

TECHNICAL REPORT:

REMEDY OPEN API JAR FILE ANALYSIS

WITH OBJECTIVE OF PORTING TO ANDROID OPERATING SYSTEM

Prepared for Pyrasoft, Inc.

by Mark Bearden, March 2011

Summary

The work described here began with the goal of porting the BMC ARS Server client-side java library (version 7.5) to the Android smart-phone platform, for the purpose of creating a client Android app that communicated with an ARS Server instance using the BMC Java library. The port turned out to be challenging and has not yet been accomplished successfully.

This report summarizes the challenges uncovered, known remedies that resulted in a “partial port” of the library, and work remaining to be done toward a fully functional port. This report can serve as a guide to future parties making a similar attempt.

The list of documented issues:

- BMC jar file (v 7.5) contains duplicate .class files, making it incompatible with Android jar-to-dex conversion tools. (A solution is given.)
- BMC jar file executes Java VM version lookup routine that is incompatible with the Android (versions 1.6 to 2.1) implementation of the Java VM version string. (A solution is given.)
- Excessive resource requirements during jar-to-dex conversion performed using Eclipse with Android SDK extensions. (A solution is given.)
- BMC jar file contains too many classes, exceeding in practice the per-Android-app limit on total number of Java classes defined. (A partial solution is described.)
- BMC jar file contains dependencies on Java API classes and methods that are not included in the Android platform. (Partial work on a solution is described.)

Potential uses of this report:

- This report provides guidance to anyone who rewrites the arapi75.jar file to run on an Android mobile device:
 - Based on the attempt documented here, it appears that the best approach would require access to the arapi75.jar source code. This report shows that the majority of the Java classes packaged in the arapi75.jar library are in one third-party package, namely the Spring framework; yet only a single dependency path may exist for the Spring framework, through just one class in the 'core' classes of the ARS Server API. If that dependency path could be eliminated through modification of the library's source code, then most of the code in the arapi75.jar library (that is, the entirety of the Spring framework classes) could be removed. This would solve most of the problems encountered during the attempt to port the library to Android. Note that this suggestion is based only on static dependency analysis, as noted in the body of the report.
 - Another possible approach, though apparently difficult at best, would be "repackaging" certain missing Java core classes that are needed by the arapi75.jar library, but which are not present in the scaled-down Android Java core libraries.
- This report identifies techniques and tools that might be applied to porting other large enterprise Java libraries, besides the BMC ARS Server library, to the Android mobile device platform.
- This report identifies weaknesses in the current publicly available toolsets for analysis of Java package and class dependencies (namely the limitation of finding just static dependencies, while not detecting dynamic

dependencies that exist through run-time resolution of object instance types). Unfortunately from the standpoint of analysis, the ARS Server API makes significant use of the dynamic type of dependency.

Problem Background

BMC Software, inc. provides an open Java library for development of client-side programs that interface with BMC's Action Request System, or ARS (also known as Remedy) [1,2]. It was desired in this work to develop an Android phone app that utilizes this Java library to communicate with an ARS server.

The work described here was performed using version 7.5 of the BMC ARS client library. Android apps were targeted for the 1.6 version of the Android platform and were execution-tested using 1.6 (emulation) and 2.1 (actual hardware) versions of the Android OS. The Java client library consists of these jar files:

File Name	Size
arapi75.jar	7.3 MB
arapiext75.jar	182 KB
log4j-1.2.14.jar	359 KB

Issue #1: Duplicate .class Files

The arapi75.jar file contains a number of duplicate .class files. This is tolerated by most standard Java VMs, but it is not supported by the `dx` tool that must be executed within the Android development environment, to convert third-party jar files to the DEX format that executes in the Android Dalvik Java VM. (If duplicate .class files are not removed, the `dx` tool generates an error saying that a particular class can't be added to the list of converted classes because it's already there.) The arapi75.jar file can be edited to remove the duplicate class files using a ZIP file format editing tool such as WinRAR, WinZip, or 7zip.

This list below is a **partial** list of the duplicate class files that must be removed from arapi75.jar. In most cases, the duplicated .class files are not identical. Rather, the duplicates have different sizes and creation dates. During the conversion effort described here, the strategy was used of retaining the newest version (based on creation date) of each file, and deleting all older versions.

Duplicated .class files:

- com\bmc\thirdparty\org\apache\commons\collections\ArrayStack.class
- com\bmc\thirdparty\org\apache\commons\collections\Buffer.class
- com\bmc\thirdparty\org\apache\commons\collections\BufferUnderflowException.class
- com\bmc\thirdparty\org\apache\commons\collections\FastHashMap.class, along with
 - FastHashMap\$1.class
 - FastHashMap\$Values.class
 - FastHashMap\$KeySet.class
 - FastHashMap\$EntrySet.class
 - FastHashMap\$CollectionView.class
 - FastHashMap\$CollectionView\$CollectionViewIterator.class

Issue #2: Incompatible Java Virtual Machine Version Reporting

The arapi75.jar file contains a (repackaged) class com.bmc.thirdparty.org.apache.commons.lang.SystemUtils, that contains code that is executed upon library initialization. The following two methods in that class are not compatible with the Android platform's Dalvik Java VM:

- `SystemUtils.getJavaVersionAsFloat()`
- `SystemUtils.getJavaVersionAsInt()`

Specifically, these methods contain code that parses the VM version string reported by the Java runtime, in order to produce a numeric version of the runtime. The Dalvik Java runtime reports no version number (null or empty string), and so the original parsing code throws an exception and interrupts Android app execution. This problem was resolved by reverse compiling the `SystemUtils.class` file, making the source modifications shown in Appendix B (to report a version value of 0 for the Dalvik error case), and then replacing the original `.class` file with the recompiled version. In the source code in Appendix B, the comments containing the string "BMC-for-Android fix" identify the changes that were made.

