Java Pathfinder on Android Devices

Alexander Kohan[1], Mitsuharu Yamamoto[1], Cyrille Artho[2,3], Yoriyuki Yamagata[2], Lei Ma[1], Masami Hagiya[4], Yoshinori Tanabe[5]

[1]Chiba University, Japan [2]AIST/ITRI, Japan [3]KTH Royal Institute of Technology, Sweden
[4]The University of Tokyo, Japan [5]Tsurumi University, Japan

DOI: 10.1145/3011286.3011292
http://doi.acm.org/10.1145/3011286.3011292

ABSTRACT

Because Android apps are written in Java and executed on a virtual machine (VM), there is an opportunity to employ Java Pathfinder (JPF) for their verification. There already exist two JPF extensions, jpf-android and jpf-pathdroid. The former executes Java bytecode on the Java VM, while the latter executes Android applications in their original format. Both do not support native methods, and thus depend on a model of the Android environment. This paper introduces an alternative approach: we run JPF as an Android application that executes Java bytecode, which gives us direct access to the Android environment. This approach allows us to verify rich Android apps that rely on native calls.

Keywords

Android; Java Pathfinder; software model checking

1. INTRODUCTION

Android apps are written in Java and executed on a virtual machine (called 'Dalvik' prior to Android 5.0 and 'Android Runtime' (ART) afterwards [3]). As Android apps also use the standard Java libraries, this makes them attractive for verification with Java Pathfinder (JPF). Two existing JPF extensions are able to execute Android applications, but they rely on model classes to replace native methods: jpf-android [13] executes Java bytecode from the early compilation stage of an Android project, while jpf-pathdroid [9] supports DEX code but not native Android methods.

The key problem is that existing extensions run on a desktop computer, where Android libraries cannot be executed. It is therefore not possible to delegate native method calls to an Android run-time environment. Because of this, verification of Android apps is limited to cases where all necessary native methods are covered by model classes.

This paper introduces an alternative approach to perform verification directly on Android devices. We implement a service, called jpf-mobile [7], that runs an adapted version of JPF on Android. This allows us to interface directly with the Android run-time environment, paving the way towards executing rich Android applications in JPF.

The rest of the paper is structured as follows: Section II provides an overview of JPF and related works on verifying Android apps; Section III explains implementation details of the jpf-mobile project; its evaluation is presented in Section IV; Section V summarizes the obtained results and explains directions for future work.

DOI: 10.1145/1235

2. BACKGROUND

2.1 Apps on the Android platform

Android is a wide-spread operating system for smartphones and tablets. It has a Linux kernel at its core, its system libraries are written in C and C++, while user applications, often called *apps*, are developed in Java. Android apps comprise components of two main types: *activities* and *services*. Activities are associated with GUI (windows), and services are usually long-running background tasks.

The apps have several unique traits that distinguishes them from Java programs on the desktop. First, their execution is mostly event-driven; an app must register callbacks to perform desired actions when system or user events occur. Second, there is no `main()` function, and an app may have multiple entry points. Third, there is only one app that can be executed at the moment; if a user switches to another app, the current app is paused or destroyed (this restriction is lifted in Android N, where multiple apps are supported simultaneously [2]). Fourth, apps can communicate with the system and with each other only by asynchronous messages, called Intents.

The differences between Android and desktop apps make application of desktop tools for analysis of apps non-trivial.

2.2 Java Pathfinder and its extensions

The JPF is a framework to find and explain defects in Java applications. Its core is a virtual machine for Java bytecode that runs on the top of the system JVM. The virtual machine is extended with capabilities to detect execution choices of a system under test (SUT), so that the JPF can perform model checking of a SUT by analyzing all its execution paths. The JPF is not limited to model checking; its core functionality is extended with modules that handle symbolic execution, network communication, specification generation, UML chart modeling, and more.

