8]http://mvnrepository.com/popular (as of Dec. 2015)

Scala Runtime), containers (e.g., Netty), and also general utility libraries (e.g., osgi.core). Additionally, it contains two libraries that have unusual properties: *jsr305* and *easymock-classextesion* both do not contain a single instance method call. The jsr305 project is just a collection of annotations and easymockclassextesion only contains interface definitions and a few classes with static methods. Lastly, the set also contains libraries that are written in other languages, such as Scala (e.g., ScalaTest), whose compilers only use a subset of the JVM's concepts. The Scala compiler, e.g., does not use package and protected visibility. This significantly limits our possibilities to identify the library-private implementation (recall that $\text{LibCHA}_{\text{CPA}}$ identifies a library's private implementation based on the evaluation of the code elements' visibilities). For each library, we also downloaded all of its dependencies to build complete class hierarchies for them.

4.2 Discussion

The results of the empirical study are shown in Tables 2-5. They list only the top 5 and the bottom 5 libraries with respect to the measured effect. Furthermore, each table contains the mean as well as the standard deviation over all 100 libraries. Next, we will use this information to answer the research questions.

RQ1. To answer the first question, we compare the number of edges in call graphs computed by $\text{LibCHA}_{\text{OPA}}$ and the naïve approach. The results are shown in Table 2.

In 98 of the 100 libraries, CBS resolution introduces additional edges. These edges are missing in the graphs constructed by the naïve approach. As discussed, the lack of these edges may prevent security analyses from hinting at potential vulnerabilities. Given that CBS significantly impacts the call graph, we conclude that the naïve approach computes an unsound approximation most of the time; the exception are the projects without instance calls. This clearly supports the claim that specialized algorithms for libraries are needed. Additionally, we observe that the more precise entry point computation leads to less edges in most projects (shown in column *Type Edges* in Table 2), but the effect is small.

The experiments also show that CBS resolution does not lead to an explosion of the call edges count. The maven-plugin-api has the most significant increase in the number of call edges because it depends on three other libraries in which a lot of CBS targets are found. For example, the maven-plugin-api defines interfaces that declare methods with the following signatures: `java.lang.String getValue()`, `java.util.Iterator iterator()` or `java.lang.String toString()` and methods with these signatures are often found in the used libraries; in particular in case of `toString()`. Yet, even in this worst case, *only* $\approx 60\%$ of all edges are due to CBS resolution. The mean is only 16%, and from Table 4, we can further conclude that the number of CBS edges decreases significantly (up to 70%) when we apply the closed-package assumption.

RQ2. To answer this question, we compare the respective precision of the graphs constructed by $\text{LibCHA}_{\text{OPA}}$ and $\text{LibCHA}_{\text{CPA}}$. We did a quantitative and a qualitative comparison. The former is discussed in the following; the latter in Section 5. The quantitative evaluation compares the number of call edges in the respective call graphs shown in Table 4.

In all cases — except of the projects without instance calls — the $\text{LibCHA}_{\text{CPA}}$ graph contains less edges; sometimes up to 30% less. Surprisingly, we observe differences even in case of the Scala libraries (scala-compiler, scalatest_2.10, etc.). A manual inspection of the code revealed that the libraries contain a minimal amount of Java code, which uses package and protected visibility. Over all projects the mean call edge reduction is 9.21% and the number of edges that is added by CBS resolution is reduced by 50.06% on average. The latter is due to the fact that a client can not inherit package visible classes or interfaces under the closed-package assumption and therefore the amount of possible subtypes is lower in $\text{LibCHA}_{\text{CPA}}$. Hence, we conclude that for libraries that make use of the visibility modifiers, distinguishing between OPA and CPA is useful.

RQ3. To answer this question, we measure how many entry points are identified by $\text{LibCHA}_{\text{CPA}}$ w.r.t. $\text{LibCHA}_{\text{OPA}}$ and how this affects the call graph.