Issue #3: Resources Exceeded During .apk App Compilation

Ensuring Eclipse has Enough Memory for JAR to DEX Conversion

When converting a large jar file, Eclipse appears to need a **lot** of memory in its Java VM. Eclipse is running in a JVM that has its memory resources (heap) capped at some default amount. On my system, and this seems to be typical default values, the memory cap was set thus:

```
-Xmx256M  
-Xms40m
```

For some help on what those settings mean, Google on "eclipse.ini Xmx Xms".

At first, attempts to include the `arapi75.jar` library in an Android app built with Eclipse resulted in out-of-memory errors, in combination with "dex" conversion to Dalvik format failure messages. The problem was not insufficient system RAM, for there was plenty of that free—it was the internal limit the Java VM was imposing on Eclipse. The problem was resolved by changing the memory cap settings for the JVM to the following values, which are typically added to an `eclipse.ini` file, usually found in the same folder that contains the main eclipse launcher executable. Note that for the Motorola Android SDK, the file has been renamed the file from 'eclipse.ini' to 'motodevstudio.ini'. The changed settings are:

```
-Xmx512m  
-Xms256m  
-XX:PermSize=128m  
-XX:MaxPermSize=256m
```

This seems important to note when working with large Java libraries and Eclipse: When Eclipse runs out of memory during the jar conversion, the resulting error is often a "random" error about some build step failing, rather than an explicit "out of memory" indication—which error messages can be rather misleading.

Issue #4: Resources Exceeded During .apk App Installation on Android Phone

After working through the problems noted above, a test App was successfully created that referenced (and included into the produced `.apk` file), the `arapi75.jar` file.

When this first App was installed on an Android mobile device (a standard Motorola A85 Droid), the installation failed. The installation log contained the following error that was generated by the phone's runtime system, during the installation attempt:

```
dexopt: "LinearAlloc exceeded capacity"
```

Research done on Google's Android support discussions revealed that the arapi75.jar file contains more classes than are allowed to be defined within a single Android app, as of the time of this work. This is a problem since the Android SDK's `dx` tool converts all of the classes contained in any third party .jar referenced during the build of an Android app. (No dependency analysis is performed by the Java SDK or installer to minimize the included set of classes.)

This problem appears to require removing some of the classes from the arap75.jar API library. Two possibilities are presented for reducing the class count in the arapi75.jar file:

1. Breaking the ARS library into two parts, an app-side library, and an Android service to be installed along with the app. The app and the service would each contain a subset of the arapi75.jar file's classes. This approach would require access to the arapi75.jar source code, in order to code new inter-object communication (using Android messaging APIs) between the generated between the library and the service. In other words, this would require converting the arapi75.jar sources into two purely Android-compatible projects. It would also result in two App installers for the resulting program, one for each project, since the separated service would have to be packaged in its own App (.apk) installer to avoid the too-many-classes limit encountered at App installation time. This approach was not investigated due to its scope, and to the author's having no access to the arap75.jar source code. (Also, as discussed below, it was not workable to reverse-compile the arapi75.jar file to obtain equivalent source code.)

2. Eliminating some of the classes in the ARS library. This possibility was investigated, based on the observation that a *very* large number of 3rd party classes were "packaged" into the arapi75.jar file. The possibility was noted that some of these classes packaged are not actually needed at run-time, but rather BMC's software developers included all the files just for convenience (since no normal enterprise Java runtime environment has the restrictions on jar file size that the Android platform presents).

It was initially hoped that some of the classes in arapi75.jar could be identified as superfluous to the ARS client API's operation, and that a subset of such superfluous classes could be removed to shrink the JAR file. This approach was *partially* successful in getting a subset of the arapi75.jar functions to execute correctly on an Android mobile device within a test App. This *partially* successful work is described below.

Formal Static Analysis and "Shrinking" Using ProGuard

Static analysis of arapi75.jar was performed using ProGuard 4.4 [4], with the aim of automatically "shrinking" the API (also supported by ProGuard) in order to remove superfluous classes not needed at runtime. The limitation of this approach is that only static class references are detected. As an example, consider this dependency of Java class A upon class B:

```
class B { }
class A { B instanceOfB = new B(); }
```

Such dependencies are detected by ProGuard. However, the following style of *dynamic* dependency is not detected by ProGuard analysis:

```
class B { }
class A { Object instanceOfB = ClassLoader.getInstanceOf("B"); }
```

Unfortunately for the goals of this project, the arapi75.jar library relies heavily on dynamic dependencies, in part due to frequent use of the Factory design pattern. This was determined based on reverse-compilation of the JAR file, and it was confirmed by run-time missing-class errors that resulted after ProGuard was used to automatically "shrink" the JAR file to contain just the classes that were detected by the static analysis. Specifically, the ProGuard shrinking experiment was performed by building a test App that called just the ARServerUser.createEntry() method

of the arapi75.jar library, and by providing that method, and its parameter types, as entry point to the ProGuard shrinking configuration.

The tested ProGuard 4.4 configuration is given for reference in Appendix A.

Thus the automated exclusion of classes based on forward static analysis using ProGuard failed to produce a running App. The App installed and started, but when ARS Server API calls were made, there were 'Class not found' exceptions thrown at runtime. At this point in the work, these run-time exceptions were tracked and "patched" by explicitly adding the missing classes using "keep" directives in the ProGuard shrinking configuration. This approach achieved a limited success, **resulting in a test Android App that successfully executed a simple set of ARS login and record read operations.**

But as additional API calls were added and exercised in this test App, run-time 'Class not found' exceptions were eventually encountered for Core Java system classes that are not present in the Android runtime (and were also, of course, not present in the original arapi75.jar file, since these would be provided by a standard Java virtual machine with J2EE support). This separate issue is documented below as "Issue #5".