The jpf-android project [13] is a JPF extension (module) to verify Android apps on the desktop. The extension partially models Android environment, including the message

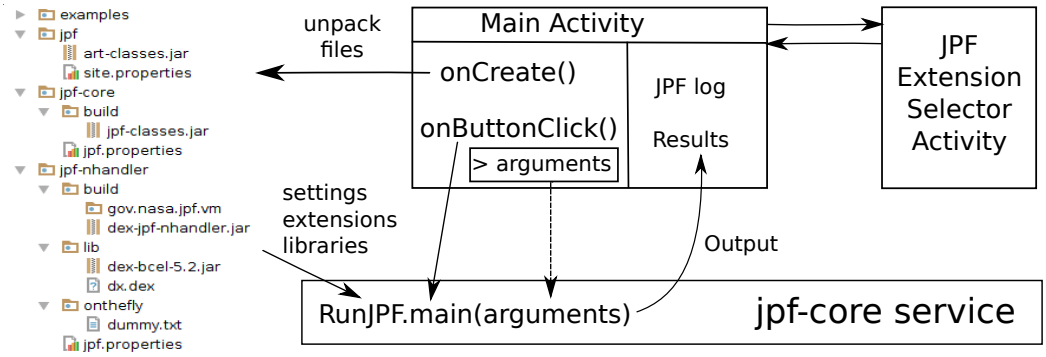

Figure 1: The Architecture of jpf-mobile

queue and Intent mechanism for interprocess communication. It allows verifying GUI-based Android apps according to a user-supplied script that defines sequences of system and user events. The extension is being actively developed; its authors are currently researching the ways to model Android environment more accurately using information collected during app's execution [12].

Another JPF extension, jpf-pathdroid [9], targets Android apps for which the source code is not available. Jpf-pathdroid implements Android Dalvik VM instruction set to perform verification based on bytecode analysis of apps distributed in binary `dex` and `apk` files. The extension allows discovering generic defects, such as race conditions, in Android apps.

3. IMPLEMENTATION

We first show the architecture of the Android application containing jpf-core, followed by a description of how the system under test is loaded into that application. Finally, we highlight some key technical challenges we faced and how we overcame them.

3.1 Architecture

The jpf-mobile project is designed as an Android application with a graphical user interface (GUI) that incorporates the entire jpf-core as a background service. JPF runs in a separate thread so the GUI remains responsive while JPF is executing.

The overview of jpf-mobile's architecture is shown on Figure 1; its GUI is demonstrated on Figure 3. The GUI contains Main Activity that provides an input field to pass arguments for underlying jpf-core (the `main` method in the `RunJPF` class) when a user clicks the button to start the verification process. The Main Activity also contains a frame to display the output of jpf-core.

The project aims to make as few changes as possible to the source of incorporated jpf-core to make easier the transition to its newer versions. Therefore, we preserved the original initialization phase that relies on `site.properties` and `jpf.properties`. The configuration files and dependent libraries are bundled in the app and are copied onto device's hard disk upon the first launch (the arrow from `onCreate` on Figure 1). These files are placed in jpf-mobile's public directory, so they can be accessed and customized by a user.

The JPF extension loading mechanism is also preserved. Extensions (currently, only `jpf-nhandler`) are loaded in the usual way, according to a list in the file `site.properties`. Because editing of textual files on a mobile device is less convenient than on a desktop, jpf-mobile includes Extension Selector Activity that provides a list with checkboxes to enable/disable available extensions.

3.2 Systems under test

As desktop version of jpf-core is mainly designed for verification of command-line Java programs, and support for GUI applications is added later by extensions such as jpf-awt, we consider the same principle for jpf-mobile project. The project is hence mainly aimed to provide model checking capabilities for non-GUI elements of Android apps, such as background services.

Modern Android apps are usually developed with Android Studio IDE that utilizes the Gradle build automation tool. The build process comprises compilation of `java` files into `class` files, conversion of `class` files into Dalvik executable (DEX) format, and creation of Android Package (APK) archive from `dex` files and app's resources.

To use such app as a SUT for jpf-mobile, the class files generated during an intermediate stage must be copied onto a user-accessible directory on an Android device. A regular `.jpf` file must accompany the SUT to specify the target class, classpath, listeners and other properties.

In addition to event-driven GUI apps, Android supports execution of compatible command-line Java programs with `dalvikvm` command. Example test cases included in the jpf-core distribution (BoundedBuffer, Racer) fall in this category. Command-line programs can be used as SUT for jpf-mobile the same way as regular Android apps.

3.3 Technical challenges

Challenges in the implementation arose from differences between Java runtime environments and resource management on the desktop and on mobile devices.