The reduction in entry points is shown in Table 3. We observe a reduction of entry points in 93 projects (in the table only the last five of these seven projects are shown). In the remaining seven cases, the entry point sets are identical; these projects all have in common that they do not declare a single package visible type and at most one package visible method. Overall, $\text{LibCHA}_{\text{CPA}}$ identifies up to 33% less entry points and the mean reduction of entry points is 8.04%. However, the effort dedicated to precise entry point computations done as part of $\text{LibCHA}_{\text{CPA}}$ are useless, if the library does not make use of package visibility.

Interestingly, the reduced number of entry points in $\text{LibCHA}_{\text{CPA}}$ does not have an effect of the same magnitude on the overall call edges, though we can still observe some effect. For example, for the hsqldb project, we observe a reduction of the number of entry points by 30% but the effect on the call graph is only $\approx 2.7\%$. This is probably due to the choice of the CHA algorithm as foundation of our algorithms; most methods that are not in the initial entry point set are still included in the call graph. It is likely that the better computation of the entry points would have a more significant effect in combination with better, e.g., context-sensitive, call-graph algorithms.

RQ4. To understand the performance characteristics of the proposed algorithms, we measured the times to compute the entry points and the call graphs. Instead of analyzing all library dependencies, we consider only the public interface of all third party libraries. Otherwise, the performance and the set of call edges would be dominated by dependent libraries ($\mathcal{L}_{dep}$). For the largest library, 82% of all methods are defined by the used libraries ($\mathcal{L}_{dep}$); in 90% of all cases, the library defines less than 5% of all methods.

The measurements were taken on an Intel i7 (2.4Ghz) with 6GB memory. The results are shown in Table 5. As expected, the proposed algorithms are slower than CHA, but still scale well up to very large libraries. The performance of the naïve implementation outperforms the two library call graphs in average by 1.15 seconds. The additional time is required to perform the more precise computations. However, this overhead is still acceptable given that the resulting call graphs are well suited for library analyses.

5. CASE-STUDY: DEAD METHODS IN THE JDK

To further understand the impact of the proposed algorithms on an analysis that builds on top of them, we con-

Table 2: Call Edges with Call By Signature

Project	Classes and Interfaces (Ifc.)				Call Edges				Ratios	
	Library		Dependencies		Naïve	LibCHA$_{OPA}$				
	Class	Ifc.	Class	Ifc.		$\sum$	By Type	By Sig.	Type Edges	CBS Edges
maven-plugin-api	22	3	28,661	3 257	8 118	19 320	8 117	11 203	99.99%	57.98%
httpcore	182	72	28,213	3 150	51 067	92 922	51 026	41 896	99.92%	45.04%
slf4j-log4j12	6	0	28,527	3 178	1 001	1 785	1 001	784	100.00%	43.92%
hsqldb	522	65	28,213	3 150	306 414	449 786	306 110	143 676	99.90%	31.88%
plexus-container-default	142	45	28,801	3 185	116 515	165 670	116 112	49 558	99.65%	29.67%
hamcrest-core	40	5	28,213	3 150	22 491	22 909	22 463	446	99.88%	1.82%
json	17	1	28,213	3 150	92 533	93 769	92 330	1 439	99.78%	1.32%
cdi-api	25	76	28,259	3 166	12 865	13 015	12 850	165	99.88%	1.15%
jsr305	5	30	28,213	3 150	88	88	88	0	100.00%	0.00%
easymockclassextension	5	2	28,213	3 150	133	133	133	0	100.00%	0.00%
mean (over all projects)									99.89%	16.78%
std dev (over all projects)									0.10%	9.90%

Table 3: Entry Points in OPA and CPA; *Pub.* is used for public and *Pkg.* for package private visibility