However, even if problem of unsupported Java calls had not arisen, it was decided that formal dependency analysis in order to automatically "shrink" the arapi75.jar file would not be practical, due to the widespread use of dynamic class references. No available Java platform tools could be located to perform run-time dependency analysis of the classes, in order to detect these class dependencies due to dynamic references. The ad-hoc approach that was partially successful, as described above, would not scale to cover the entire arapi75.jar API, because of the large number of classes (thousands) and class/method dependencies (at least tens of thousands) that would need to be manually tested.

Issue #5: Dependencies on Unsupported Java API Calls

The arapi75.jar file, unfortunately, has Java class dependencies on many classes that are not included in the Android runtime. Further dependency analysis was performed to identify the unsupported classes, in order to determine if a workaround is achievable.

The first effort was an informal manual analysis of the BMC jar file, with the following results.

Manual Analysis of the arapi75.jar File Contents and Dependencies

The following analysis was performed by manual inspection of the arapi75.jar file, using an unzip utility to examine the contents of the JAR file.

Only 39% of the classes in the jar are part of the BMC ARS Server API. The other 61% are re-packaged third party classes consisting entirely of open-source Java libraries. (Repackaging these other libraries into one large JAR file along with the BMC AR Sever API classes removes problems of version dependencies when the arapi75.jar file is deployed in heterogeneous Java runtimes.) The distribution of classes defined in the arapi75.jar file is as follows, and is illustrated in Figure 1:

- Total number of classes in the arapi75.jar file: 5,061 (~7.5 MB jar-compressed)
 - "CORE" API, com.bmc.arsys.* 1,967 classes (~2.5 MB jar-compressed)
 - Third-party repackaged classes: 3,094 classes (~3.5 MB jar-compressed)
 - com.bmc.thirdparty.org.springframework.* 1,943 classes (~2.2 MB)
 - com.bmc.thirdparty.org.apache.* 986 classes (~1.2 MB)
 - other com.bmc.thirdparty.org.*: 165 classes (~100 KB)

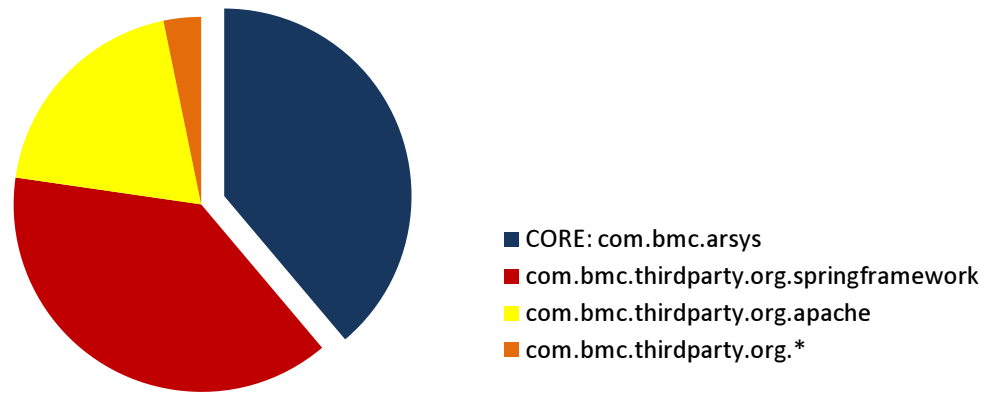


Figure 1. Contents of arapi75.jar by Package (Proportions from Compressed .jar size)

Formal Static Analysis Using DependencyFinder

Package-level static analysis was attempted using DependencyFinder 1.2.1 [3] to determine which packages (namespaces) have dependencies on unsupported libraries. This analysis would be at best incomplete, due to the possibility of dynamic class references (as discussed above), and the fact that DependencyFinder is also not able to detect the dynamic-reference-driven dependencies. But it provides at least a minimum view of dependencies.

The highest-level view of dependencies among these packages is given in Figure 2 below. It is noteworthy that within the "core" of the arapi75.jar, that is, the classes implemented within the `bmc.com.arsys.*` package space, there are no missing dependencies for the Android (Dalvik) Java runtime. **All of the detected missing run-time dependencies are due to third-party classes that are repackaged.** This seems to raise the possibility that some, even most, of the ARS Server API might be executable on an Android Dalvik VM, depending on whether or not the ARS Server API calls depend upon the particular repackaged third-party classes that, in turn, depend on core Java classes not supported by Android.

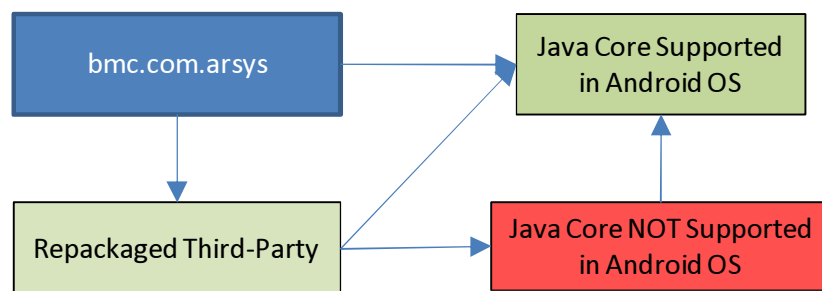


Figure 2. Dependencies Between `bmc.com.arsys.*`, Repackaged Third-Party classes, and Java Core

Appendix D shows the detected package-level dependencies between "core" ARS Server API classes (in `arsys.api` package namespace) in more detail. Appendix E lists (most of) the detected dependencies at the class level between the repackaged third-party classes within `arapi75.jar` and Java classes not supported by the Android OS. In summary, approximately 150 third-party repackaged classes depend on these standard Java package spaces that are **not supported** on the Android Dalvik virtual machine:

- `java.applet`
- `java.beans`
- `java.rmi`
- `javax.naming`
- `javax.faces`

- javax.transaction
- javax.servlet
- javax.xml.rpc
- javax.xml.transform
- javax.persistence
- javax.mail
- javax.jms
- javax.management
- javax.resource.cci
- javax.rmi.CORBA
- javax.activation

It also appears significant that **all of the Spring framework static dependencies occur through just one class in the ARS Server "core" API classes, namely the `arsys.utils.CatalogReader` class.** If this class's use of the Spring framework could be eliminated, then it might be possible (depending on the presence, or not, of undetected dynamic dependencies) to remove most of the classes in `arapi75.jar` by excluding the repackaged Spring framework in its entirety.

