Getting to Know You

Towards a Capability Model for Java

Ben Hermann, Michael Reif, Michael Eichberg, and Mira Mezini
Technische Universität Darmstadt, Germany
{hermann,reif,eichberg,mezini}@cs.tu-darmstadt.de

ABSTRACT

Developing software from reusable libraries lets developers face a security dilemma: Either be efficient and reuse libraries as they are or inspect them, know about their resource usage, but possibly miss deadlines as reviews are a time consuming process. In this paper, we propose a novel capability inference mechanism for libraries written in Java. It uses a coarse-grained capability model for system resources that can be presented to developers. We found that the capability inference agrees by 86.81% on expectations towards capabilities that can be derived from project documentation. Moreover, our approach can find capabilities that cannot be discovered using project documentation. It is thus a helpful tool for developers mitigating the aforementioned dilemma.

Categories and Subject Descriptors

F3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Design, Languages, Security, Reuse, Software Engineering

Keywords

library, reuse, analysis, security, capability

1. INTRODUCTION

The efficiency of software development largely depends on an ecosystem of reuse [4, 13]. Numerous software libraries are available that solve various problems ranging from numerical computations to user interface creation. The safe use of these libraries is an exigence for the development of software that meets critical time-to-market constraints.

However, when including software libraries into their products software developers entrust the code in these libraries with the same security context as the application itself regardless of the need for this excessive endorsement. For instance, a system that makes use of a library of numerical functions also enables the library to use the filesystem or make network connections although the library does not need these capabilities. If the library contains malicious code it could make use of them.

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

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786829

In commonly used languages like Java no effective mechanism to limit or isolate software libraries from the application code exists. So developers face a dilemma: Either trust the component and finish the project in time or be secure, review the library's source code and possibly miss deadlines.

We propose to consider this excessive assignment of authority as a violation of the *Principle of Least Privilege [22]*. The principle states that every program should operate under the least set of privilege necessary to complete its job. In order to alleviate the described dilemma, we introduce an effective mechanism in this paper to detect the actual permission need of software libraries written in Java.

Drawing inspiration from Android, we construct a capability model for Java. It includes basic, coarse-grained capabilities such as the authority to access the filesystem or to open a network socket. As Java programs by themselves cannot communicate with the operating system directly, any interaction with those capabilities has to happen through the use of the Java Native Interface (JNI). By tracking the calls backwards through the call graph, we produce a capability set for every method of the Java Class Library (JCL) and by the same mechanism towards methods of a library. We can thus effectively infer the necessary capabilities of a library using our approach. We can also infer the subset of these capabilities used by an application, as it may not use every functionality supplied by the library.

As the precision of our approach is directly depending on the precision of the algorithm used to calculate the call graph of the library, we took several measures to compute a reasonably precise call graph while not compromising the scalability of the algorithm too severely.

We evaluated our approach by comparing our results against expectations derived from API documentation. We found that for 70 projects from the Qualitas Corpus [29], that we evaluated against, actual results exceeded expectations and produce a far more accurate footprint of the projects capability usage. Thereby, our approach helps developers to quickly make informed decisions on library reuse without the need for manual inspection of source code or documentation.

In our pursuit to mitigate the software developer's dilemma w.r.t. library reuse, we thus contribute the following in our paper:

- an algorithm to propagate capability labels backwards through a call graph (presented in Section 3.3),
- a labeling of native methods with their necessary capabilities to bootstrap the process (in Section 3.2),
- a collection of efficient analysis steps to aid the precision of common call-graph algorithms (explained in Section 3.3),
- an evaluation of the algorithm (Section 4) against extracted capability expectations from documentation.

We furthermore motivate our work in more detail in Section 2. An overview over related work is given in Section 5. Section 6 provides concluding remarks and discusses interesting challenges.

2. MOTIVATION

One of the major drivers of efficient software development is the ability to reuse parts of software systems in the development of other systems in the form of software libraries. Software developers can thus concentrate on the key requirements of their programs and reuse functionality that is common to many software systems. Developing software this way leaves more time for the creation of new functionality, mitigates rewriting already existing functionality and also makes errors less likely as functionality in form of libraries gets used more often than their individual counterparts. It has been observed that programming with library reuse can speed up software development by over 50% [4]. Therefore, library reuse is an important part of an efficient software engineering process.

The concept of library reuse has been adopted by many different programming environments from C and C++ over Java which ships with the Java Class Library to academic languages like Racket or even embedded languages. Library reuse is part of software development processes from the smallest wearable devices to the largest cloud-based solutions. For instance, the Java Runtime Environment is shipped with multiple libraries that are not part of the runtime but can be used by Java programs. Their functionality includes graphical user interfaces, cryptography and many more. By providing the functionality in form of libraries the Java platform gives developers the ability to use the functionality without implementing it first or leaving them out of the program if not needed. The C standard library is probably the most used software library worldwide. It is loaded in basically any system, although in different versions, from mobile devices, desktop or laptop computers to cloud computing environments.

However, in order to be efficient developers rely on the described and observed behavior of the libraries. The source code of the libraries hardly ever gets reviewed. Even in open source projects the rigor of code reviews is quite lax [3]. Examples of this can be seen in the quite prominent Heartbleed incident [7], where the server-side implementation of OpenSSL had a major programming error that allowed clients to read the server's complete memory in 64kb blocks exposing inner state that can be and was used against the server. Although being a de-facto standard with a large user base the error was not discovered through a code review but through an exploit. What makes this exploit particularly interesting is that the error was in code not necessary for the pure functionality of SSL connections but for a convenience heartbeat functionality. Developers using server-side OpenSSL never opted in to such functionality.

As library code runs in the same security context as application code a developer automatically entrusts the library code with the same privileges the end user has provided to the application. She implicitly transfers the users trust towards her to the developers of the library. Thus, any capability the complete application may use is available to library code as well.

Moreover, consider a scenario where a central point for library delivery like Maven Central gets compromised. Many applications may be shipped with malicious versions of libraries and be delivered to end users without noticing.

Considering all the aforementioned problems it is rather surprising that developers still blindly trust libraries. Besides code reviews, that may not be possible for a closed source component, very little tool support for developers in need to make the right choice of library is available. Developers can use tools like FindBugs [16] to check for well-known patterns of bad behavior in a library, but will only find instances of known problems not a full footprint of the library's critical resource usage.

