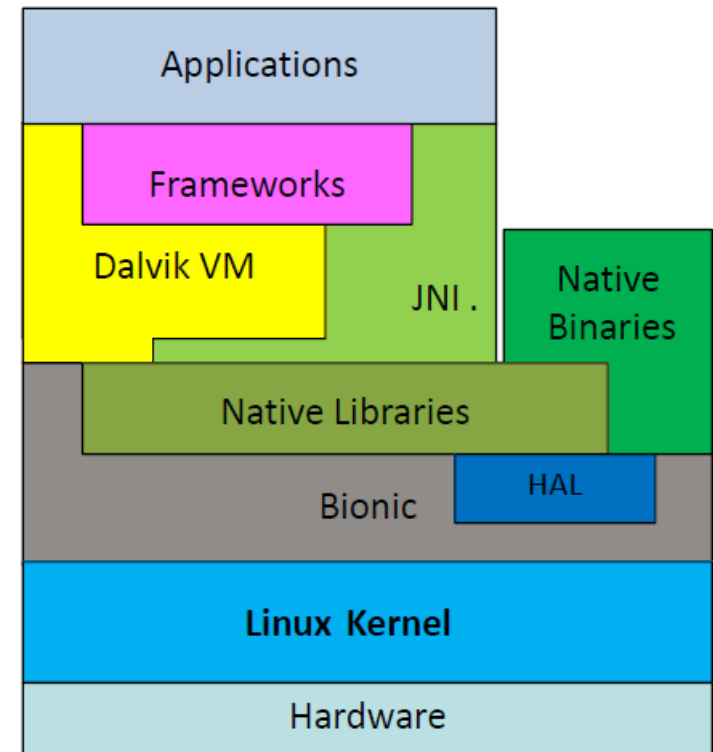


# Android Debugging ART

▪ Khaled JMAL ▪  
2016 / 11 / 17

# The Dalvik Virtual Machine

- Up to version 4.4 „KitKat“, Android was based on the Dalvik Virtual Machine
- Java compiles into DEX code
- DEX code is compiled just-in-time or interpreted by the Dalvik VM interpreter



Source: „Dalvik and ART“, Jonathan Levin

# The TRACE32 Dalvik Awareness

---

- Written by HAP
- Based on the symbols of the Dalvik VM (libdvm.so)
- Features:
  - List VM threads with their class descriptors and names
  - Display the VM stack for a single VM thread
  - List the Dalvik (Java) source code for each method
  - Display the stack frame with Java to native and native to Java transitions
  - Stepping through the DEX code not possible!

# The TRACE32 Dalvik Awareness

- VM threads and thread stack

The screenshot displays the TRACE32 Dalvik Awareness interface, which is used for debugging Android applications. It consists of two main windows: 'B::EXTension.VMList' and 'B::EXT.VMView 0xEFF79040'.

**B::EXTension.VMList** window:

| magic    | pid  | process/task name        | method name | class descriptor    | class source  | lvl |
|----------|------|--------------------------|-------------|---------------------|---------------|-----|
| EFCA1160 | 05ED | system_server            |             |                     |               |     |
| EFD279E0 | 062C | com.android.systemui     |             |                     |               |     |
| EFF79040 | 063B | android.process.media    | next        | Landroid/os/Message | MessageQueue. | 11  |
| EFDB5400 | 063F | GC                       |             |                     |               | 12  |
| EFC9ECC0 | 0640 | Signal.Catcher           |             |                     |               | 11  |
| EFC9E420 | 0641 | JDWP                     |             |                     |               | 14  |
| EFC9E140 | 0642 | Compiler                 |             |                     |               | 12  |
| EFC9E9E0 | 0643 | ReferenceQueueD          | wait        | Ljava/lang/Object;  | Object.java   | 10  |
| EFC9CCA0 | 0644 | FinalizerDaemon          | wait        | Ljava/lang/Object;  | Object.java   | 10  |
| EFC9C400 | 0645 | FinalizerWatchd          | wait        | Ljava/lang/Object;  | Object.java   | 10  |
| EFC9C120 | 0646 | Binder_1                 |             |                     |               | 16  |
| EFCE6660 | 0647 | B::EXT.VMView 0xEFF79040 |             |                     |               |     |
| EFCE1620 | 0649 | tl                       |             |                     |               |     |
| E8F8E0E0 | 0780 | B::EXT.VMView 0xEFF79040 |             |                     |               |     |
| EFCE1900 | 064A | co                       |             |                     |               |     |
| EFE0DA20 | 0656 | co                       |             |                     |               |     |
| EFC50660 | 0669 | co                       |             |                     |               |     |

A blue button labeled 'Display detailed' is overlaid on the 'android.process.media' row.