Dealing with the Missing Java System Classes

We can identify the following two approaches to deal with the missing runtime classes.

1. Repackaging the missing classes. Unfortunately, the present state of the art is poor regarding tools for repackaging. Repackaging of Java core system classes is problematic at best; most repackaging tools are designed only to automate the repackaging of an entire jar inside another Jar. That's not an option for this problem; instead, we need to repackage a subset of Java system classes. But those system classes will tend to depend on Java core classes, leading ultimately to repackaging most or all of the Java runtime classes into a modified `arapi75.jar` file! This is unlikely to be workable, in part because the `java.beans` subsystem is one of the unsupported items, and it has dependencies on `java.awt`, which is completely unsupported on Android (and unsupportable, due to even deeper dependencies on `java.sun.*` classes, while Android has a completely different GUI implementation from Standard Java.) Thus it seems likely that simple repackaging will fail.

Making this challenge even harder, there appears at present to be a lack of maintained utilities for this. The JarJar utility from Google [7] appears to be the most current, but after about three hours of effort, it would not execute on a Windows XP system. An examination of the support forums at Google seem to indicate that the latest JarJar utility is being maintained poorly or not at all. Several similar error reports have been reported, with no response from the JarJar development team.

2. Removing classes from `arapi75.jar` that depend on classes not supported in Android. The other approach is hopefully removing a subset of the `arapi75.jar` classes that depend on the missing API classes. Such an approach would inevitably lead to compromised functionality of the `arapi75.jar` class. The most optimistic case is that it is only "non-critical" functionality that depends on the missing classes. It would seem that a quality solution would involve rewriting at least a part of the `arapi75.jar` classes, either from original source code or from functional reverse-compiled code.

This approach would require access to the `arapi75.jar` source code, because no Java decompile utility could be identified that would produce correct code from the `arapi75.jar` class files. Here again, current support for Java utilities seems to be slipping. The best results at decompilation resulted from A. Neshkov's DJ Java Compiler (version 3.11) [5] and E. Dupuy's JD-Core Decompiler (version 0.6.0) [6]. Yet neither of these produced correct code that could be recompiled after modification to remove dependencies on unsupported classes, even were that possible.

Conclusion

In conclusion, due to Issue #5 (dependencies on Java classes not supported in the Android OS), only a rewrite of the arapi75.jar library from source code appears feasible for producing a complete port, given the state of currently available Java dependency analysis, class repackaging, and class decompilation utilities.

On the other hand, it appears that with a great deal of effort, a partial port of some of the ARS Server API to Android may be feasible. The amount of effort to determine how much can be ported, and to achieve the port, might be significant, and might have to rely on either ad-hoc analysis, or the creation of new Java analysis tools.

References

- [1] <http://www.bmc.com/>
- [2] http://wikipedia.org/wiki/Action_Request_System
- [3] <http://sourceforge.net/projects/depfind/> (or <http://depfind.sourceforge.net/>)
- [4] <http://sourceforge.net/projects/proguard/> (or <http://proguard.sourceforge.net/>)
- [5] <http://members.fortunecity.com/neshkov/>
- [6] <http://java.decompiler.free.fr/>
- [7] <http://code.google.com/p/jarjar/>

Appendix A: ProGuard version 4.4 Settings Used for Shrinking arapi75.jar

```
-injars '7.5-API-From-Server\C_Program Files_BMC
Software_ARSystem_Arserver_api_lib\arapi75_fixed_dups.jar'
-outjars '7.5-API-From-Server\C_Program Files_BMC
Software_ARSystem_Arserver_api_lib\arapi75_shrunk.jar'
-dontskipnonpubliclibraryclasses
-dontskipnonpubliclibraryclassmembers
-dontoptimize
-dontobfuscate
-keeppackageNames
-dontwarn
-ignorewarnings
# Implements: Runnable interface
-keep class ** extends java.lang.Runnable {
    ;
}
# Subclasses: java.util.AbstractMap abstract class
-keep class ** extends java.util.** {
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.collections.ArrayStack {
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.collections.BufferUnderflowException {
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.collections.FastHashMap** {
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.collections.map.LinkedMap {
    ;
}
-keep class com.bmc.arsys.api.ARServerUser {
    ;
    (...);
    *** createEntry(...);
}
-keep class com.bmc.arsys.api.ARNativeAuthenticationInfo {
    ;
}
-keep class com.bmc.arsys.api.Entry {
    ;
}

-keep class com.bmc.arsys.api.ARException {
    ;
}
-keep class com.bmc.arsys.api.Proxy {
    ;
    ;
}
-keep class com.bmc.arsys.api.ARTypeMgr {
    ;
    ;
}
```

```

-keep class com.bmc.arsys.api.NoPrefixToStringStyle {
    ;
}
-keep class com.bmc.arsys.api.** {
    ;
}
-keep class com.bmc.arsys.arrypc.** {
    ;
}
-keep class com.bmc.arsys.utils.** {
    ;
}
-keep class com.bmc.arsys.arenencrypt.** {
    ;
}
-keep class com.bmc.arsys.arthreadlocal.** {
    ;
}
-keep class com.bmc.thirdparty.org.acplt.oncrpc.** {
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.configuration.BaseConfiguration.** {
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.configuration.** {
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.logging.impl.** {
    *** run(...);
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.logging.** {
    *** run(...);
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.digester.** {
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.beanutils.** {
    ;
}
-keep class com.bmc.thirdparty.org.apache.commons.codec.** {
    ;
}
}
# Keep names - Native method names. Keep all native class/method names.
-keepclasseswithmembers,allowshrinking class * {
    native ;
}
# Keep names - _class method names. Keep all .class method names. This may be
# useful for libraries that will be obfuscated again with different obfuscators.
-keepclassmembers,allowshrinking class * {
    java.lang.Class class$(java.lang.String);
    java.lang.Class class$(java.lang.String,boolean);
}

