Android cross-platform support study

Name: Xinwen Li

About this study

Android has been most popular developing platform provide usuful services and fearures in recent years. Android apps run within the Dalvik Virtual Machine which is similar to Java VM optimized for devices with limited memory and processing capacity. Android app with Java source code are compiled into the Java byte code using the Java compiler. However, this approch introduces difficulties for developers farmiliar with other languages rather than Java. Then people dive into a solution build runtime library support developing language other than Java. This solution allow developers develop application use same source code for different platforms. For example, Xamarin allow developers develop application for Android, IOS, and OS X using C# and the .NET framwork.

While it brings benificial allow developers use same source code for different platforms, it also introduces difficulties reverse engineering the application with third-party language support runtime libraries. This study focus on third-party runtime libraries on Andoird platform

Pros and cons of cross-platform support

Pros:
•Allow developers leverage exsiting language skills.
•Code re-use.

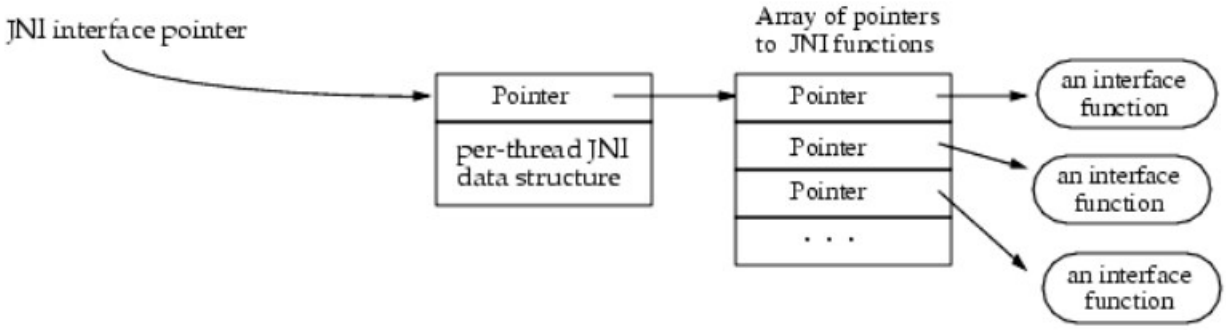
Cons:
•Third-party architecture lincensing requirement.
•Update delay.
•Performance and bugs.
•Understnad memory manage and API calling.
•Extra runtime libraries stored.

How does cross-platform work?

The trick is utilize Java Native Interface (JNI). JNI is a framwork that allows non-Java code to be called by a Java code running in side a JVM.

JNI Interface Functions and Pointers

Native code access Java VM features by calling JNI functions. Functions are pointed by an array of pointers pointed by JNI interface pointer. See figure below:



Native methods receive JNI interface pointer as an argument. For the same Java thread, multiple native method calls are guranteed to get the same interface pointer. For different different Java threads, it's possible to get different JNI interface pointers.

Compiling, Loading and linking Native Methods

Since the Java VM is multithreaded, native libraries should also be compiled and linked with multithread aware native compilers. e.g -D_REENTRANT flag for gcc compiler.

Native methods are loaded with System.loadLibrary method. The system will follows a standard convert library name to a native dynamic linking library name. For example, Solaris system converts libXXX to libXXX.so and Win32 system converts libXXX to LibXXX.dll.

If operating system does not support dynamic linking, the VM completes the System.loadLibrary call to prelink the native methods with VM.

Native Method Arguments

JNI interpace pointer is the first argument of type JNIEnv. The second argument depends on method type. If native method is static. the second argument refers to its Java Class. If the method is nonstatic, the second argument refers to the object. The possible remaining arguments depends on the native method arguments.

Referencing Java Objects

Primitive types such as int, char are copied between Java and native code. Arbitrary Java objects are passed by reference. The VM keep track of all objects that have been passed to native code. The native code must inform VM the objects is no longer needed.

Accessing Java Objects, Primitive Arrays, Fields and Methods

- Java Objects
  Java VM creats a registry for each transition of control from Java to a native method to implement reference. JNI provides a set of accessor functions on references.

- Primitive Arrarys
  JNI provide a set tof functions copy primitive array elements between a segment of a Java array and a native memory buffer.

- Fields and Methods
  JNI identify methods and fields by their symbolic names and type signatures. For example: to call method f in class cls, the native code dirst obtain a method ID:
  jmethodID method_id = env->GetMethodID(cls, “f”, “(ILjava/lang/String;)D”);
  The native code can then use the method ID repeatedly:
  jdouble result = env->CallDoubleMethod(obj, method_id, 10, str);

For more JNI information

Please refer: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

What is inside an APK file?

An Apk file are a zip file containing resources and assembled java code. If we use apktool decode APK file, we usually get following part:

- AndroidManifest.xml
  Every app project must have one at the root of the project source set describing important information of the app. The information includes: app package name, components of the app, the permission the app needs, features app requires.

- smali
  Contains Kotlin/Java code that disassembles from APK’s DEX files. Each .smali file corresponds to a Kotlin/Java class.

- res
  Resources are the additional files and static content the app code uses, such as layout, bitmaps.

- lib
  If app includes native code, this directory contains APK’s native libraries (.so files).

How to identify native library exsist and in use?

- Native libraries exsist
  If we decode/unzip APK file, we can find .so file.

- Native libraries in use
  According to JNI working flow, Native methods are loaded with System.loadLibrary method. So if native libraries is in use, System.loadLibrary method must be called either from code or JVM.

Native library identify implementation

Please see native_lib_identify.py