**B::EXT.VMView 0xEFF79040** window:

| method name  | class descriptor                           | class source        | frmCurPC  | frameptr |
|--------------|--------------------------------------------|---------------------|-----------|----------|
| next         | Landroid/os/MessageQueue;                  | MessageQueue.java   | 58C84D46  | 571C0E24 |
| loop         | Landroid/os/Looper;                        | Looper.java         | 58C8375E  | 571C0E74 |
| main         | Landroid/app/ActivityThread;               | ActivityThread.java | 58ACCF A6 | 571C0EB4 |
| (n/a)        | (n/a)                                      | (n/a)               | 00000000  | 571C0ED8 |
| invokeNative | Ljava/lang/reflect/Method;                 | Method.java         | 00000006  | 571C0EEC |
| invoke       | Ljava/lang/reflect/Method;                 | Method.java         | 584FB00E  | 571C0F20 |
| run          | Lcom/android/internal/os/ZygoteInit\$Metho | ZygoteInit.java     | 58DFDA5A  | 571C0F60 |
| main         | Lcom/android/internal/os/ZygoteInit;       | ZygoteInit.java     | 58DFDDE8  | 571C0F94 |
| (n/a)        | (n/a)                                      | (n/a)               | 00000000  | 571C0FC0 |
| main         | Ldalvik/system/NativeStart;                | NativeStart.java    | 00000000  | 571C0FD4 |
| (n/a)        | (n/a)                                      | (n/a)               | 00000000  | 571C0FEC |

# The TRACE32 Dalvik Awareness

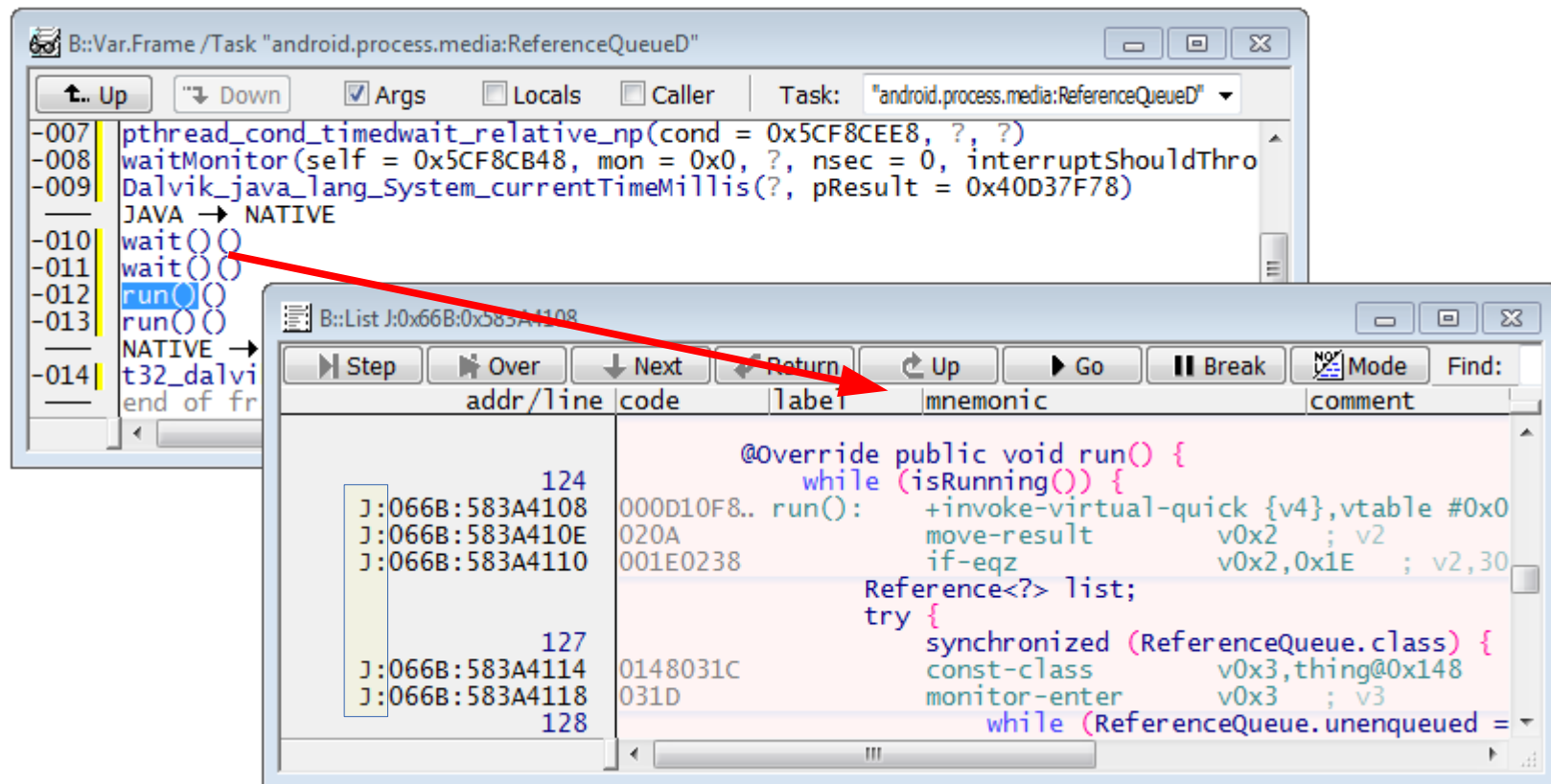
- JAVA to NATIVE / NATIVE to JAVA transitions

```

B::Var.Frame /TASK "android.process.media"
Task: "android.process.media"
-000 | need_resched()
-001 | schedule_hrtimeout_range_clock(expires = 0x0, delta = 0, mode = HRTIM
-002 | sys_epoll_wait(?, events = 0xBEF79530, maxevents = 16, ?)
-003 | ret_fast_syscall(asm)
-004 | exception
-005 | epoll_wait(asm)
-006 | android::Looper::pollInner(this = 0x5D3B12A0, timeoutMillis = -1)
-007 | android::Looper::pollOnce(this = 0x5D3B12A0, timeoutMillis = -1, outF
-008 | ZN7androidL38android_os_MessageQueue_nativePollOnceEP7_JNIEnvP8_jobje
-009 | dvmPlatformInvoke(asm)
-010 | dvmCallJNIMethod(args = 0x412ABE18, pResult = 0x41491A20, ?, self = 0
-011 | dvmCheckCallJNIMethod(?, pResult = 0x41491A20, method = 0x56F33978, s
-012 | JAVA → NATIVE
-013 | next()()
-014 | next()()
-015 | loop()()
-016 | main(java.lang.String []())
-017 | J:0x656:0x6(bytecode)
-018 | invoke(java.lang.Object,java.lang.Object []())
-019 | run()()
-020 | main(java.lang.String []())
-021 | J:0x656:0x0(bytecode)
-022 | NATIVE → JAVA
-023 | t32_dalvik_vm()
-024 | end of frame
  