As library reuse is an essential and widely adopted practice for software development it is essential that developers have access to trustworthy software libraries. Trust in those libraries can be gained by inspecting them manually – which is often tiresome – or with bug detection tools – which is often not helpful. Hence library code runs within the same security context as the application code and it is seldom under the scrutiny of a code review, it may pose a significant risk when rolling out libraries as part of an application. Thus, new methods to determine the actual usage of system capabilities of software libraries are needed.

Therefore, in this paper, we present an approach to capability inference for software libraries written in Java. In the following sections we present our approach and the bootstrapping necessary for it in detail.

3. CAPABILITY INFERENCE

We use the OPAL framework [8], an extendable framework for static analysis of Java bytecode based on abstract interpretation, to perform all static analyses relevant for our inference algorithm. After giving an overview of our approach, we will discuss it in detail.

3.1 The Approach in a Nutshell

The purpose of the analysis presented in this paper is to uncover the capability footprints of software libraries written in Java. Making them aware of these footprints helps developers to make informed decisions as whether to integrate a library into their application based on how it uses system resources. For instance, consider a library for the decoding and display of images. An application developer considering to use this library expects the library to use functionality for reading from the filesystem and for displaying images on the graphical user interface. However, it is unlikely that this library needs to perform operations to play sounds or to open network sockets. If the library does use these capabilities, a developer might want to invest more time into the inspection of the library.

In order to access system functionalities a library written in Java will have to make use of the Java Native Inter-

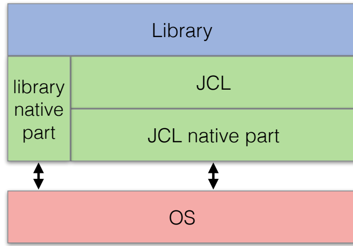

Figure 1: Architectural layers for Java libraries

face (JNI) either directly or through the Java Class Library (JCL) (cf. Figure 1). Our analysis uncovers the use of system capabilities through the JCL. It consists of three main steps.

We first manually inspected all native functions of the core part of the JCL. According to their name, implementation and documentation we assigned them capability markers manually.

These capability sets are then propagated backwards through a call graph of the JCL and the library under test. Through refinement and filtering steps on the call graph we ensure a precise but practical result. In the example presented above the native functions using the capabilities for the filesystem and the graphical user interface are traced backward to library functionality (cf. Figure 2).

In a last step the results of the analysis are consolidated to receive either a complete footprint of the library or a partial footprint depending on the use of library functionality, because only parts of the library may have been used in the application. In our example from Figure 2, the union set of all used capabilities in the library consists of the capabilities for the filesystem and the graphical user interface as expected, but it does not include capabilities for playing sound and using network sockets. Developers can thus save the time for an inspection of the library's source code.

3.2 Bootstrapping

In Java, the capabilities considered in our case, are created outside of the Java Virtual Machine and are reached only via a call through the JNI (cf. Figure 1). As any system resource like the filesystem, network sockets or the graphical user interface is a matter of the underlying operating system – when viewed from the Java side – this assumption naturally holds. As libraries seldom bring their own native code, they rely on the native functions and their wrapping functions inside the JCL. Therefore, to calculate meaningful results for arbitrary Java libraries it is necessary to first infer capability usage for the JCL.

We identified 14 distinct capabilities presented in Table 1, whose usage we want to track. They represent different critical resources that are accessible through the native part of the JCL. The `CLASSLOADING`, `DEBUG`, `REFLECTION` and `SECURITY` capabilities refer to internal resources of the JVM used to achieve dynamic code loading, instrumentation, introspection or securing code. Even if these capabilities are not system resources, we decided to include them as they represent authority to circumvent other mechanisms like information hiding, memory safety or even the security model. For instance, using reflection it may be possible for a library

Table 1: Capabilities in our approach

capability	#m	description
CLASSLOADING	24	Definition and loading of classes
CLIPBOARD	9	Access to the system clipboard
DEBUG	5	Debugging instrumentation
FS	377	Access to the filesystem
GUI	449	Graphical user interface
INPUT	10	Retrieve values from input devices
NATIVE	419	No specific facility, but calls into native code
NET	274	Network socket access
PRINT	54	Print on physical printers
REFLECTION	78	Introspective access to objects
SECURITY	14	Influence the security mechanisms of Java
SOUND	36	Play or record sound
SYSTEM	126	Operating system facilities
UNSAFE	85	Direct memory manipulation

to call into filesystem functionality although our analysis does not recognize it, because the call graph we use to extract the information does not contain a respective call edge. We also decided to include a marker capability (`NATIVE`) for all native calls that do not access system resources (e.g. `java.lang.StrictMath.cos()`). Although these functions do not provide access to a capability, they still are native functions with the possibility to read and write arbitrary memory locations in the JVM heap. All other capability names have been chosen w.r.t. the system resource they represent.

To retrieve a list of all native method stubs of the JCL, we use the OPAL framework. The core part of the JCL is included in the rt.jar file shipped with the Java Runtime Environment. Therefore, we limited our bootstrapping analysis to this file. All libraries that ship with the JCL are based on the rt.jar and those that also have native code are mainly concerned with graphical user interface display (e.g. JavaFX). We implemented a simple analysis that collects all methods with the `ACC_NATIVE` marker in the `method_info` structure [18]. This corresponds to `native` methods in Java source code. On the 64-bit version for Windows of the OpenJDK version 7 update 60 this analysis returned $1,795$ methods.

We manually assigned each of these methods the set of its capabilities. To assign the correct capabilities, we reviewed the source code of the native implementation of the method. We also took the naming and the documentation of the function into account, as they provide insight into its purpose. Column $\#m$ in Table 1 presents the number of methods for each capability in the result set of the bootstrapping process. As some methods are annotated with more than one capability, the sum of these figures is higher than the number of methods in the result set.

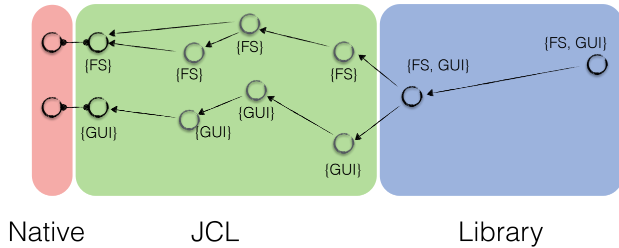

Figure 2: Call graph with annotated capabilities

3.3 Building Capability Footprints

Our analysis propagates capabilities backwards from JCL methods to the library's API (cf. Figure 2). The propagation consists of the following three major steps which are detailed afterwards:

1. The call graph that includes the JCL and the analyzed library is build.
2. The capabilities identified in the bootstrapping phase are propagated backwards through the graph. As part of the propagation, call edges in the call graph are filtered that will result in excessively overapproximated capability sets.
3. In the last step, either the complete footprint of the library (e.g. `{FS,GUI}` in Figure 2) or a footprint based on the usage context of the library is build.