Project	Types		Methods		Entry Points (EPs)			Call Edges
	Pub.	Pkg.	$\sum$	Pkg.	LibCHA$_{OPA}$	LibCHA$_{CPA}$	Reduction	Reduction
lombok	58	56	242	64	161	108	32.92%	0.02%
hsqldb	189	125	4 687	1 502	4 008	2 739	31.66%	2.73%
guava	370	1 350	13 200	3 562	11 714	8 402	28.27%	0.06%
derby	997	755	23 345	3 875	17 721	13 033	26.45%	0.54%
gson	59	106	957	185	826	608	26.39%	1.47%
scalacheck_2.10	1 997	0	8 651	0	8 266	8 266	0.00%	0.00%
scalac-scoverage-plugin_2.11	172	0	1 068	0	1 006	1 006	0.00%	0.00%
scalatest_2.10	6 755	0	82 680	0	79 197	79 197	0.00%	0.00%
jsr305	35	0	30	1	15	15	0.00%	0.00%
easymockclassextension	7	0	27	0	27	27	0.00%	0.00%
mean (over all projects)							8.04%	0.41%
std dev (over all projects)							7.27%	1.47%

ducted a case study. We implemented an analysis that uses a call graph to collect all non-entry point methods that are not called by another method (excluding self-recursive calls). These methods are then reported as being dead. For the case study, we build the analysis on top of the three different call graphs constructed by the naïve algorithm, LibCHA$_{OPA}$, and LibCHA$_{CPA}$. The subject library was the part of JDK 1.7.0 update 80 that defines Java's public API, but which also contains library-private code (specifically, we analyzed the code in the packages starting with `java` and `javax`). The results are reported in Table 6.

As shown in the second row of the table, the analyses using the call graphs computed by the naïve algorithm and LibCHA$_{OPA}$ initially reported the same 218 methods, a much smaller number compared to 2,119 methods reported by the analysis on top of the call graph constructed by LibCHA$_{CPA}$. A manual evaluation of the results revealed that some methods are dead "on purpose". For example, it is a common Java idiom to define a private default constructor to ensure that no instances of the class can be created. This idiom is, e.g., used by `java.lang.Math` and always results in an intentionally dead constructor. We call appearances of this idiom technical artifacts: Adapting the analysis revealed that 114 of the initially reported methods belong in this category (cf. third row in the table).

The manual evaluation further revealed that we must filter out methods in packages starting with `javax.swing.plaf.*`. The respective classes and methods are responsible for the look and feel of Java GUIs and are — as documented in the API — generally instantiated or called by reflection. Given that our case study analysis has no support to identify reflective calls, we decided to consider all methods in the respective packages as being called using reflection, hence not dead. This filtering left us with 100, respectively 680, dead methods reported by the analyses using the naïve or LibCHA$_{OPA}$ based call graphs, respectively the LibCHA$_{CPA}$ call graph.

Next, we randomly selected 80 out of the 680 presumable dead methods to perform a manual inspection. From these 80 methods, 40 methods are also reported based on the naïve/LibCHA$_{OPA}$ based call graph. Which revealed that, 32 out of the 40 methods (80%) were correctly classified as dead. From the remaining 40 methods, one further method was misclassified. Hence, 71 ($\approx$ 89%) out of the 80 reported methods are indeed dead. The majority of the latter methods are non-private methods defined in package visible classes. Some of them were marked as deprecated, some could be clearly identified as left-over debug or test code, some were unused method overloads, and others seemed to be overlooked due to the complexity of surrounding code. In nine cases, we concluded that the reported methods are (most likely) not dead, because they seem to be called from native code or via Java's reflection mechanism.

Overall, we are confident that we found at least 550 ($\approx$ 80%) true dead methods in the core of the Java Class

Table 4: Reduction of call edges from $\text{LibCHA}_{\text{OPA}}$ compared to $\text{LibCHA}_{\text{CPA}}$