3.3.1 Java version and the standard classes

The current version of jpf-core depends on Java 8, while the maximum supported version on Android is Java 7. To achieve compatibility, we replaced all Java 8 specific constructions in jpf-core source code. The changes are mainly related to default methods in interfaces; a few modifications were also required in native peers. For example, the field `seed` in the class `java.util.Random` has type `long` in Java 7 and type `AtomicLong` in Java 8. The corresponding peer class need to be adjusted, because casting to a wrong type

Table 1: Evalution of jpf-mobile

No	Test Case	Pass	Memory	T_{desk}	T_{emu}	t_{device}	Observed Output
Test cases from jpf-core distribution							
1	HelloWorld	+	57 Mb	0 s	1 s	4 s	No errors are detected
2	BounderBuffer	+	57 Mb	0 s	1 s	4 s	A deadlock is detected
3	oldclassic	+	57 Mb	0 s	1 s	4 s	A deadlock is detected
4	Crossing	+	102 Mb	1 s	12 s	57 s	The solution is shown
5	NumericValueCheck	−	57 Mb	0 s	1 s	3 s	The listener does not report a violation
6	Rand	+	57 Mb	0 s	1 s	4 s	Uncaught arithmetic exception is found
7	StopWatch	+	57 Mb	0 s	1 s	3 s	Time constraint violation is reported
8	Coverage	+	57 Mb	0 s	1 s	3 s	Coverage statistics are displayed
9	RobotManager	+	166 Mb	3 s	2 m 56 s	−	Uncaught NPE is found
10	Racer	+	57 Mb	0 s	1 s	5 s	A race condition is detected
11	DiningPhil (n = 4)	+	106 Mb	1 s	30 s	2 m 37 s	A race condition is detected
	DiningPhil (n = 5)	+	408 Mb	5 s	4 m 21 s	−	A race condition is detected
	DiningPhil (n = 6)	−	853 Mb	41 s	(31 m 31 s)	−	Abortion due to an out of memory error
Test cases for jpf-nhandler							
12	Hostname	+	72 Mb	1 s	2 s	6 s	Hostname is printed
13	HeapMemory	+	11 Mb	−	1 s	4 s	Size of the used native heap is printed
14	SystemClock	+	15 Mb	−	1 s	4 s	Elapsed time from device boot is printed
15	AndroidOpenFile	+	35 Mb	−	1 s	−	Existing file is opened in read-only mode

using Unsafe methods results in a segmentation fault. Unsafe methods access a value directly in memory, and the use of a different data type results in accessing unallocated memory in this case.

The other problem was that some classes from the standard Java library are missing on Android. In particular, the only class available in `sun.misc` package is `sun.misc.Unsafe`, which is supposed to be accessed only by Android internal apps. Because jpf-core extensively uses the `Unsafe` class, we achieve the access to it with Java reflection API. The other classes from this package, such as `SharedSecrets`, seem to be used only for performance improvements, so their use is replaced with analogous constructions.

3.3.2 System libraries

During initialization phase, jpf-core requires access to all Java classes that are used in SUT, including classes from the standard Java library. The desktop version of jpf-core reads the `sun.boot.class.path` system variable to get the JRE location, and then loads standard classes from `rt.jar` included in JRE distribution. However, this variable is not set on Android, there is no `rt.jar` file, and system libraries are distributed across several files in `/system/framework/` directory.

The Android libraries are usually stored in DEX or optimized DEX (ODEX) formats. However, on devices with Android 5.0 and greater, the libraries can also be stored in the new OAT format [10], which is based on Linux ELF format. It contains native code generated during ahead-of-time (AOT) compilation of original Java bytecode, but also incorporates original DEX files as is.

We currently supply system libraries as a jar archive that contains compiled classes from the latest (6.0.1) version of Android source code. This temporary measure allows us to execute jpf-mobile on older devices or emulators as well due to the backward compatibility of the libraries.

3.3.3 Class loaders and class generation

To handle native calls in Android apps, we include in the project the jpf-nhandler extension that delegates the calls to the underlying Java virtual machine [11]. The extension uses Apache BCEL library [4] to generate and compile peer classes, which are then loaded by jpf-core.

Classes generated by jpf-nhandler are loaded using the `URLClassLoader`. However, this class loader is not implemented on Android, and loading from `class` and `jar` files is not possible. Hence, the only supported format for external code that can be loaded by an Android app is `dex`.

To overcome this problem, we modified jpf-nhandler to use the same class loader that is being used for jpf-core. Moreover, we incorporated an invocation of `dx` utility [1] to convert `class` files into `dex` format on the fly.