Call Graph Construction Step.

We build the call graph using a variant of the VTA algorithm that we implemented in OPAL. Compared to the original VTA algorithm [26], the algorithm only propagates precise type information intra-procedurally and does not first calculate a whole-program *type propagation graph*. However, intra-procedurally the analysis is more precise as it uses the underlying abstract interpretation framework and is therefore flow- and path-sensitive. This way we are able to construct an equally precise call graph, but be more efficient at the same time. To further increase precision, we added two steps. First, we added a shallow object-insensitive analysis of all `final` or `private` fields to OPAL to determine a more precise type of the values stored in the respective fields than the field's declared type. Second, we added a basic data-flow analysis of all methods that return a value to determine a more precise return type than the declared one if available. Both information is used during the call graph construction. Overall, the number of call edges of the resulting call graphs are comparable, but our VTA algorithm is more efficient than the original one.

For the call graph's entry points we use all non-private methods and constructors, all static initializers and those private methods that are related to serialization. The latter are added to capture the implicit calls performed by the JVM when serializing or deserializing objects.

Propagation Step.

In the bootstrapping phase, we manually assigned each native method of the JCL the set of its capabilities. To determine a library method's capabilities, we propagate the capabilities through the call graph by traversing the call graph backwards, starting with the native methods of the JCL. While traversing the graph, we propagate the initial sets of capabilities to compute the set of capabilities attached to each method of the JCL and the library. At the end, each method of the library is annotated with a set of its capabilities based on the transitive native calls it performs.

Unfortunately, this naïve propagation of the capabilities results in a hugely overapproximated capability distribution as basically every method of the JCL would be annotated with all capabilities. This is due to the fact that often no precise type information for the receiver of a method call is available, so all possible receivers have to be taken into account by the analysis although these receivers are never called.

For instance, if the `toString()` method is called by some method and the runtime type of the receiver is unknown and, hence, the upper type bound `java.lang.Object` needs to be assumed, the call graph will contain $1,304$ edges[1] for this call.

These edges will propagate capabilities to methods that will in practice never be reached. Hence, we filter these problematic edges during the capability propagation step. For instance in the above example, it is rather unlikely that each of these $1,304$ alternative implementations will be called in practice from this call site. However, as all of these implementations are annotated with their respective capability sets the method including the call site will be annotated with the union set of all these sets. We decided for filtering although it gives up soundness of the analysis by replacing the overapproximation with an underapproximation. We ignore calls to unlikely implementations and thus get a much stricter capability propagation that is closer to runtime reality. Alternatively a point-to analysis could have been used to determine more exact receiver types, but as our approach needs to scale up to the JCL such a costly analysis is not permissible. We implemented two filters in our approach.

The first filter removes edges for method calls where the receiver type is `java.lang.Object`. Listing 1 shows an example where this filter effectively removes problematic call edges. The method `currentContent()` calls `toString()` on the field `o` of the receiver object. Although the field is initialized with the concrete type `MyInt`, the call cannot be resolved to the implementation in this type, because the field is public and may be overwritten with an object of any other type. This results in call graph edges to all implementations of `toString()` of subclasses of `java.lang.Object`. Our filter removes these edges from the capability propagation.

```
1  public class A {
2      public Object o;
3
4      public A() { o = new MyInt(42); }
5
6      public String currentContent(){ return o.toString(); }
7  }
```

Listing 1: Example for inconclusive type information

The second filter handles a similar problem that occurs with the use of interface types and abstract classes. Like receivers, whose type cannot be refined more precisely than `java.lang.Object`, receivers of interface and abstract types also have a large number of subtype alternatives within the

[1]The `toString` method is implemented by $1,304$ JCL classes.

Table 2: Effect of filter application in OpenJDK

filter type	call edges	reachable methods
Without filter	2.068.946	102.896
Object filter	1.221.293	102.411
Abstract filter	1.974.261	101.656
Interface filter	1.322.949	98.728
Subtype filter	1.194.172	91.813
all filters	368.231	91.135

JCL. Again, the call graph algorithm does not have any other choice than to include edges to all these alternatives, resulting in a over-propagation of capabilities.

However, in this subtype filtering process we perform a prefix match on the package name of callee and caller. We only propagate capabilities for alternative edges that point to a callee located in the same package as the caller, as we assume that the most relevant implementations often reside in the same package. For example, consider a call to the `compare` method of the `java.util.Comparator<T>` interface in some method of a class `C`. Further assume that the implementation of `compare` to which the call dispatches is inside an anonymous class located inside `C` and is thus implicitly in the same package as `C`. When detecting a call to the `compare` method our filter only accepts implementations in the same package, which in this case is only the one in the anonymous class.

Without filtering, the call graph of the OpenJDK version 7 update 60 (Windows, 64bit) includes roughly $2M$ edges and over $100k$ methods that transitively call into native methods. The exact figures are presented in Table 2. Applying the first filter regarding receivers of type `java.lang.Object`, will drop $800k$ methods ($\sim 40\%$). When using abstract receiver types for the second filter, the considered edges for capability propagation drop only to $1.9M$, but when we apply the filter to interface type receivers, the number drops to $1.3M$ edges. If we combine interface and abstract receiver types (named *Subtype filter* in Table 2), we have only about $1.2M$ edges. Finally, when combining both kinds of filters, we only consider about $370k$ edges. This reduces the processed edges by 82% and the number of methods in the inverse transitive hull by 13%.

As discussed before, overapproximation in the analysis produces overly populated results in our approach. Due to that we choose to filter, which has the subsequent effect that we find a capability set of a given library that is closer to the runtime situation. For instance, a simple call of the `read` method and `InputStream` as receiver type would imply the capabilities `CLASSLOADING`, `SOUND`, `NET`, `UNSAFE`, `FS`, `SYSTEM`, `SECURITY` and `GUI` which are eight out of the 13 capabilities in our model. In case of the more concrete type `FileInputStream`, the call of `read` would only yield the `FS` capability. Hence, we are underapproximating to that end that instead of assuming that all call edges are possible, we only consider edges where caller and callee are in the same package. Furthermore, we decided to apply our filter mechanism only for callees within the JCL as we did not find large sets of alternative call edges inside libraries.

Result usage and reporting.