Project	Types		Methods		Call Edges				Call Edge Reduction	
					$\text{LibCHA}_{\text{OPA}}$		$\text{LibCHA}_{\text{CPA}}$			
	Pub.	Pkg.	Total	Pkg.	All	CBS	All	CBS	All	CBS
httpcore	238	16	1 652	62	92 922	41 896	65 104	15 204	29.94%	63.71%
hsqldb	446	141	10 196	1 880	449 786	143 676	342 577	43 060	23.84%	70.03%
spring-tx	168	37	1 108	70	60 808	17 551	48 331	5 595	20.52%	68.12%
groovy-all	2 905	1 497	37 150	1 467	3 125 366	794 454	2 493 057	280 747	20.23%	64.66%
plexus-container-default	182	5	1 142	17	165 670	49 558	133 112	17 000	19.65%	65.70%
scalac-scoverage-plugin_2.11	172	0	1 068	0	440 868	10 180	438 389	7 701	0.56%	24.35%
scala-compiler	8 557	37	59 147	200	14 476 580	629 696	14 408 370	561 514	0.47%	10.83%
scalatest_2.10	6 755	0	82 680	0	7 780 661	305 601	7 749 991	274 931	0.39%	10.04%
jsr305	35	0	30	1	88	0	88	0	0.00%	0.00%
easymockclassextension	7	0	27	0	133	0	133	0	0.00%	0.00%
mean (over all projects)									9.21%	50.06%
stddev (over all projects)									5.79%	15.64%

Table 5: Measured time for computing the entry point set and constructing the call graph (in seconds)

Project	Types	Methods	Naïve			$\text{LibCHA}_{\text{OPA}}$			$\text{LibCHA}_{\text{CPA}}$		
			eps	cg	$\sum$	eps	cg	$\sum$	eps	cg	$\sum$
easymockclassextension	7	27	0.0001	0.0035	0.0036	0.3590	0.6424	1.0014	0.3861	0.6415	1.0276
hamcrest-core	45	275	0.0002	0.2317	0.2319	0.3333	0.8544	1.1877	0.3824	0.8570	1.2394
json	18	128	0.0002	0.2214	0.2216	0.3362	0.8930	1.2292	0.3890	0.8687	1.2577
reflections	96	619	0.0002	0.1908	0.1910	0.3614	0.8317	1.1931	0.4026	0.8191	1.2217
aspectjrt	130	722	0.0002	0.1858	0.1860	0.3544	0.8830	1.2374	0.3885	0.8458	1.2343
groovy-all	4402	37150	0.0086	1.1409	1.1495	0.4074	2.3045	2.7119	0.4684	1.9579	2.4263
gwt-user	5497	46599	0.0092	1.5877	1.5969	0.4193	2.5968	3.0161	0.4786	2.4562	2.9348
scala-library	4899	59519	0.0127	1.4340	1.4467	0.4528	2.5445	2.9973	0.5175	2.3973	2.9148
scalatest_2.10	6755	82680	0.0160	3.2804	3.2964	1.0270	4.5554	5.5824	1.0599	4.9914	6.0513
scala-compiler	8594	59147	0.0578	5.7433	5.8011	1.1644	7.4983	8.6627	0.9313	7.6521	8.5834
mean (over all projects)					0.3826			1.5047			1.5359

Table 6: Number of dead methods found in the JDK

Algorithm	naïve/$\text{LibCHA}_{\text{OPA}}$	$\text{LibCHA}_{\text{CPA}}$
Reported Methods	218	2 119
Technical Artifacts	114	114
Swing PLAF related	4	1 325
Potentially Dead	100	680

Library using the $\text{LibCHA}_{\text{CPA}}$ based call graph.[9] Using a call graph computed by $\text{LibCHA}_{\text{OPA}}$ or the naïve call graph construction algorithm, we identified only $\approx 80 (\approx 15\%)$ of these methods.

6. RELATED WORK

Next, we first discuss general call graph algorithms before we discuss points-to analyses. The latter are either build on top of existing call graphs or (implicitly) compute them on-the-fly. After that, we discuss general approaches that analyze program fragments.

6.1 Call-Graph Algorithms

Existing call graph algorithms make a closed-world assumption, i.e., they assume the whole program is analyzed, hence do not fit the needs of libraries.

[9]These overall quality results are in line with the results reported by Eichberg et al. in [11]

Grove et al. present a framework that allows uniform modeling of multiple context-sensitive and context-insensitive call graph algorithms [13]. They distinguish three contour selection functions that allow varying levels of context-sensitivity. Thereby, a contour denotes each context-sensitive version of a procedure. These functions enabled them to extend Shivers k-CFA [25] to the more precise k-l-CFA algorithm. Contrary to their framework, that parameterizes call graph algorithms in terms of precision, we propose a framework which parameterizes library call graph algorithms w.r.t. their usage scenario.