```

# The TRACE32 Dalvik Awareness

- DEX code disassembly



# The Android RunTime (ART)

- Introduced in 4.4 „KitKat“ (available only through developer options)
- Supersedes Dalvik since 5.x Lollipop
- Introduces ahead-of-time (AOT) compilation instead of just-in-time (JIT)
  - Android Framework compiled to native code when building Android
  - Apps are compiled at install time



# The Android RunTime (ART)

- ART compiles DEX to native code (OAT files)
- OAT files are ELF files with additional OAT data and embedded dex files (up to Android 7.0)

The screenshot displays a debugger interface with three windows:

- Top Left Window (B::d 0):** Shows a memory dump starting at address 0. The first row is highlighted, showing address 464C457F and a value of 03010102. The dump includes columns for address, hex values, and a character representation.
- Bottom Left Window (B::Data.dump D:0x1000):** Shows another memory dump starting at address 0A74616F. The first row is highlighted, showing address 0A74616F and a value of 00393730. The dump includes columns for address, hex values, and a character representation.
- Top Right Window (B::y.l.sec):** Shows a list of files and sections. The first row is highlighted, showing the path \\base\\.rodata and the section name .rodata. The list includes various system files and sections.



# The Android RunTime (ART)

- OAT files can also be compiled with DWARF debug info
- DWARF info can be enabled for apps before installation with

```
$ adb shell setprop debug.generate-debug-info true
```

- Most debug info can also be extracted from the OAT data  
→ new TRACE32 command: **Data.LOAD.OAT**
- Autoloader first tries to load the OAT file as ELF using **Data.LOAD.ELF**
- If no debug info is found then the Autoloader loads the files using **Data.LOAD.OAT**

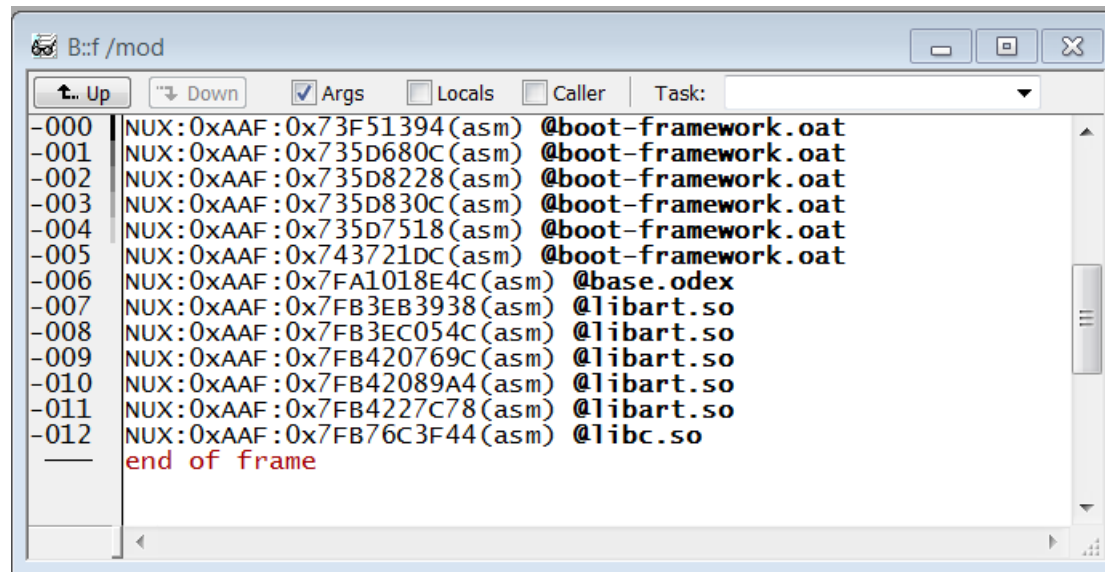
# The Android RunTime (ART) - Demo

- RAM Dump from the Hikey board (ARM Cortex-A53) Loaded on the Simulator
- Android 6.0 Marshmallow



# The Android RunTime (ART) - Demo

- Stack frame for the current task (sieve app) before loading the symbols. The app code has been compiled to base.odex and Android framework to boot-framework.oat. Both files are OAT files.



```

B:f/mod
[Up] [Down] [x] Args [ ] Locals [ ] Caller Task:
-000 NUX:0xAAF:0x73F51394(asm) @boot-framework.oat
-001 NUX:0xAAF:0x735D680C(asm) @boot-framework.oat
-002 NUX:0xAAF:0x735D8228(asm) @boot-framework.oat
-003 NUX:0xAAF:0x735D830C(asm) @boot-framework.oat
-004 NUX:0xAAF:0x735D7518(asm) @boot-framework.oat
-005 NUX:0xAAF:0x743721DC(asm) @boot-framework.oat
-006 NUX:0xAAF:0x7FA1018E4C(asm) @base.odex
-007 NUX:0xAAF:0x7FB3EB3938(asm) @libart.so
-008 NUX:0xAAF:0x7FB3EC054C(asm) @libart.so
-009 NUX:0xAAF:0x7FB420769C(asm) @libart.so
-010 NUX:0xAAF:0x7FB42089A4(asm) @libart.so
-011 NUX:0xAAF:0x7FB4227C78(asm) @libart.so
-012 NUX:0xAAF:0x7FB76C3F44(asm) @libc.so
    end of frame
  