The last step in the analysis is the construction of a report of the results. Depending on the developer's needs we can construct two different reports. First, by building the union set over all methods of a library we can construct the complete capability footprint of a library. Second, when taking the usage context of a library into account, we can report on the actual used capabilities as developers might not always use a library to its full extent and thus may not use capabilities included in the complete footprint.

Beyond our motivating usage scenario, where this information is displayed to the application developer to decide on libraries to use, it may also be interesting in other situations. Considering the example in our motivation regarding the Heartbleed incident, we could also display the difference between the complete footprint of a library and the used part. This can hint to features of a library that are unused by the application, just as the heartbeat feature was not used in the OpenSSL scenario. Also this information could be used to slice the library down to the used part in order to prevent unintended or malicious use.

The information we infer can also be interesting for library developers. We already calculate a list of methods and the assigned capabilities which can be used to support code inspections for libraries. Because through this list developers have assistance to identify and find critical methods with ease, our approach can help to guide code inspections and make them more efficient.

By analyzing and collecting the information from open source libraries in a database, we could also build a recommendation system that, based on metrics like the delta from expectation or the number of used capabilities, could assist developers to find a suitable library.

4. EVALUATION

In this section, we present the measures taken to evaluate the capability inference in our approach. We are guided by these research questions:

RQ1 Does the capability inference algorithm effectively find capability sets equal to developer's expectations of library capability usage?

RQ2 Does the capability inference algorithm exceed these expected sets by more true positive values?

Setup.

Developers make educated guesses on the capabilities used in a library when integrating it into their application. A math library, for example, should not use the network or the filesystem. To check such an assumption developers can leverage the documentation of the library. However, to save time developers might perform only a key phrase scan to search for the presence (or absence) of certain terms. In our evaluation, we mimic this process and use techniques from Information Retrieval to scan documentation for key phrases.

We use a subset of the Qualitas Corpus [29] incorporating 70 Java libraries. They were selected based on two criteria. First, the documentation for the library must be accessible. Second, we choose libraries with a low number of dependencies on other libraries, so that capabilities used by dependents will not influence the outcome of the experiments. That is because we assume that capabilities used by dependencies will only be documented in the original docu-

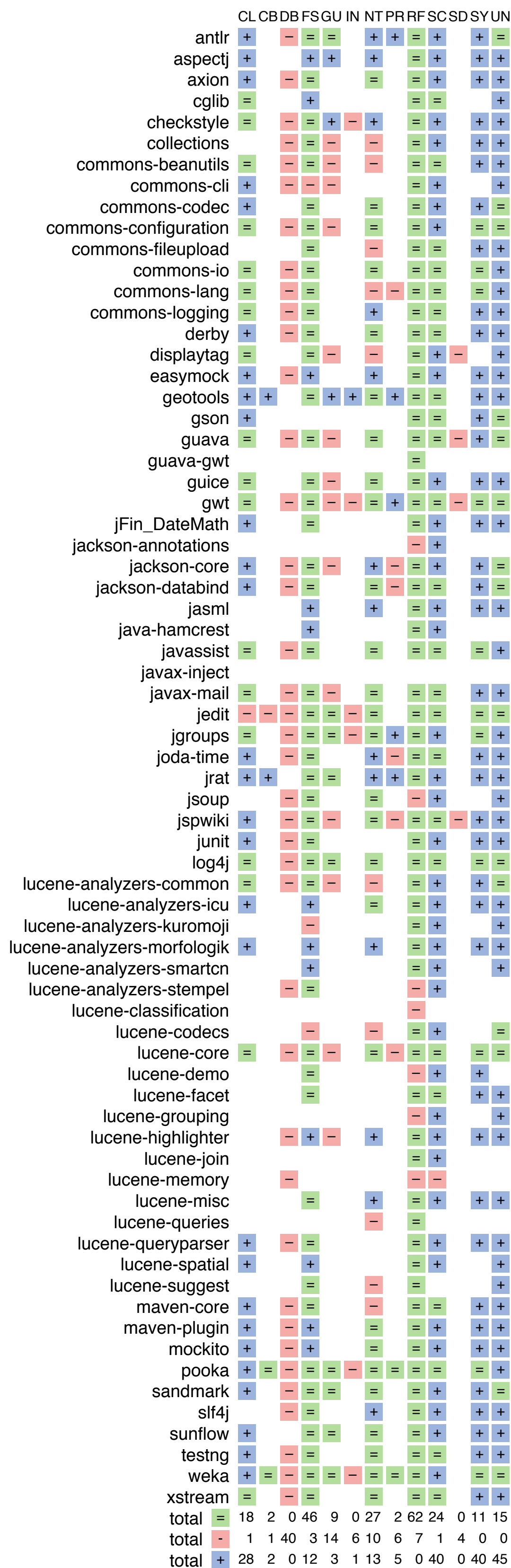

Figure 3: Capability-usage observed in libraries

mentation of that library and not in the documentation of the library depending on it.

The selected libraries have different sizes and capture various domains. Ranging from ANTLR, which is a parser generator, to Weka, a collection of machine-learning algorithms, or Pooka, a mail client with user interaction, multiple domains are captured in the selection. Therefore, the capability expectations for these libraries vary as different domains require different capabilities. For instance, a developer will expect pooka to use the `GUI` capability, because it should display information for the user, while she would not expect it from ANTLR.

As all libraries need the Java Class Library to work, we set the baseline for all our experiments to the 64-bit version of the OpenJDK version 7 update 60 running under Windows.

We constructed a list of key phrases for each capability (cf. Table 3). For this, we used terms in the English language, that can be assigned to the capabilities without ambiguity between capabilities. We derived these key phrases from the documentation of the JCL functions we annotated manually in the bootstrapping phase. As the `NATIVE` capability is just used as a marker, we do not assign any keyword to this capability.

In order to search in API documentation for key phrases, we first obtain the documentation either by downloading the complete set of documentation or by requesting it directly from the projects webserver. We then traverse the complete documentation starting at `allclasses-noframe.html`, which contains a complete list of all available classes, abstract classes and interfaces in the project. For each of the listed classes we extract the documentation given for each method of that class. This text as well as the method signature for reference is passed to Apache Lucene.

Lucene creates an inverted index using all tokens found inside the provided text. We then use this index to search for the key phrases in the key phrase list. Depending on whether we search for a word, e.g., ”classloader”, or a phrase e.g., ”load a class”, we use either a `TermQuery` or a `PhraseQuery`. While term queries have support for stemming, phrase queries do not. Stemming is a process where a term is reduced to its root, the stem. For example, the term ”paint” would match ”painting” as well as ”painted”. The use of stemming results in a more complete set of hits for this term.