Bacon and Sweeney [5] show that rapid type analysis (RTA) improves over CHA by only considering subtypes that are instantiated by the considered application. In particular, RTA used for applications benefits from the fact that libraries usually define many types that are not used by an application, but nevertheless will be considered for call edges in CHA. In Section 3, we outlined how our techniques could generalize to RTA.

Tip and Palsberg [31] attribute different precisions of call graphs to the number of sets used to approximate run-time values of expressions. They introduce the algorithm family CTA, FTA, MTA, and XTA. CTA uses distinct sets for classes, MTA uses distinct sets for classes and fields, FTA uses distinct sets for classes and methods, and XTA uses distinct sets for classes, fields, and methods. These algorithms can be adapted for analyzing libraries in similar ways as the adaptations discussed for RTA.

Sundaresan et al. [30] introduce declared-type analysis (DTA) and variable-type analysis (VTA). The more popular VTA uses a type propagation graph, a directed graph where nodes represent variables and edges assignments between those. Sets of possible types are then assigned to each node, representing the runtime type a variable could potentially point to. Starting with allocation sites, these type sets are propagated along the directed edges of the graph. To determine possible call targets at call sites, the type set of the receivers node is intersected with its statically possible subtypes. Both can be adapted for analyzing libraries by considering all possible subtypes for parameters of application callable methods and applying CBS at call sites where the receiver may be instantiated by the application.

6.2 Points-to Analysis

Given a points-to analysis, it is easy to generate a call graph from its results, because a points-to analysis knows the runtime types a variable — used as receiver of a call — can point to. Even if points-to analyses is an extensively investigated field of research [21, 27, 28, 29, 32, 33, 34], we are not aware of works comprehensively discussing points-to analysis for analyzing libraries only. In particular, in such a scenario not all allocation sites can be known. The correct result to what a parameter of a method callable by an application may point to is therefore not well defined; i.e., if the user creates new subtypes which extend a library class and which additionally implement a library interface. Moreover, which result is useful may depend on the use case the points-to analysis is applied to. For example, one could use a single allocation site to represent unknown allocation sites outside the library, or multiple unique allocation sites for distinct entry points into the library. While the former is cheaper to compute and useful to answer *may alias* problems, the latter yields wrong results. Contrary, for *must alias* problems the former is wrong.

Lately Dietrich et al. [10] presented a points-to analysis via transitive closure structure. They evaluate their approach on the library shipped with OpenJDK and can compute precise results in less than a minute. While they evaluate on a library only, they do not discuss whether the results remain correct in cases where a variable may point to an unknown allocation site outside the library.

Rountev and Ryder [23] present an approach to construct summary information for libraries, which assumes all possible client applications. The summaries can be applied when constructing points-to information for a client application. They show that the results of their approach are equal to those computed by a whole-program analysis. They assume clients to be able to use all *exported variables*. While these include function references, a discussion is missing as what to include in exported variables for soundness and what can be excluded to increase precision. For example, some function references must be included to avoid trusted method chaining attacks as discussed in Section 2.2 and others may be excluded under closed-package assumption to increase precision.

Allen et al. [3] discuss how to compute points-to information, when a Java library is analyzed in isolation. The core idea is to determine the so-called *most general application* (MGA) that subsumes all possible applications by using a single abstract allocation site per statically declared type of an entry point. Yet, a discussion about what the correct result of a points-to analysis should be is missing. From their description it seems that the approach misses call edges due to possible library extensions as discussed in Section 2.2, which would violate of their MGA assumption.

To recap, at first glance, many of the works published in the field seem to address the problem of computing call graphs (or the larger points-to problem) for libraries only. As argued, they turn out to address only parts of the problem; in particular, they lack systematic considerations of the issues related to inheritance of library classes.