```

# The Android RunTime (ART) - Demo

- The symbol autoloader load base.odex as ELF/DWARF since it contains DWARF info and boot-framework.oat as OAT.

```

B::f/mod
[Up] [Down] [Args] [Locals] [Caller] Task:
-000 android.os.MessageQueue.enqueueMessage(android::os::Message,long)(?, ?) @boot-framework.oa
-001 android.os.Handler.enqueueMessage(android::os::MessageQueue,android::os::Message,long)(?,
-002 android.os.Handler.sendMessageAtTime(android::os::Message,long)(?, ?) @boot-framework.oat
-003 android.os.Handler.sendMessageDelayed(android::os::Message,long)(?, ?) @boot-framework.oat
-004 android.os.Handler.post(java::lang::Runnable)(?) @boot-framework.oat
-005 android.view.View.post(java::lang::Runnable)(?) @boot-framework.oat
-006 void com.lauterbach.ltest2.MainActivity$SieveThread.run()(this = ()) @base.odex
-007 NUX:0xAAF:0x7FB3EB3938(asm) @libart.so
-008 NUX:0xAAF:0x7FB3EC0594(asm) @libart.so
-009 art::Trace::DumpMethodList(this = 0x0, ?, ?) @libart.so
-010 art::verifier::MethodVerifier::VerifyPrimitivePut() @libart.so
    end of frame
  
```

# The Android RunTime (ART) - Demo

The screenshot displays the Android Studio IDE with the `MainActivity.java` file open. The code defines a `SieveThread` class that extends `Thread`. It includes static variables `SIZE` and `flags`, and instance variables `mContinueFlag` and `mCount`. The `run()` method is overridden to perform a sieve algorithm, setting the thread name to "SieveThread", initializing flags, and entering a loop that updates `mCount` and posts to `mPrimeCounter`. A `Runnable` inner class is also defined to update the text of `mPrimeCounter`.

Below the code editor, a debugger window titled `B:f/mod` shows the stack trace for the `SieveThread.run()` method. The stack trace includes the following frames:

- 006 void com.lauterbach.lbttest2.MainActivity\$SieveThread.run()(this = ()) @base.odex
- 007 NUX:0xAAF:0x7FB3EB3938(asm) @libart.so
- 008 NUX:0xAAF:0x7FB3EC0594(asm) @libart.so
- 009 art::Trace::DumpMethodList(this = 0x0, ?, ?) @libart.so
- 010 art::verifier::MethodVerifier::VerifyPrimitivePut() @libart.so
- end of frame

# The Android RunTime (ART) - Demo

The screenshot displays the Android Studio interface with the ART runtime stack trace and assembly code. The main window shows the stack trace for the method `android.os.Handler.post(java.lang Runnable):`. The stack trace is as follows:

```

Nux:0AAF:735D74BC 00000000
Nux:0AAF:735D74C0 00000084
Nux:0AAF:735D74C4 D1400BF0 android.os.Handler.post(java.lang Runnable):
Nux:0AAF:735D74C8 B940021F
Nux:0AAF:735D74CC F81B0FE0
Nux:0AAF:735D74D0 A90357F4
Nux:0AAF:735D74D4 A9047BF6
Nux:0AAF:735D74D8 79400270
Nux:0AAF:735D74E0
Nux:0AAF:735D74E4
Nux:0AAF:735D74E8
Nux:0AAF:735D74EC
Nux:0AAF:735D74F0
Nux:0AAF:735D74F4
Nux:0AAF:735D74F8
Nux:0AAF:735D7500
Nux:0AAF:735D7504
Nux:0AAF:735D7508
Nux:0AAF:735D750C F941EC00
Nux:0AAF:735D7510 F940181E
Nux:0AAF:735D7514 D63F03C0
Nux:0AAF:735D7518 A94357F4
Nux:0AAF:735D751C A9447BF6
  
```

The assembly code for the method `android.os.Handler.post(java.lang Runnable):` is shown below the stack trace:

```

    00000000: undef 0x0
    00000084: undef 0x84
    00000088: sub x16,SP,#0x2,ls1 #0x0C ; x16
    0000008C: ldr wZR,[x16]
    00000090: str x0,[SP,#-0x50]! ; x0,[SP,#-
    00000094: stp x20,x21,[SP,#0x30] ; x20,x2
    00000098: stp x22,x30,[SP,#0x40] ; x22,x3
    0000009C: ldrh w16,[x19]
  
```

The stack trace also shows the following methods:

```

-004 android.os.Handler.post(java.lang Runnable)(?) @boot-framework.oat
-005 android.view.View.post(java.lang Runnable)(?) @boot-framework.oat
-006 void com.lauterbach.ltest2.MainActivity$SieveThread.run()(this = ()) @base.odex
-007 Nux:0xAAF:0x7FB3EB3938(asm) @libart.so
-008 Nux:0xAAF:0x7FB3EC0594(asm) @libart.so
-009 art::Trace::DumpMethodList(this = 0x0, ?, ?) @libart.so
  