```

Appendix B: Modified Source Code for com.bmc.thirdparty.org.apache.commons.lang.SystemUtils

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Source File Name: SystemUtils.java

package com.bmc.thirdparty.org.apache.commons.lang;
import java.io.PrintStream;

public class SystemUtils
{
    . . .
    private static float getJavaVersionAsFloat()
    {
        if(JAVA_VERSION_TRIMMED == null)
            return 0.0F;

        // BMC-for-Android fix:
        // Moving these inside the try/catch, because on
        // Android platform they don't work; exception is thrown.
        //
        // String str = JAVA_VERSION_TRIMMED.substring(0, 3);
        // if(JAVA_VERSION_TRIMMED.length() >= 5)
        //     str = str + JAVA_VERSION_TRIMMED.substring(4, 5);
        try
        {
            String str = JAVA_VERSION_TRIMMED.substring(0, 3);
            if(JAVA_VERSION_TRIMMED.length() >= 5)
                str = str + JAVA_VERSION_TRIMMED.substring(4, 5);

            return Float.parseFloat(str);
        }
        catch(Exception ex)
        {
            return 0.0F;
        }
    }

    private static int getJavaVersionAsInt()
    {
        if(JAVA_VERSION_TRIMMED == null)
            return 0;

        // BMC-for-Android fix:
        // Moving these inside the try/catch, because on
        // Android platform they don't work; exception is thrown.
        //
        //     String str = JAVA_VERSION_TRIMMED.substring(0, 1);
        //     str = str + JAVA_VERSION_TRIMMED.substring(2, 3);
        //     if(JAVA_VERSION_TRIMMED.length() >= 5)
        //         str = str + JAVA_VERSION_TRIMMED.substring(4, 5);
        //     else
        //         str = str + "0";
        try
        {
            String str = JAVA_VERSION_TRIMMED.substring(0, 1);
            str = str + JAVA_VERSION_TRIMMED.substring(2, 3);
            if(JAVA_VERSION_TRIMMED.length() >= 5)
                str = str + JAVA_VERSION_TRIMMED.substring(4, 5);
            else
                str = str + "0";

            return Integer.parseInt(str);
        }
        catch(Exception ex)
        {
            return 0;
        }
    }
    . . .
}
```

Appendix C: Modified Source for com.bmc.thirdparty.org.apache.commons.logging.LogFactory

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Source File Name:   LogFactory.java

package com.bmc.thirdparty.org.apache.commons.logging;

import java.io.*;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.net.URL;
import java.security.AccessController;
import java.security.PrivilegedAction;
import java.util.*;

// Referenced classes of package com.bmc.thirdparty.org.apache.commons.logging:
//      LogConfigurationException, Log

public abstract class LogFactory
{
    . . .
    private static Properties getProperties(final URL url)
    {
        // BMC-for-Android fix
        //      PrivilegedAction action = new Object() {
        //
        //      PrivilegedAction action = new PrivilegedAction

        public Object run()
        {
            try
            {
                InputStream stream = url.openStream();
                if(stream != null)
                {
                    Properties props = new Properties();
                    props.load(stream);
                    stream.close();
                    return props;
                }
            }
            catch(IOException _ex)
            {
                try
                {
                    if(LogFactory.isDiagnosticsEnabled())
                        LogFactory.logDiagnostic("Unable to read URL " + url);
                }
                catch(Exception anyE)
                {
                    throw new RuntimeException(
                        "Unable to resolve property (and cannot log): " + url);
                }
            }
            return null;
        }
    }
    ;
    return (Properties)AccessController.doPrivileged(action);
}
. . .
private static Enumeration getResources(final ClassLoader loader, final String name)
{
    // BMC-for-Android fix
    //      PrivilegedAction action = new Object() {
    //
    //      PrivilegedAction action = new PrivilegedAction() {
    public Object run()
    {
        try
        {
```