For each project we record the capabilities whenever the respective key phrases are found in the documentation. The result of this process is a union set over the complete project and hence is expected to be the complete footprint of the library.

Results.

The results of the evaluation are presented in Figure 3. For each project a line in the diagram shows the expected and found capabilities. It uses the abbreviations already introduced in Table 3. A green square with an equality symbol (`=`) denotes that the capability was expected from the documentation and also found with the capability inference algorithm. A red square with a minus symbol (`-`) means that the capability was expected from the documentation, but not found. A blue square with a plus symbol (`+`) represents a capability that was found by the inference algorithm, but was not expected from the documentation. Empty squares represent capabilities that where neither found in documentation nor inferred.

Table 3: Expected key phrases for capabilities

capability	key phrases
CLASSLOADING (CL)	classloader, load a class, classloading, jar
CLIPBOARD (CB)	clipboard, paste
DEBUG (DB)	jvm, debug, intrumentation, debugging
FS (FS)	filesystem, file, folder, directory, path
GUI (GU)	textfield, menubar, user interface, canvas, paint, editor, color, rendering, render
INPUT (IN)	press key, mouse, keyboard, trackball, pen
NATIVE (N)	
NET (NT)	network, socket, port, tcp, udp, ip, host, hostname, protocol, connection, http, imap, pop3, smtp, ssl, mail, transport, mime
PRINT (PR)	printer, paper
REFLECTION (RF)	reflection, class, field, method, attribute, annotation
SECURITY (SC)	security, security provider, policy, privilege, authority
SOUND (SD)	sound, midi, wave, aiff, mp3, media
SYSTEM (SY)	system, command line, execute, process, thread, environment, runtime, hardware
UNSAFE (UN)	pointer, memory address, memory access, unsafe

For example, for the ANTLR project, the `FS`, `GUI`, `REFLECTION`, and `UNSAFE` capabilities were expected from key phrases in the documentation and also found through capability inference. The documentation also suggested the `DEBUG` capability, but no evidence for this was found in the call graph. Moreover, the `CLASSLOADING`, `NET`, `PRINT`, `SECURITY`, and `SYSTEM` capabilities were found, but not expected from documentation.

We measured the agreement of our capability inference with the results obtained from the key phrase search. For the agreement we consider every capability the analysis and the key phrase search agree upon – both positively and negatively. For example, for the ANTLR project the agreement is 87.50%. The mean agreement[2] in the inspected project set was 86.81% while the algorithm missed only 3.90% of the capabilities detected from documentation.

When looking closer to individual capabilities, we see a similar result (cf. Table 4). The only outlier here is the `DEBUG` capability.

Moreover, the capability inference algorithm was able to find 14.1% more capabilities than documented in mean over all projects. It found evidence for undocumented use for all capabilities except `DEBUG`, `REFLECTION`, and `SOUND`. The `UNSAFE` capability was found in 1.8 times more projects than it was documented.

[2]As the values per project are already normalized, we use the geometric mean. Zero values are set to 0.001.

Table 4: Results by capability

capability	agreement	miss	excess
`CLASSLOADING`	97.62%	2.38%	66.67%
`CLIPBOARD`	98.53%	1.47%	2.94%
`DEBUG`	42.86%	57.14%	0.00%
`FS`	94.83%	5.17%	20.69%
`GUI`	79.10%	20.90%	4.48%
`INPUT`	91.30%	8.70%	1.45%
`NET`	82.46%	17.54%	22.81%
`PRINT`	90.77%	9.23%	7.69%
`REFLECTION`	90.00%	10.00%	0.00%
`SECURITY`	96.67%	3.33%	133.33%
`SOUND`	94.29%	5.71%	0.00%
`SYSTEM`	100.00%	0.00%	133.33%
`UNSAFE`	100.00%	0.00%	180.00%

Figure 4 shows the capability distribution of each capability over all projects. For the `SYSTEM` and `UNSAFE` capability it can be seen that every documented capability use was successfully detected and for most other capabilities the figures are very close. However, in contrast to Table 4 we present the true positives and not the agreement here, so true negatives are not represented in the figure.

Discussion.

First and foremost, we are interested in meeting the developers expectations (RQ1) of the capability usage of libraries. As our approach has a mean agreement of 86.81% over the inspected projects, we clearly reach this goal. The capability inference only misses a mean of 3.90% in our experiments.

What is even more interesting is that the capability inference systematically discovers usage of capabilities that are not to be expected by the developers when they use documentation as a source of information (RQ2). While a mean of 14.1% more capabilities over all inspected projects were found only shows this as a tendency, when looking at technical capabilities like `CLASSLOADING` (66.67%), `SECURITY` (133.33%), `SYSTEM` (133.33%), and `UNSAFE` (180.00%), it is apparent that the use of these capabilities by a library is often not documented.

These technical capabilities may give hints towards whether a library might be vulnerable to exploits. For instance, as it may not be surprising that Junit is using the `CLASSLOADING` capability, developers might not expect this from jFin_DateMath, which is a library for financial date arithmetic. However, in our evaluation results both libraries use the `CLASSLOADING` capability although not suggested by the project documentation. A quick inspection of jFin_DateMath revealed that two methods both named `newInstance`[3] use the `forName` method of `java.lang.Class`. According to Oracle's Secure Coding Guideline (Guideline 9-9)[1], however, this method performs its task using the immediate caller's class loader. As both implementations of the `newInstance` method do not perform further checks on the provided class name and simply pass it to the `forName` method, attackers in an untrusted security context could use this so-called *Confused Deputy* [14] in order to execute code in the library's

[3]In `org.jfin.date.daycount.DaycountCalculatorFactory` and `org.jfin.date.holiday.HolidayCalendarFactory`.