```

The stack trace also shows the following methods:

```

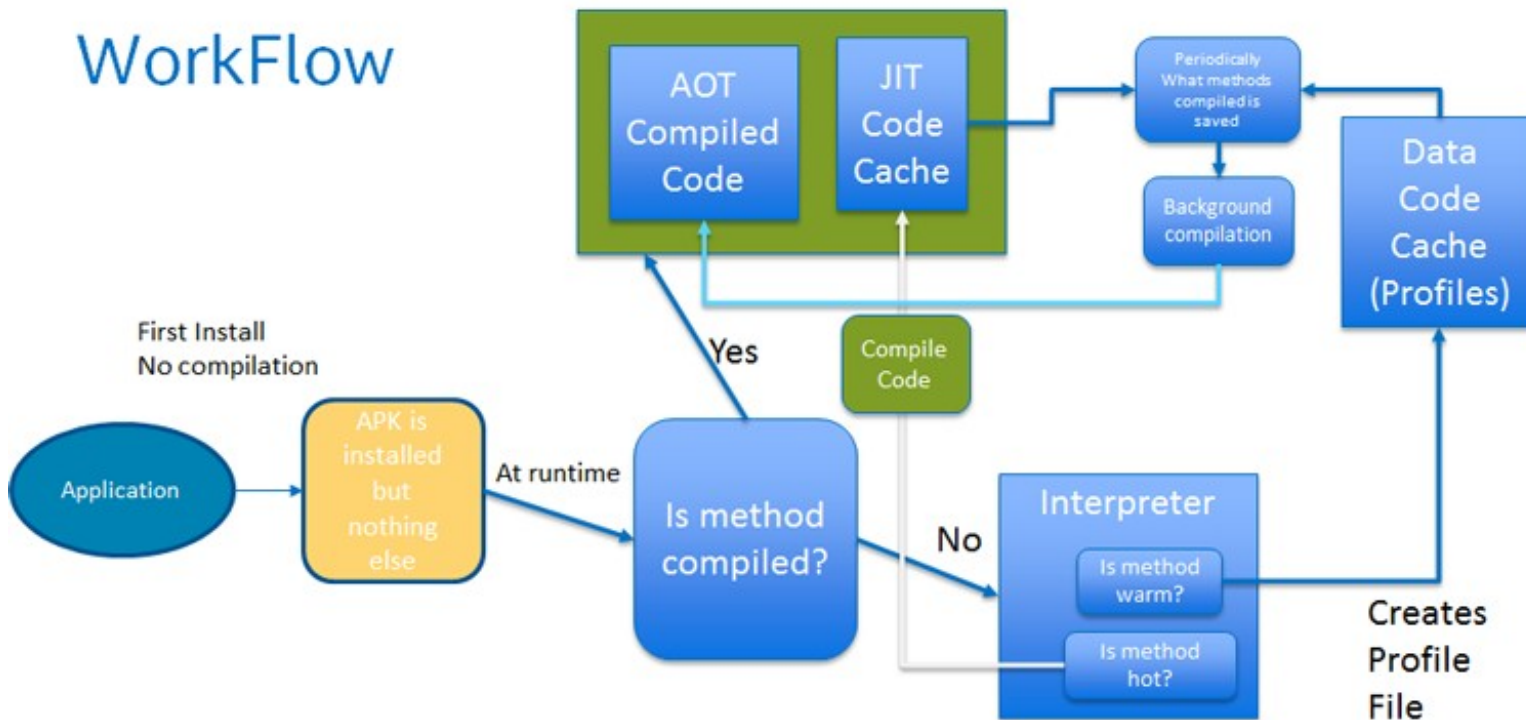
    00000000: ldr x0,[x0,#0x3D8] ; x0,[x0,#98
    00000004: ldr x30,[x0,#0x30] ; x30,[x0,#4
    00000008: blr x30
    0000000C: ldp x20,x21,[SP,#0x30] ; x20,x2
    00000010: ldp x22,x30,[SP,#0x40] ; x22,x3
  
```

# ART and Hybrid Compilation

- Android 7 (Nougat) introduces JIT/AOT hybrid compilation
- The Android Framework is still compiled ahead-of-time
- Apps are per default not compiled at install time:
  - An interpreter initially runs all the byte code and profiles often-executed methods („hot“)
  - „hot“ methods are compiled by the JIT compiler into native executable code which stored in the JIT cache along with the collected profile information
  - When the device is unused and charging for over long duration, a service will compile the hot methods and save the generated code



# ART and Hybrid Compilation



Source: Android: The Road to JIT/AOT Hybrid Compilation-Based Application User Experience By Rahul K. (Intel), Jean Christophe Beyler (Intel), Paul H. (Intel)

# ART and Hybrid Compilation

- JIT can be disabled with following commands:

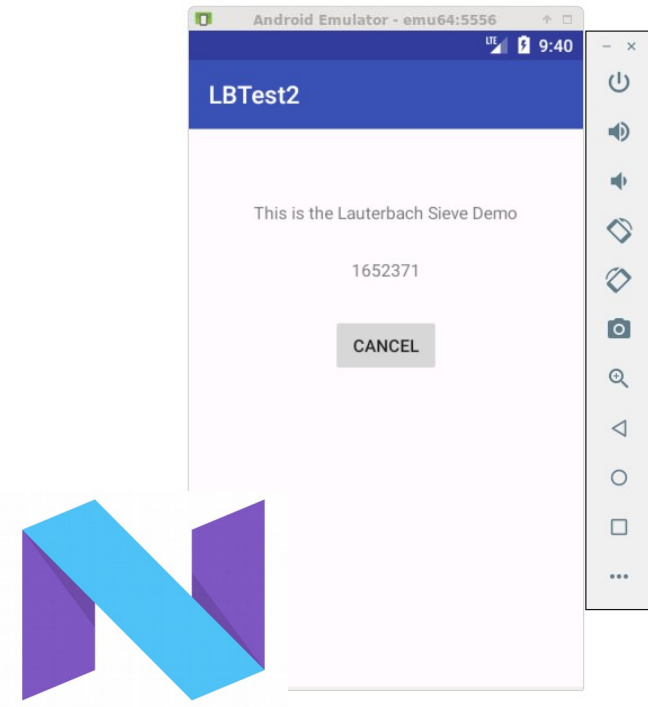
```
$ adb shell stop  
$ adb shell setprop dalvik.vm.usejit false  
$ adb shell start
```

- Ahead-Of-Time compilation for apps can be enabled using the following setup before installation:

```
$ adb shell setprop pm.dexopt.install everything
```

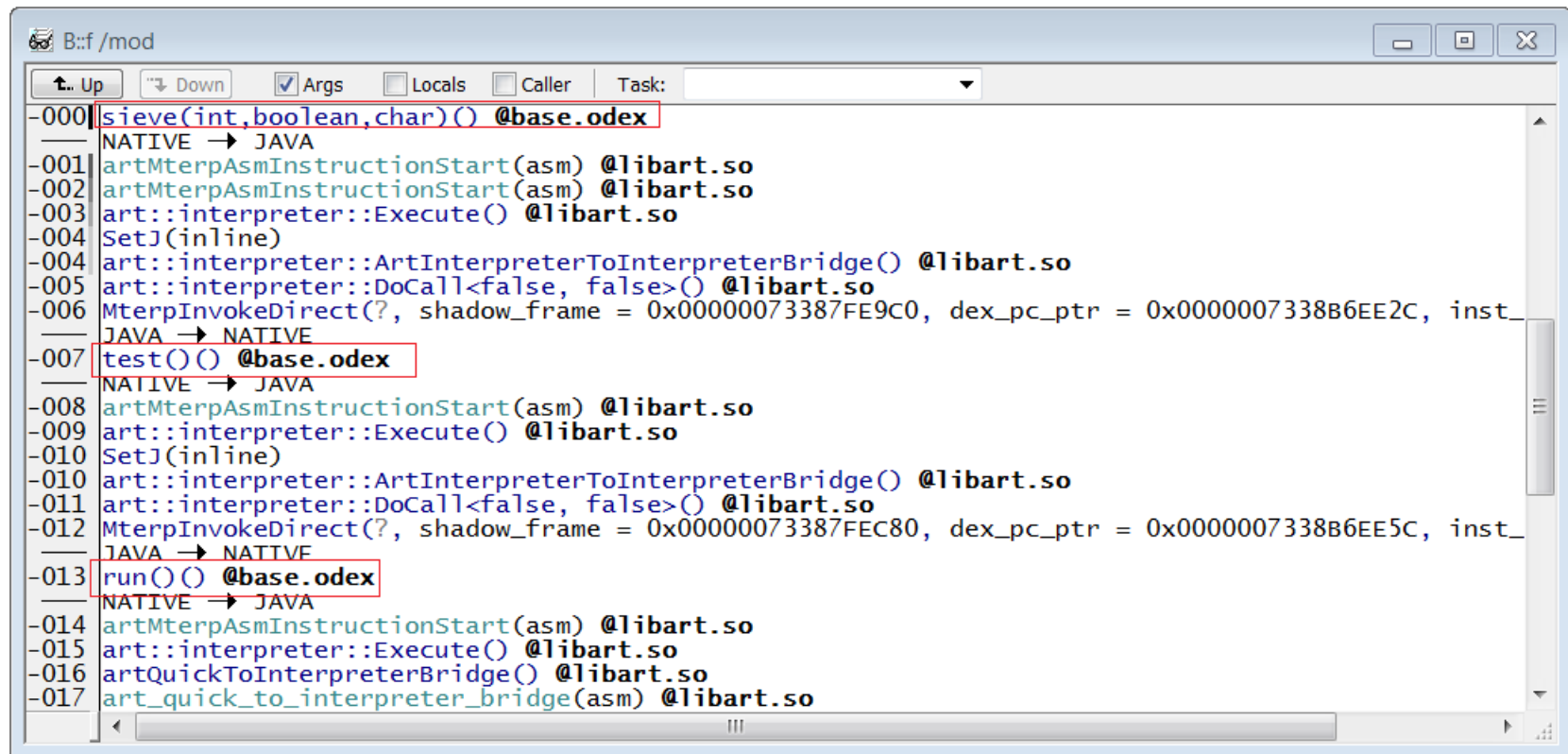
# ART and Hybrid Compilation - Demo

- Android 7.0
- RAM dumps from the Android Emulator (QEMU)
- Disabled JIT



# Android 7.0 – Enabled JIT

- TRACE32 can display the stack frame with the Java to native and native to Java transitions.