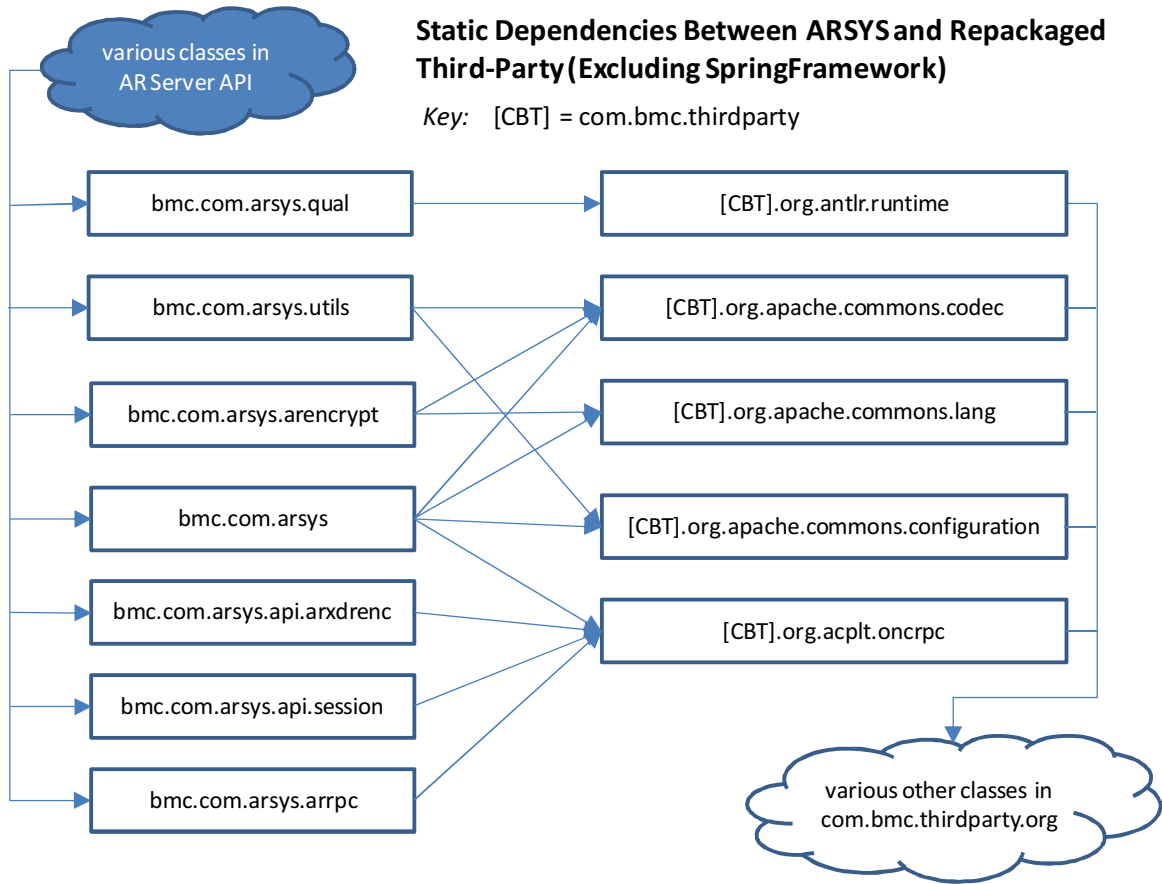
```

        if(loader != null)
            return loader.getResources(name);
        else
            return ClassLoader.getSystemResources(name);
    }
    catch(IOException e)
    {
        // Android-fix: Added try/catch and RuntimeException to report err
        try
        {
            if(LogFactory.isDiagnosticsEnabled())
                LogFactory.logDiagnostic(
                    "Exception while trying to find configuration file " +
                    name + ":" + e.getMessage());
        }
        catch(Exception anyE)
        {
            throw new RuntimeException(
                "Unable to resolve in classloader (and cannot log): " + name);
        }
    }
    catch(NoSuchMethodError _ex)
    {
        return null;
    }
    return null;
}

;
Object result = AccessController.doPrivileged(action);
return (Enumeration)result;
}
}

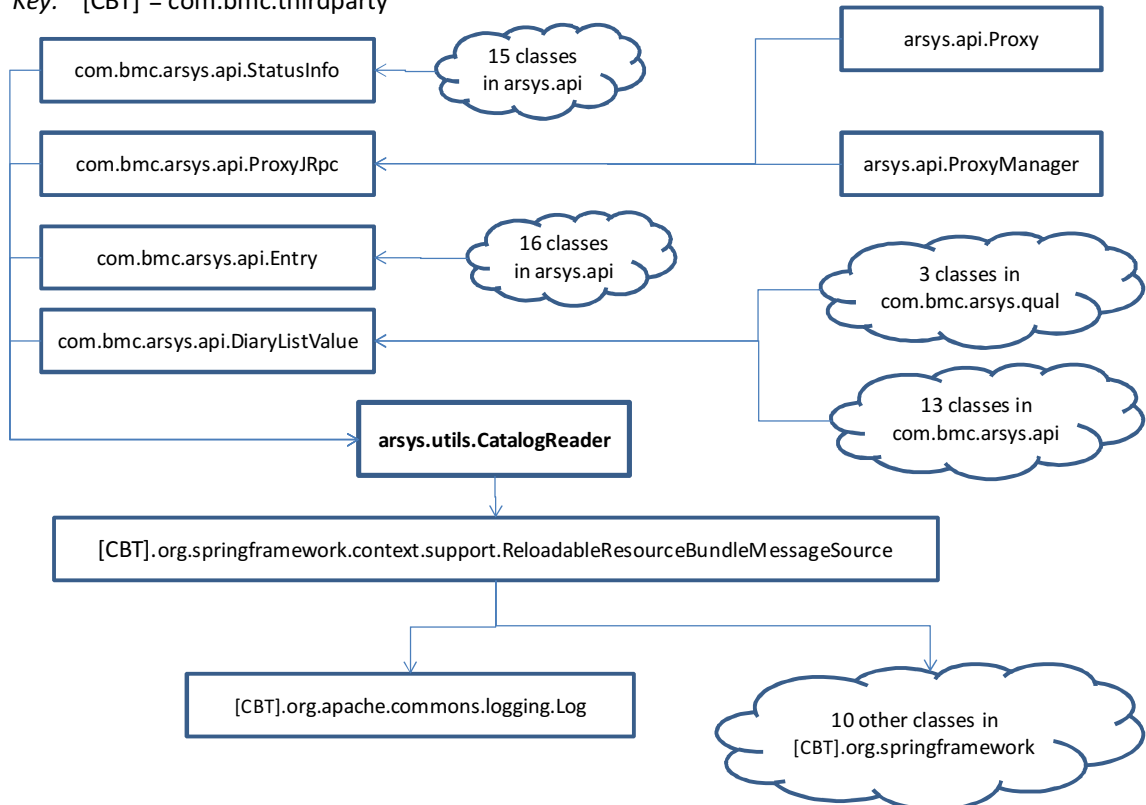
```

Appendix D: Static Dependencies Among Packages in arapi75.jar File



Static Dependencies Between AR Server API and Repackaged SpringFramework

Key: [CBT] = com.bmc.thirdparty



Appendix E: DependencyFinder version 1.2.1 Static Analysis of arapi75.jar

Note that this listing is not complete, but it includes the majority of the classes that depend on non-supported (in Android) Java core classes. Approximately 50 other classes are not shown in the table (but they share a subset of the same non-supported Java core class dependencies).