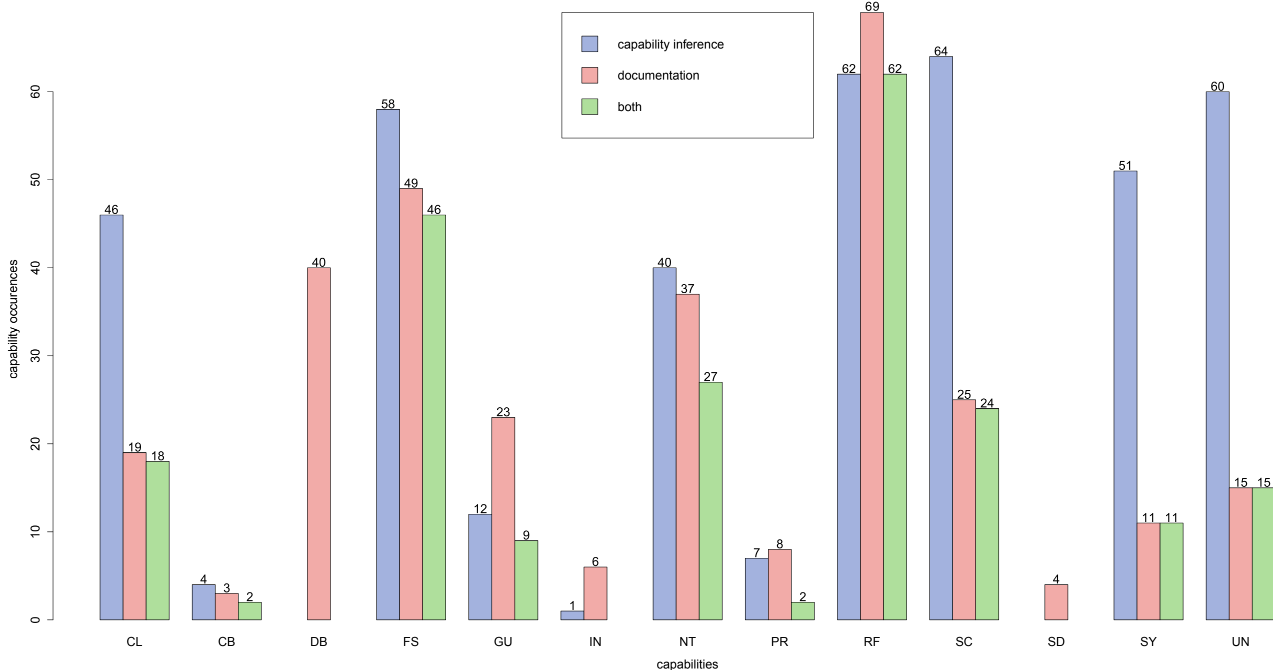

Figure 4: The capability distribution over the projects

context given that it is more privileged than the context of the caller.

Results show that our capability inference algorithm works well for most of the capabilities. The capability with the least convincing results regarding its agreement is the `DEBUG` capability. Our approach misses every occurrence found through key phrase scanning. While we choose a rather narrow set of methods for this capability during the bootstrapping phase, the key phrases assigned to the capability occur in comparatively many method documentations. For instance, the phrase "Method that is most useful for debugging or testing; ..." occurred in the documentation of Jackson-core. While this phrase clearly does not imply any kind of JVM debugging on the methods part, it still is included in the results for the term query for the "debug" key phrase included in the key phrase list for the `DEBUG` capability. However, when we apply our algorithm to the jvm monitoring tool jconsole, we are indeed seeing the `DEBUG` capability included in the results. In general, this observation also applies to the `SOUND` and to the `INPUT` capability. This is what leads us to the conclusion that documentation might not be precise enough in these cases.

However, we receive rather good results for `CLIPBOARD`, `FS`, `NET`, `PRINT` and `REFLECTION` where we find most of the capabilities and even undocumented ones. This indicates that our capability inference algorithm is a helpful approximation. Moreover, the excellent results for the capabilities `CLASSLOADING`, `SECURITY`, `SYSTEM` and `UNSAFE`, where the inference significantly exceeds the expectations, are a very interesting outcome. It may indicate that API documentation is not a reliable source of information to build capability expectations in a security aware context.

Most projects in our evaluation set make good use of capabilities; one project is an exception. The Javax-inject project shows no capability usage w.r.t. our inference, but also no expectations can be extracted from the documentation. This is because the project consists entirely out of interfaces, which by definition cannot have concrete methods up to Java 7. We deliberately included the project in the evaluation set to inspect whether the documentation of this project may raise expectations. As it did not raise any expectations and also by its contents could not use any capabilities, it is a good example for true negatives.

As the evaluation answers both research questions positively, we conclude that our approach is effective in helping to mitigate the developer's dilemma when making library choices for applications.

Threats to validity.

We only used API documentation for the extraction of developer expectations. However, there are alternatives, which we did not consider. In an open source setting, for instance, one could inspect the source code. Another alternative would be to statically analyze the code for the presence or absence of specific, interesting calls, as e.g., static bug checkers do. Manuals or other documents could also be a source of information to developers. However, we think that capability information should be denoted clearly in the public API documentation of a project.

We constructed the key phrase list manually from the observations we made while inspecting the source code and the documentation of the annotated native methods in the bootstrapping phase. It is possible that we missed key phrases that denote capabilities in API documentation. Moreover, it is possible that we included key phrases that are not suitable

to detect capabilities and introduce many false expectations. Finally, the API documentation of the projects could be outdated [10], incomplete, or even wrong. In order to mitigate this threat, we reviewed the key phrase list and the results of search queries to remove questionable key phrases.

There are also threats to validity with regard to our inference approach. First, we are aware that we might introduce false negatives through our filtering phase. However, we use these filters to reduce false positives by removing call graph edges introduced by incomplete receiver type information. If we get rid of the filters while keeping our coarse-grained capability model, we would receive a result, where almost every time the complete set of possible capabilities would be found in every library. In order to mitigate the risk of introducing false negatives, we limit the filter to be only applied for receiver types inside the JDK. Second, the analysis we use is unaware of everything that is beyond the scope of the used call graph algorithm, so that we do not detect calls made through reflection and direct manipulation of the heap through native code.

The largest manual step in our approach is the bootstrapping of native methods of the JCL. As we inspected the implementation and documentation of these methods manually, we may have documented too less, too much, or incorrect capabilities for methods, even though we reviewed the dataset multiple times. In future work, we would like to exchange this process with an automated analysis of the syscalls made in the implementation. Beyond the process being more efficient, it is also repeatable for new versions of the JCL and applicable to native code shipped with libraries. It might also be more reliable in terms of precision.

5. RELATED WORK

As Java was designed with the idea of loading code dynamically, it was also equipped with an elaborate security mechanism to protect clients from malicious code [12]. It is based on runtime, stack-based access control checks w.r.t. policies provided by the platform, code owners or users. However, writing these policy files has proven to be challenging for developers and thus Koved et al. [17] derived a method (and improved it in [11]) to infer these access rights so that the resulting policies honor the Principle of Least Privilege [22]. Their algorithm traces calls to methods for permission checks in the Java security architecture just as we trace back actual native calls. Hereby, they can effectively create a set of necessary permissions. Nevertheless, they rely on the assumption that every critical resource is indeed effectively protected with a runtime check, as they trace the check and not the resource itself.