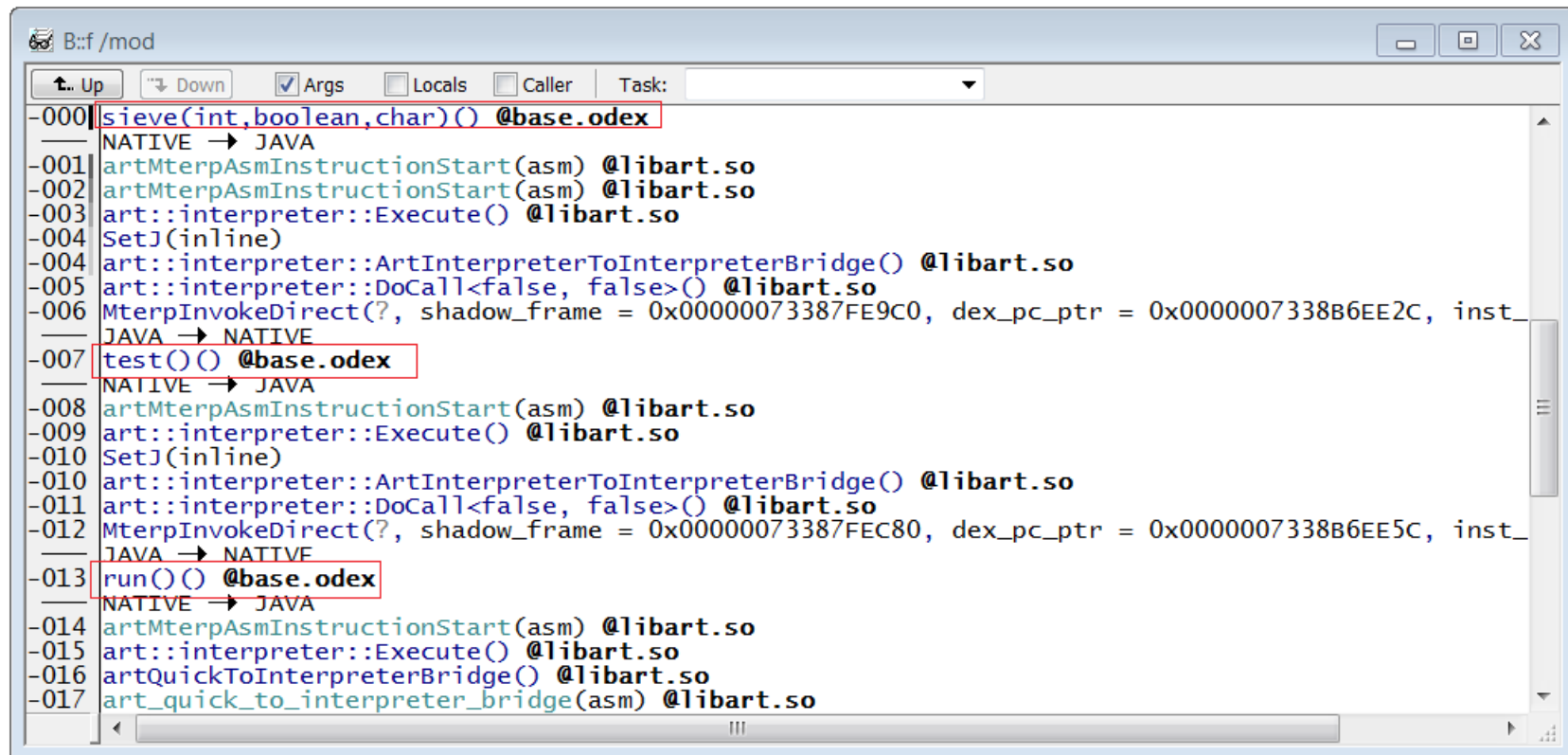
The screenshot shows the TRACE32 debugger window with the following stack trace:

```

B::f/mod
[Up] [Down] [Args] [Locals] [Caller] Task:
-000 sieve(int,boolean,char)() @base.odex
    NATIVE → JAVA
-001 artMterpAsmInstructionStart(asm) @libart.so
-002 artMterpAsmInstructionStart(asm) @libart.so
-003 art::interpreter::Execute() @libart.so
-004 SetJ(inline)
-004 art::interpreter::ArtInterpreterToInterpreterBridge() @libart.so
-005 art::interpreter::DoCall<false, false>() @libart.so
-006 MterpInvokeDirect(?, shadow_frame = 0x00000073387FE9C0, dex_pc_ptr = 0x0000007338B6EE2C, inst_
    JAVA → NATIVE
-007 test()() @base.odex
    NATIVE → JAVA
-008 artMterpAsmInstructionStart(asm) @libart.so
-009 art::interpreter::Execute() @libart.so
-010 SetJ(inline)
-010 art::interpreter::ArtInterpreterToInterpreterBridge() @libart.so
-011 art::interpreter::DoCall<false, false>() @libart.so
-012 MterpInvokeDirect(?, shadow_frame = 0x00000073387FEC80, dex_pc_ptr = 0x0000007338B6EE5C, inst_
    JAVA → NATIVE
-013 run()() @base.odex
    NATIVE → JAVA
-014 artMterpAsmInstructionStart(asm) @libart.so
-015 art::interpreter::Execute() @libart.so
-016 artQuickToInterpreterBridge() @libart.so
-017 art_quick_to_interpreter_bridge(asm) @libart.so
  
```

# Android 7.0 – Enabled JIT

- TRACE32 can display the stack frame with the Java to native and native to Java transitions.