6.3 Program Fragment Analysis

Program fragment analysis refers to analysis techniques capable of analyzing parts of a program. So far, existing work in this field of research only addresses scenarios in which the analyzed program is the application. We address in this work the opposite case: analyzing a library while not knowing the client application.

Ali and Lhoták present the tool CGC, capable of creating sound call graphs without analyzing library code [1]. It makes use of the *separate compilation assumption*, i.e., that the library has been compiled without access to the code of the application. Hence, the library cannot instantiate application classes. Building upon this work, the authors introduce the tool AVERROES, which generates placeholder code behaving as an over-approximation of the original library code [2].

Rountev and Ryder [22] present a fragment class analysis for testing. They generate a main method that over-approximates the behavior of a test suite, which enables them to apply existing whole-program analyses on a program's fragment. Their approach addresses the computation of test coverage only, making use case specific assumptions that do not hold in general.

7. CONCLUSION

In this paper, we have discussed the design space for call graph algorithms for libraries. We have in particular discussed the issues related to the use of established call graph algorithms and have discussed how to adapt the classical CHA algorithm to make it useable for the construction of library call graphs. Constructing call graphs for libraries requires – compared to the construction of call graphs for applications and components – necessarily different algorithms to satisfy the needs of different categories of subsequent analyses. As the evaluation has shown, both algorithms are necessary as the number of call edges in the call graphs differ significantly and each algorithm is able to identify unique issues. In future work, we will analyze how other established call graph algorithms can be adapted to the analysis of libraries to determine the best suited algorithm w.r.t. the precision/performance ratio.

8. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported by the DFG as part of CRC 1119 CROSSING, by the European Research Council, grant No. 321217, and by the German Federal Ministry of Education and Research (BMBF) within CRISP (www.crisp-da.de).

9. ARTIFACT DESCRIPTION

As a companion, we provide the following artifacts:

- The list of dead methods found in the Oracle JDK7 update 80 (Windows) using the three discussed approaches (naïve, open- and closed package assumption).
- The source code of the implementation of the proposed algorithm.
- A docker container which contains the compiled version of the complete OPAL framework as well as the complete runtime to (re)compile and run OPAL.

In the following we will guide you through the necessary steps to reproduce the results in the paper's evaluation. Though it is possible to build OPAL on one's own, it is recommended to use the pre-configured docker container. It contains a pre-build version of OPAL that was used as the foundation for the paper. The proposed artifact ($\approx$ 2GB) is publicly available on DockerHub[10].

9.1 Getting the Artifact

First, docker must be installed[11]. After that, it is then possible to use the docker command line tools to download the container:

```
$docker pull mreif/fse2016:evaluation
```

The container was tested using Windows 10 and MacOS X 10.11. Docker was configured with 6GB of RAM and 8 cores.

9.2 Usage

After pulling the container it can be run using:

```
$docker run -ti mreif/fse2016:evaluation
```

From now on you have multiple options. The first option is to reproduce the results of our evaluation. The second option is to generate statistics related to arbitrary call graphs, e.g., the number of call edges or entry points of a call graph.

Once the container is started, you will find yourself in the directory of the evaluation project.

9.2.1 Reproducing the Evaluation

In case you want to reproduce the paper's evaluation, you must use the following command:

```
sbt run
```

This will bring up a menu with two different options:

```
[1] CallBySignatureCountEvalationAnalysis
[2] EvaluationStarter
```

To start the evaluation you must run the `EvaluationStarter` program (number `2`). It may run – depending on your available resources – up to 1 hour. The generated data derived from all constructed call graphs will be written to the file `/home/libcg/output/results.txt` in your docker container. For subsequent analyses it is recommended to copy the file to your host system using:

```
$docker ps -alq
OUTPUT: <container-id>
$docker cp
| <container-id>:/home/libcg/output/results.txt
| <path-on-your-system>
```

[10]https://hub.docker.com/r/mreif/fse2016/

[11]https://www.docker.com/

9.2.2 Generate Custom Call Graph Numbers

If you want to analyze a custom JAR file, you must download the JAR file of the target library as well as all necessary dependencies. To retrieve them, use the `wget` tool as follows:

```
wget -P /home/libcg/customLibraries/ <url to jar>
```