In this BMC "thirdparty" Repackaged Classes Namespace...	
This BMC Repackaged Class Name...	Depends on this Java Core Class (not supported in Android)
com.bmc.thirdparty.org.apache.commons.beanutils	
BeanMap	java.beans.BeanInfo java.beans.IntrospectionException java.beans.Introspector java.beans.PropertyDescriptor
MappedPropertyDescriptor	java.beans.IntrospectionException java.beans.PropertyDescriptor
PropertyUtils	java.beans.PropertyDescriptor
PropertyUtilsBean	java.beans.BeanInfo java.beans.IndexedPropertyDescriptor java.beans.IntrospectionException java.beans.Introspector java.beans.PropertyDescriptor
BeanUtilsBean	java.beans.IndexedPropertyDescriptor java.beans.PropertyDescriptor
com.bmc.thirdparty.org.apache.commons.configuration	
JNDIConfiguration	javax.naming.Context javax.naming.InitialContext javax.naming.NameClassPair javax.naming.NameNotFoundException javax.naming.NamingEnumeration * javax.naming.NamingException
XMLConfiguration	javax.xml.transform.Result javax.xml.transform.Source javax.xml.transform.Transformer javax.xml.transform.TransformerException javax.xml.transform.TransformerFactory javax.xml.transform.TransformerFactoryConfigurationError javax.xml.transform.dom.DOMSource javax.xml.transform.stream.StreamResult
com.bmc.thirdparty.org.apache.commons.configuration.web	
AppletConfiguration	java.applet.Applet
com.bmc.thirdparty.org.apache.commons.digester	
BeanPropertySetterRule	java.beans.PropertyDescriptor
SetNestedPropertiesRule	java.beans.PropertyDescriptor
SetPropertyRule	java.beans.PropertyDescriptor
com.bmc.thirdparty.org.apache.commons.logging.impl	
ServletContextCleaner	javax.servlet.ServletContextEvent javax.servlet.ServletContextListener
com.bmc.thirdparty.org.springframework.beans	
BeanUtils	java.beans.BeanInfo java.beans.PropertyDescriptor
BeanWrapper	java.beans.PropertyDescriptor
BeanWrapperImpl	java.beans.BeanInfo java.beans.PropertyChangeEvent java.beans.PropertyDescriptor
CachedIntrospectionResults	java.beans.BeanDescriptor java.beans.BeanInfo java.beans.IntrospectionException java.beans.Introspector java.beans.PropertyDescriptor
DirectFieldAccessor	java.beans.PropertyChangeEvent

MethodInvocationException	java.beans.PropertyChangeEvent
PropertyAccessException	java.beans.PropertyChangeEvent
PropertyEditorRegistry	java.beans.PropertyEditor
PropertyEditorRegistrySupport	java.beans.PropertyEditor
PropertyMatches	java.beans.PropertyDescriptor
PropertyValuesEditor	java.beans.PropertyDescriptor
TypeConverterDelegate	java.beans.PropertyDescriptor java.beans.PropertyEditor java.beans.PropertyEditorManager
TypeMismatchException	java.beans.PropertyChangeEvent
com.bmc.thirdparty.org.springframework.beans.factory.annotation	
RequiredAnnotationBeanPostProcessor	java.beans.PropertyDescriptor
com.bmc.thirdparty.org.springframework.beans.factory.config	
ConfigurableBeanFactory	java.beans.PropertyEditor
CustomEditorConfigurer	java.beans.PropertyEditor
InstantiationAwareBeanPostProcessor	java.beans.PropertyDescriptor
InstantiationAwareBeanPostProcessorAdapter	java.beans.PropertyDescriptor
com.bmc.thirdparty.org.springframework.beans.factory.support	
AbstractAutowireCapableBeanFactory	java.beans.PropertyDescriptor
AbstractBeanFactory	java.beans.PropertyEditor
AutowireUtils	java.beans.PropertyDescriptor
com.bmc.thirdparty.org.springframework.beans.propertyeditors	
ByteArrayPropertyEditor	java.beans.PropertyEditorSupport
CharArrayPropertyEditor	java.beans.PropertyEditorSupport
CharacterEditor	java.beans.PropertyEditorSupport
ClassArrayEditor	java.beans.PropertyEditorSupport
ClassEditor	java.beans.PropertyEditorSupport
CustomBooleanEditor	java.beans.PropertyEditorSupport
CustomCollectionEditor	java.beans.PropertyEditorSupport
CustomDateEditor	java.beans.PropertyEditorSupport
CustomNumberEditor	java.beans.PropertyEditorSupport
FileEditor	java.beans.PropertyEditorSupport
InputStreamEditor	java.beans.PropertyEditorSupport
LocaleEditor	java.beans.PropertyEditorSupport
PropertiesEditor	java.beans.PropertyEditorSupport
ResourceBundleEditor	java.beans.PropertyEditorSupport
StringArrayPropertyEditor	java.beans.PropertyEditorSupport
StringTrimmerEditor	java.beans.PropertyEditorSupport
URLEditor	java.beans.PropertyEditorSupport
com.bmc.thirdparty.org.springframework.beans.support	
ArgumentConvertingMethodInvoker	java.beans.PropertyEditor
ResourceEditorRegistrar	java.beans.PropertyEditor
com.bmc.thirdparty.org.springframework.ejb.access	
AbstractRemoteSlsbInvokerInterceptor	java.rmi.RemoteException
SimpleRemoteSlsbInvokerInterceptor	java.rmi.Remote java.rmi.RemoteException
com.bmc.thirdparty.org.springframework.remoting.jaxrpc	
JaxRpcPortClientInterceptor	java.rmi.Remote java.rmi.RemoteException
JndiRmiClientInterceptor	java.rmi.Remote java.rmi.RemoteException
JndiRmiServiceExporter	java.rmi.Remote java.rmi.RemoteException
RmiBasedExporter	java.rmi.Remote
RmiClientInterceptor	java.rmi.Naming java.rmi.NotBoundException java.rmi.Remote java.rmi.RemoteException java.rmi.registry.LocateRegistry java.rmi.registry.Registry java.rmi.server.RMIClientSocketFactory
RmiClientInterceptorUtils	java.rmi.ConnectException