The Android platform provides a permission-model for the authorization of apps. When installing an app on an Android-based device, the system explicitly asks the user to confirm a list of permissions granted to the app. Similar to our approach, a coarse-grained permission model is used to represent system resources. Developers have to supply a manifest with their app listing every used permission in form of a list of strings. For example, the string `android.permission.READ_CONTACTS` represents the permission to read the contacts on the device from the app. If the app then calls a platform function related to a permission the Android runtime checks if the permission has indeed been granted to the app.

Similarly to Java policies, it is also hard for developers to write manifests that only include necessary permissions. Accordingly, Bartel et al. [2] follow a similar approach to the one provided by Koved et al. With their COPES tool they were able to discover a significant number of apps that suffer from a gap between declared and used permissions. Also, as Sun et al. [25] point out, the majority of the Top 50 apps are shipped with native libraries. Building on their outstanding work for the Java Native Interface [24, 27, 28, 23], they build an isolation model for native libraries in Android apps effectively protecting users from malicious or faulty native code.

An alternative for developers would be to isolate libraries and control their effective capabilities by means of the object-capability model [21]. For Java currently two approaches to the model exist. First, there is the older J-Kernel system by Hawblitzel et al. [15, 31] and, secondly, the newer Joe-E system by Mettler et. al. [20]. Both approaches are in their nature constructive and thus require excessive change to the complete system including the JCL, which might not be possible in industry environments.

The protection of applications or platforms from malicious code is also the focus of active research from a system perspective. Cappos et al. [5] construct a Python-based sandbox with a small, isolated kernel library. By this, they prevent attackers from leveraging bugs in the kernel library for privilege escalation. Moreover, De Groef et al. [6] are concerned with the integrity of the execution process itself. As they point out, Write-XOR-Execute protection mechanisms of operating systems cannot be applied for applications with Just-in-time compilation as the runtime needs to write into and execute the same memory block. Their system separates sensitive from non-sensitive code and protect the system by blocking sensitive code.

Intrusion detection works in a similar way to our approach as there system calls are analyzed, which correspond to native calls in our approach. There it has been well established as a useful indicator. Maggi et al. [19], for instance, use sequence and arguments of system calls with a behavioral Markov model to detect anomalies. Whereas, we want to understand critical resource usage, they go beyond and want to detect abnormal usage patterns.

Of course, developers can use tools like FindBugs [16] to determine a footprint of a library, but will only find issues for patterns that have already been included in the bug checker. Other tools like HAVOC-LITE [30] or ESC/Java [9] are able to determine a resource usage of a library, but first have to be configured with the entire set of resources and the methods to reach them.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a novel approach towards a capability inference for Java. It is based on a manually created dataset of native methods and their capabilities in the JCL. An automated process propagates these capabilities along a call graph through the JCL and a library in order to find the capability footprint of the library. To produce helpful results, we added a refinement step to the used call graph algorithm and also apply filtering for alternative edges during capability propagation. Although filtering may impede the soundness of the approach, the produced results are closer to expectation.

Being able to produce a capability footprint for a library helps developers in their task of selecting libraries for their projects. With our analysis, a developer can get answers towards capability usage in a few seconds instead of inspecting the source code or the documentation manually. We show that the approach is able to find capabilities expected from documentation with an agreement of 86.81% . Moreover, we show that the approach exceeds these expectations by finding 14.1% more capabilities than expected from the documentation and produces a more accurate footprint of a library's actual capability usage.

We plan to study the accuracy of the approach regarding developer expectation more closely with the help of a user study. Developer experts will be asked to document their expectations regarding multiple software libraries. By this, we will be able to further determine the utility of the approach for secure software development.

Furthermore, we plan to infer the capabilities for native methods by an automated process rather than the manual process presented in this paper. A static analysis on the native implementations of the methods can extract and trace system calls in order to achieve a reproducible result for new versions of the JCL. Furthermore, it is useful to include native library code into the process.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers of the paper and the replication package for their helpful comments.

This work was supported by the BMBF within EC SPRIDE, by the BMBF within the Software Campus initiative (01IS12054), by the BMBF within ACCEPT (16BY1206E) and by the Hessian LOEWE excellence initiative within CASED.

This work is part of the PEAKS project.

8. LIBRARIES USED IN EVALUATION

We selected 70 projects, shown in Table 5, from the Qualitas Corpus that have a non-generated, non-empty API documentation.

Table 5: Project selection used for evaluation

project	version
antlr	v4.5
aspectj	v.1.8.5
axion	v1.0-M2
checkstyle	v6.3
cglib	v3.1
collections	v4.0
commons-beanutils	v1.9.2
commons-cli	v1.2
commons-codec	v1.10
commons-configuration	v1.10
commons-fileupload	v1.3.1
commons-io	v2.4
commons-lang	v3.3.2
commons-logging	v1.2
derby	v10.1
displaytag	v1.2
easymock	v3.3.1
jFin_DateMath	v1.0.1
jackson-annotations	v2.5.1
jackson-core	v2.5.1
jackson-databind	v2.5.1
jasml	v0.10
javassist	v3.19.0-GA
java-hamcrest	v2.0.0.0
javax-inject	v1
javax-mail	v1.5.2
jedit	v5.2.0
joda-time	v2.7
gson	v2.3.1
gwt	v2.7.0
guice	v4.0-beta
guava	v18.0
guava-gwt	v18.0
geotools	v0.9.0
jgroups	v3.6.1.Final
jrat	v1-beta1
jspwiki	v2.10.1
jsoup	v1.8.1
junit	v4.12
log4j	v1.2.17
lucene-*	v4.10.3
maven-core	v3.2.5
maven-plugin	v3.2.5
mockito	v1.9.5
pooka	v2.0_080505
sandmark	v3.1.1
slf4j	v1.7.9
sunflow	v0.07.2
testng	v6.8.21
weka	v3.7
xstream	v1.4.7

9. REPLICATION PACKAGE

Our implementation of the capability inference algorithm has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations.

We provide the replication artifact at the following URL:

`http://www.st.informatik.tu-darmstadt.de/artifacts/peaks-capmodel/`

9.1 Introduction

The purpose of this manual is to guide you through the steps necessary to run our analysis and reproduce the results in the paper's evaluation. As we make assumptions about the folder structure in our software, it is important to check whether the JAR files of the pure analysis and the evaluation are located in the same folder as the resource folder. This is the case, when you use the replication package ZIP file provided by us. When you build from source, please make sure this is the case. The `resource` folder contains the mapping of native calls to capabilities and the `rt.jar` file of the unofficial windows build of the OpenJDK 1.7 update 60.