3.3.4 Heap size limit

Android imposes a heap size limit per Android app, which usually varies between 24 and 128 Mb and is quite small for the needs of JPF. However, this restriction can be lifted by setting `largeHeap` property in the jpf-mobile app's manifest. In this case, once the soft limit is reached, the app's heap size starts to slowly increase until it hits the hard limit. The hard heap size limit is set for a device by a manufacturer; it cannot be modified unless the device is rooted.

However, the hard limit can be adjusted for a standard QEMU emulator by setting the corresponding system property. This allows us to allocate at most of 740 Mb of heap, which is enough for small and medium-sized apps (setting a larger size results in errors due to a possible bug in QEMU [5]).

4. EVALUATION

Performance of the current implementation of jpf-mobile is tested on the examples included in the distribution of jpf-core. Each example is executed on a desktop machine using standard version of jpf-core to record execution time and memory in use. The example is then executed on an emulator and on a smartphone using jpf-mobile. The obtained performance data is shown in Table 1. The successful detection of a deadlock in one of the examples, BoundedBuffer,

within an emulator, is shown on Figure 3. All tests were executed on a desktop machine with the following specifications: Intel Core i5-3210M 2.50 GHz, 4 Gb RAM, Gentoo Linux; and on a mobile device with Krait Quad-Core 1.5 GHz, 2 Gb RAM, Android 4.4.2.

The experiments show that the performance of jpf-mobile is satisfactory for cases with low memory requirements. The performance is degraded when memory required by JPF exceeds the soft heap size limit. In this case, the app cannot allocate all the memory it needs at once, but slowly proceeds with increasing the heap size by the predefined value. If the memory required is below the hard limit, jpf-mobile finishes quickly once the heap is resized. However, when the hard limit is reached, the execution aborts due to an out of memory error, which we observe for the "Dining Philosophers" example with number of philosophers set to six.

The obtained data also shows that jpf-mobile is executed few times faster on an emulator than on a real device. Some examples with large memory requirements (i.e. RobotManager and DiningPhil with $n = 5$ and $n = 6$) can be executed only on an emulator, because the hard heap size limit set on the device cannot be easily increased.

The integration of jpf-nhandler is tested on several examples that make use of native calls to obtain data on the environment. For the example 12 listed in Table 1, we try to obtain the host name of the device. During the execution, jpf-nhandler successfully generates source files for classes `libcore.io.Posix` and `android.system.OsConstants`, produces `.class` files, and converts them into `dex` format. The JPF loads the generated classes, prints the host name, `localhost`, and successfully finishes the verification.

In the example 13, we obtain the size of natively allocated heap by calling the `Debug.getNativeHeapAllocatedSize()` method, which forces jpf-nhandler to generate the peer class `OTF_JPF_android_os_Debug`. Unlike the host name example, we use Android-specific API, so that the example cannot be executed on the desktop version of jpf-core. Finally, we invoke an Android-specific native methods to print time elapsed since the device is booted (example 14) and to open a file in the read-only mode using low-level method `open` from `android.system.Os` class (example 15, Figure 2).

```
import java.io.*;
import android.system.*;

public class AndroidOpenFile {
  public static void main(String[] args) {
    String path = "/sdcard/Android/data/" +
      "jp.ac.chiba_u.s.math.jpf/files/" +
      "testing/test.txt";
    try {
      FileDescriptor fd = Os.open(path,
        OsConstants.O_RDONLY, 0);
      System.out.println(fd.valid());
      Os.close(fd);
    } catch (ErrnoException e) {
      e.printStackTrace();
    }
  }
}
```