	java.rmi.ConnectIOException java.rmi.MarshalException java.rmi.NoSuchObjectException java.rmi.Remote java.rmi.RemoteException java.rmi.StubNotFoundException java.rmi.UnknownHostException java.rmi.UnmarshalException
RmiInvocationHandler	java.rmi.Remote java.rmi.RemoteException
RmiInvocationWrapper	java.rmi.RemoteException
RmiInvocationWrapper_Stub	java.rmi.Remote java.rmi.RemoteException java.rmi.UnexpectedException java.rmi.server.RemoteObject java.rmi.server.RemoteRef java.rmi.server.RemoteStub
RmiRegistryFactoryBean	java.rmi.Remote java.rmi.RemoteException java.rmi.registry.LocateRegistry java.rmi.registry.Registry java.rmi.server.RMIClientSocketFactory java.rmi.server.RMIServerSocketFactory java.rmi.server.UnicastRemoteObject
RmiServiceExporter	java.rmi.NoSuchObjectException java.rmi.NotBoundException java.rmi.Remote java.rmi.RemoteException java.rmi.registry.LocateRegistry java.rmi.registry.Registry java.rmi.server.RMIClientSocketFactory java.rmi.server.RMIServerSocketFactory java.rmi.server.UnicastRemoteObject
_RmiInvocationHandler_Stub	java.rmi.RemoteException java.rmi.UnexpectedException
_RmiInvocationWrapper_Tie	java.rmi.Remote
com.bmc.thirdparty.org.springframework.web.servlet.tags	
TransformTag	java.beans.PropertyEditor
com.bmc.thirdparty.org.springframework.web.servlet.tags.form	
AbstractDataBoundFormElementTag	java.beans.PropertyEditor
AbstractFormTag	java.beans.PropertyEditor
HiddenInputTag	java.beans.PropertyEditor
InputTag	java.beans.PropertyEditor
OptionTag	java.beans.PropertyEditor
OptionWriter	java.beans.PropertyEditor
RadioButtonTag	java.beans.PropertyEditor
SelectTag	java.beans.PropertyEditor
SelectedValueComparator	java.beans.PropertyEditor
TextareaTag	java.beans.PropertyEditor
ValueFormatter	java.beans.PropertyEditor
com.bmc.thirdparty.org.springframework.web.servlet.view.xslt	
IntrospectorCleanupListener	java.beans.Introspector
com.bmc.thirdparty.org.springframework.jca.cci.connection	
ConnectionSpecConnectionFactoryAdapter	javax.resource.cci.ConnectionFactory
DelegatingConnectionFactory	javax.resource.cci.ConnectionFactory
com.bmc.thirdparty.org.springframework.jca.cci.core	
CciTemplate	javax.resource.cci.Interaction
com.bmc.thirdparty.org.springframework.jca.support	
LocalConnectionFactoryBean	javax.resource.spi.ManagedConnectionFactory
com.bmc.thirdparty.org.springframework.jdbc.support.lob	
LobCreatorUtils	javax.transaction.Transaction
com.bmc.thirdparty.org.springframework.jms	
[15 classes]	javax.jms
com.bmc.thirdparty.org.springframework.jmx	
[10 classes]	javax.management

com.bmc.thirdparty.org.springframework.mail.javamail	
JavaMailSenderImpl	javax.mail.Session javax.mail.Transport javax.mail.internet.MimeMessage
MimeMessageHelper	javax.activation.DataHandler javax.activation.FileDataSource javax.mail.internet.MimeBodyPart javax.mail.internet.MimeMessage javax.mail.internet.MimeMultipart
SmartMimeMessage	javax.mail.internet.MimeMessage
com.bmc.thirdparty.org.springframework.orm.hibernate3	
SessionFactoryUtils	javax.transaction.Transaction
com.bmc.thirdparty.org.springframework.orm.hibernate3.support	
OpenSessionInViewFilter	javax.servlet.FilterChain
com.bmc.thirdparty.org.springframework.orm.jpa	
LocalContainerEntityManagerFactoryBean	javax.persistence.spi.PersistenceProvider
com.bmc.thirdparty.org.springframework.remoting.jaxrpc	
JaxRpcPortClientInterceptor	javax.xml.rpc.Service
com.bmc.thirdparty.org.springframework.remoting.rmi	
_RmiInvocationHandler_Stub	javax.rmi.CORBA.Util
com.bmc.thirdparty.org.springframework.web.jsf	
DecoratingNavigationHandler	javax.faces.application.NavigationHandler
[various other package spaces	
[approximately 30 other classes]	[not specified]