9.2 Prerequisites

The analysis and the evaluation tooling is written in Scala and compiles down to Java Bytecode. Thus, it will run on every machine with a Java 1.8 runtime installed. We were successfully able to run it using the Windows 8 and MacOS X 10.10.3 operating systems. For reproducing the evalua-

tion charts and statistics you may need R[4] and a standard browser (we recommend using Firefox).

9.3 How to use the Tool (Quick Software Manual)

If prerequisites are met you can start our tool from the command line interface. Open a console and navigate to the folder where you placed the jar file and the resource folder. If you want to run the analysis you have to provide a project that you want to test. Therefore, you have to specify it via the `-cp` parameter. To execute the jar and analyze the "xstream" library you type in the following command:

```
java -jar PEAKS_JavaCapAnalysis.jar
     -cp=resources/projects/xstream
```

Notice that "xstream" is not a .jar but a folder. If you pass a folder to the analysis every class file and jar in this folder will be included to the analysis. If you start the analysis, there will be a menu.

```
[1] Start capability analysis for libraries.
[2] Sliced capability analysis for projects.
[3] Help.
```

The normal usage (used in the paper) is the first option. If you only provide a project it will print capability set to the command line. You could specify other parameters if you are only interested in some capabilities or if you want get the methods which transitively use certain capabilities. Use the third option to get an overview over all available parameters.

Option 2 from the menu above is a bit more advanced. The analysis will only take care of the actually used part of the libraries in an application. So far, this works only under the assumption that all the application dependencies are packaged in the same jar as the application. As this is the common case for the deployment of most applications it should fit most development processes nicely.

9.4 Reproducing the Evaluation

This section describes how to reproduce our evaluation results.

Requirements.

Note that you will need a connection to the Internet as we used the online documentation for some libraries. If you work without an Internet access the capability set from the keyword scan of libraries with online documentation will be empty and the results will differ from ours. Please make sure that the `PEAKS_Eval_JavaCapAnalysis.jar` file is located in the same folder as the `output` and `resources` folder.

Reproducing the Evaluation.

You can start the evaluation by opening a console. Then navigate to the folder where the jar is located and execute the following command:

```
java -jar PEAKS_Eval_JavaCapAnalysis.jar
```

After executing this command, a menu will appear. Press `1` to trigger the evaluation process. The evaluation will start now, the output will be written to the output folder. The evaluation can take up to 6 hours (depending on your computer). Due to heavy parallelization of the OPAL Framework your computer can be quite busy while running the evaluation. You will then find the results in the `output` folder in the file `EvaluationResults.csv`. You can compare them to our results from the `Evaluation` folder. If the evaluation is done and you may want to recreate the capability distribution chart (cf. Figure 4) you have to execute the jar again. Choose the second menu item this time. It will trigger a transformation of the `EvaluationResults.csv` to another representation needed to execute the R script. This new file is also located in the `output` folder and is named `transformedResults.csv`. The R script (`peaks.R`) is located in the `Evaluation` folder. If you want to use it, adapt the working directory to the `output` folder before. To create the capability matrix (cf. Figure 3), copy the `EvaluationResults.csv` file from the `output` folder into the `Evaluation` folder and open the `capmap.html` file with a browser. As Google Chrome (and other browsers) suppress locally run java scripts from loading files (in this case the CSV), we suggest Mozilla Firefox or running a small webserver (e.g. with NodeJS).

[4]`http://www.r-project.org/`

10. REFERENCES

[1] Secure coding guidelines for java se. `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`, 2014.

[2] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 274–277, 2012.

[3] M. Bland. Finding more than one worm in the Apple. *Communications of the ACM*, 2014.

[4] B. W. Boehm. Managing Software Productivity and Reuse. *IEEE Computer*, 32(9):111–113, 1999.

[5] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, 2010.

[6] W. De Groef, N. Nikiforakis, Y. Younan, and F. Piessens. Jitsec: Just-in-time security for code injection attacks. In *Benelux Workshop on Information and System Security (WISSEC 2010)*.

[7] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[8] M. Eichberg and B. Hermann. A software product line for static analyses: the OPAL framework. In *SOAP '14: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, New York, New York, USA, June 2014. ACM Request Permissions.

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. 37(5):234–245, 2002.

[10] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33. ACM, 2002.

[11] E. Geay, M. Pistoia, T. Tateishi, B. G. Ryder, and J. Dolby. Modular string-sensitive permission analysis with demand-driven precision. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 177–187. IEEE Computer Society, May 2009.

[12] L. Gong. Java security architecture revisited. *Communications of the ACM*, 54(11), Nov. 2011.

[13] M. L. Griss. Software Reuse: From Library to Factory. *IBM Systems Journal*, 32(4):548–566, 1993.

[14] N. Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[15] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. *USENIX Annual Technical Conference 1998*, 1998.

[16] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2004.

[17] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. *ACM Sigplan Notices*, 37(11):359, Nov. 2002.

[18] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. *Addison-Wesley*, 1997.

[19] F. Maggi, M. Matteucci, and S. Zanero. Detecting Intrusions through System Call Sequence and Argument Analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2008.

[20] A. Mettler, D. Wagner, and T. Close. Joe-E: A Security-Oriented Subset of Java. *NDSS 2010*, 2010.

[21] M. S. Miller. Robust composition: towards a unified approach to access control and concurrency control. 2006.

[22] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[23] J. Siefers, G. Tan, and G. Morrisett. Robusta: Taming the native beast of the jvm. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 201–211, 2010.

[24] M. Sun and G. Tan. Jvm-portable sandboxing of java's native libraries. *Computer Security–ESORICS 2012*, 2012.

[25] M. Sun and G. Tan. NativeGuard: protecting android applications from third-party native libraries. In *WiSec '14: Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM Request Permissions, July 2014.

[26] 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.

[27] G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang. Safe java native interface. In *In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.

[28] G. Tan and J. Croft. An empirical security study of the native code in the jdk. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 365–377, Berkeley, CA, USA, 2008. USENIX Association.

[29] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, Dec. 2010.

[30] J. Vanegue and S. K. Lahiri. Towards Practical Reactive Security Audit Using Extended Static Checkers. In *2013 IEEE Symposium on Security and Privacy (SP) Conference*, pages 33–47. IEEE, 2013.

[31] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, D. Hu, and D. Spoonhower. J-Kernel: A Capability-Based Operating System for Java. In *Computer Aided Verification*, pages 369–393. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.