Before running the analysis, make sure that you are in the directory of the evaluation project (`/home/libcg/evaluation`). You can then type `sbt` to start the `sbt` console. If you need help in specifying the parameters you can enter the `run help` command. A custom analysis would look like:

```
run -cp="/home/libcg/customLibraries/junit-4.12.jar"
-libcp="/usr/lib/jvm/java-8-openjdk-amd64/jre/lib"
-analysisMode=library_with_closed_packages_assumption
```

The custom JAR will be analyzed now. The output will be printed on the console. Note that the entry points of the call graph will be calculated from the projects on the class path (`-cp` parameter) and that the libraries specified using `-libcp` are only used to build the overall class hierarchy. The different assumptions under which the call graphs can be build can be passed via the `analysisMode` parameter. All available modes are shown in the help command of the analysis.

9.3 License

The source code of the OPAL framework and the artifacts are licensed under the BSD 2-Clause license.

10. REFERENCES

[1] K. Ali and O. Lhoták. Application-only call graph construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 688–712, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] K. Ali and O. Lhoták. Averroes: Whole-program analysis without the whole program. In G. Castagna, editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 378–400. Springer, 2013.

[3] N. Allen, P. Krishnan, and B. Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 13–18, New York, NY, USA, 2015. ACM.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[5] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.

[6] M. Bartoletti, P. Degano, and G. Ferrari. Static analysis for stack inspection. *Electronic Notes in Theoretical Computer Science*, 54:69 – 80, 2001. ConCoord: International Workshop on Concurrency and Coordination (Workshop associated to the 13th Lipari School).

[7] F. Besson, T. Blanc, C. Fournet, and A. Gordon. From stack inspection to access control: a security analysis for libraries. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 61–75, June 2004.

[8] B.-M. Chang. Static check analysis for java stack inspection. *SIGPLAN Not.*, 41(3):40–48, Mar. 2006.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In M. Tokoro and R. Pareschi, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer Berlin Heidelberg, 1995.

[10] J. Dietrich, N. Hollingum, and B. Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 535–551, New York, NY, USA, 2015. ACM.

[11] M. Eichberg, B. Hermann, M. Mezini, and L. Glanz. Hidden truths in dead software paths. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 474–484, New York, NY, USA, 2015. ACM.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 2000*. Sun Microsystems, 2009.

[13] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, Nov. 2001.

[14] B. Hermann, M. Reif, M. Eichberg, and M. Mezini. Getting to know you: Towards a capability model for java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 758–769, New York, NY, USA, 2015. ACM.

[15] S. Koivu. Java trusted method chaining (cve-2010-0840/zdi-10-056). http://slightlyrandombrokenthoughts.blogspot.de/2010/04/java-trusted-method-chaining-cve-2010.html, apr 2010.

[16] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 359–372, New York, NY, USA, 2002. ACM.

[17] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[18] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[19] J. Lerch, B. Hermann, E. Bodden, and M. Mezini. Flowtwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 98–108, New York, NY, USA, 2014. ACM.

[20] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[21] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 47–56, New York, NY, USA, 2000. ACM.

[22] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. *IEEE Trans. Softw. Eng.*, 30(6):372–387, June 2004.

[23] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *CC '01*, pages 20–36. Springer Berlin Heidelberg, 2001.

[24] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131 – 170, 1996.

[25] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 164–174, New York, NY, USA, 1988. ACM.

[26] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *ACM Trans. Softw. Eng. Methodol.*, 10(2):209–254, Apr. 2001.

[27] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM.

[28] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[29] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[30] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 264–280, New York,

NY, USA, 2000. ACM.

[31] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, New York, NY, USA, 2000. ACM.

[32] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 225–236, New York, NY, USA, 2008. ACM.

[33] G. Xu, A. Rountev, and M. Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 98–122, Berlin, Heidelberg, 2009. Springer-Verlag.

[34] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su. Efficient subcubic alias analysis for c. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 829–845, New York, NY, USA, 2014. ACM.