Figure 2: Source code for Example 15

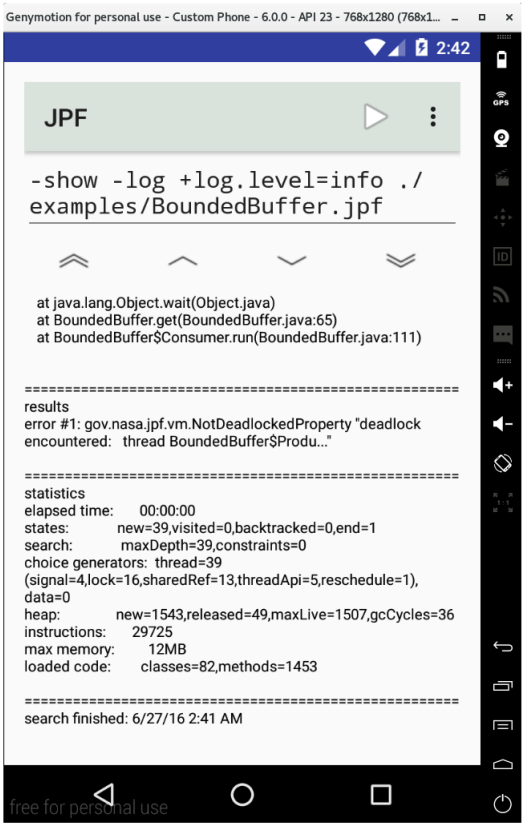

Figure 3: The Interface of jpf-mobile App

5. FUTURE WORK

The next step of the project is the verification of small and medium-sized open-sourced Android apps with background services. A straightforward way to apply the tool to a real Android app is by injection of the `main()` method that sets up services to be verified and their clients. However, it may be difficult to properly instantiate a service in `main()`, because app's components are typically launched by Android OS itself. For this reason, we plan to use a JUnit 4-compatible test runner from Android Testing Support Library [8], which enables component instantiation inside JUnit tests that are executed on an Android device. As the JPF already supports JUnit [6], it can be used to execute the code inside each test, verifying the behavior of a service.

Besides the additions necessary to proceed to the next stage, we aim to improve the core functionality of the tool by implementing a mechanism to load Android system libraries from DEX, ODEX, and OAT formats and providing workarounds for memory allocation restrictions. We also plan to improve UI of the tool to simplify SUT selection and modification of `.jpf` configuration files.

6. CONCLUSION

In this work we proposed an alternative approach to apply JPF for verification of Android apps directly on the Android platform. The main benefit of this approach is that JPF can now interact with the Android framework, so that delegation of native methods calls to the underlying JVM becomes

possible, which is an important step towards verification of sophisticated Android apps.

At the current stage of the project, we are able to execute jpf-core on an Android device or an emulator and confirm detection of deadlocks, races, and other issues on the examples included in jpf-core distribution. We also observe the correct behavior of jpf-nhandler extension, as the peer classes are generated, and execution of native methods is delegated to a Dalvik (ART) virtual machine. The integration of jpf-nhandler lays the foundation for the future work of applying JPF to full-fledged Android apps, which can be accomplished by `main()` injection and the usage of Android test runners.

7. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number JP26280019. The authors would like to thank Google Inc. for providing support to this project via Google Summer of Code 2016. The authors are also grateful to NASA JPF team for mentoring this project and clarifying various aspects of jpf-core.

8. REFERENCES

[1] Android developing – other tools. http://wing-linux.sourceforge.net/guide/developing/tools/othertools.html. [Online; accessed 3-August-2016].

[2] Android N for developers – multi-window support. https://developer.android.com/preview/features/multi-window.html. [Online; accessed 3-August-2016].

[3] ART and Dalvik. https://source.android.com/devices/tech/dalvik/. [Online; accessed 28-June-2016].

[4] The byte code engineering library API. https://commons.apache.org/proper/commons-bcel/manual/bcel-api.html. [Online; accessed 3-August-2016].

[5] Issue 214093: Launching emulator with a very high ram value crashes the emulator. https://code.google.com/p/android/issues/detail?id=214093. [Online; accessed 1-September-2016].

[6] Java Pathfinder wiki – writing JPF tests. http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/jpf_tests. [Online; accessed 1-September-2016].

[7] Jpf-mobile project on BitBucket. https://bitbucket.org/matsurago/jpf-mobile-devices. [Online; accessed 4-August-2016].

[8] Testing support library. https://developer.android.com/topic/libraries/testing-support-library/index.html. [Online; accessed 1-September-2016].

[9] Peter Mehlitz. Jpf-pathdroid – readme. http://babelfish.arc.nasa.gov/hg/jpf/jpf-pathdroid/file/85aa01d0112c/README. [Online; accessed 3-August-2016].

[10] P. Sabanal. Hiding behind ART. https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf. [Online; accessed 28-June-2016].

[11] N. Shafiei and F. v. Breugel. Automatic handling of native methods in Java PathFinder. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 97–100, New York, NY, USA, 2014. ACM.

[12] H. van der Merwe, O. Tkachuk, S. Nel, B. van der Merwe, and W. Visser. Environment modeling using runtime values for jpf-android. *SIGSOFT Softw. Eng. Notes*, 40(6):1–5, Nov. 2015.

[13] H. van der Merwe, B. van der Merwe, and W. Visser. Verifying Android applications using Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.