The screenshot shows the TRACE32 debugger window with the following stack trace:

```

B::f/mod
[Up] [Down] [Args] [Locals] [Caller] Task:
-000 sieve(int,boolean,char)() @base.odex
    NATIVE → JAVA
-001 artMterpAsmInstructionStart(asm) @libart.so
-002 artMterpAsmInstructionStart(asm) @libart.so
-003 art::interpreter::Execute() @libart.so
-004 SetJ(inline)
-004 art::interpreter::ArtInterpreterToInterpreterBridge() @libart.so
-005 art::interpreter::DoCall<false, false>() @libart.so
-006 MterpInvokeDirect(?, shadow_frame = 0x00000073387FE9C0, dex_pc_ptr = 0x0000007338B6EE2C, inst_
    JAVA → NATIVE
-007 test()() @base.odex
    NATIVE → JAVA
-008 artMterpAsmInstructionStart(asm) @libart.so
-009 art::interpreter::Execute() @libart.so
-010 SetJ(inline)
-010 art::interpreter::ArtInterpreterToInterpreterBridge() @libart.so
-011 art::interpreter::DoCall<false, false>() @libart.so
-012 MterpInvokeDirect(?, shadow_frame = 0x00000073387FEC80, dex_pc_ptr = 0x0000007338B6EE5C, inst_
    JAVA → NATIVE
-013 run()() @base.odex
    NATIVE → JAVA
-014 artMterpAsmInstructionStart(asm) @libart.so
-015 art::interpreter::Execute() @libart.so
-016 artQuickToInterpreterBridge() @libart.so
-017 art_quick_to_interpreter_bridge(asm) @libart.so
  
```

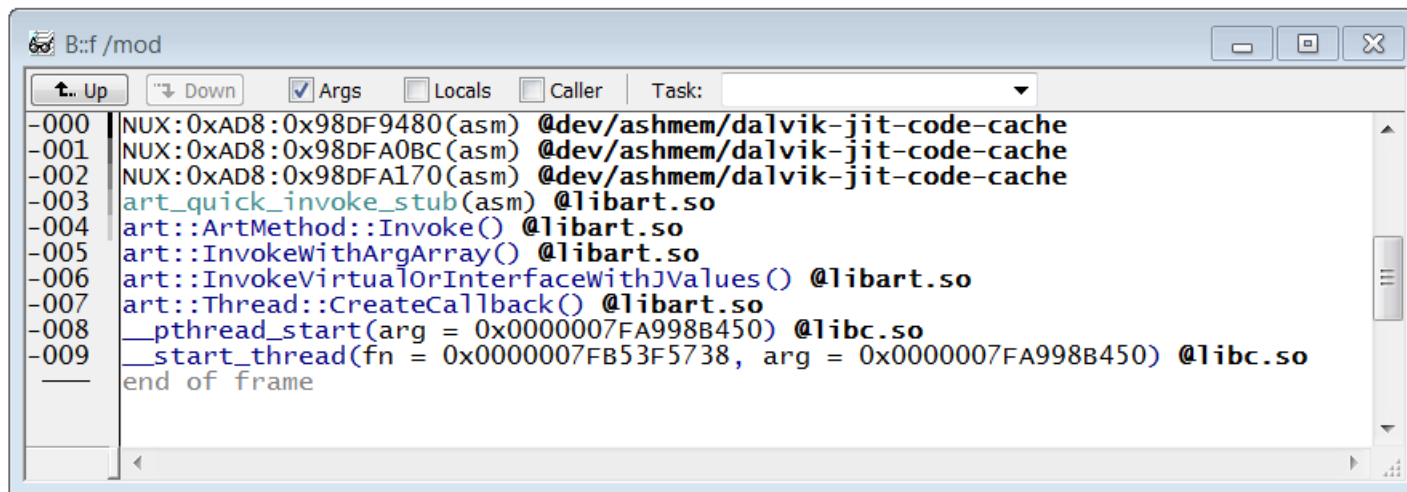
# ART and Hybrid Compilation - Demo

---

- Android 7.1
- RAM dumps from HiKey board
- Enabled JIT

# Android 7.1 – Enabled JIT

- Frame window shows that the application is executing in the dalvik jit code cache.



```

B:\f/mod
└ Up  └ Down  [x] Args  [ ] Locals  [ ] Caller  Task:
-000 | NUX:0xAD8:0x98DF9480(asm) @dev/ashmem/dalvik-jit-code-cache
-001 | NUX:0xAD8:0x98DFA0BC(asm) @dev/ashmem/dalvik-jit-code-cache
-002 | NUX:0xAD8:0x98DFA170(asm) @dev/ashmem/dalvik-jit-code-cache
-003 | art_quick_invoke_stub(asm) @libart.so
-004 | art::ArtMethod::Invoke() @libart.so
-005 | art::InvokeWithArgArray() @libart.so
-006 | art::InvokeVirtualOrInterfaceWithJValues() @libart.so
-007 | art::Thread::CreateCallback() @libart.so
-008 | __pthread_start(arg = 0x0000007FA998B450) @libc.so
-009 | __start_thread(fn = 0x0000007FB53F5738, arg = 0x0000007FA998B450) @libc.so
    | end of frame
  
```



# Android 7.1 – Enabled JIT

- Android ART Awareness can get the name of the methods corresponding to the jit code cache addresses (**work in progress**)

```

B::f /mod
┌ Up ──┐ ── Down ┐ [x] Args [ ] Locals [ ] Caller Task:
-000 com.lauterbach.ltest2.MainActivity$SieveThread.sieve(int,boolean,char)() @dev/ashmem/dalvik-jit-code-cache
-001 com.lauterbach.ltest2.MainActivity$SieveThread.test()() @dev/ashmem/dalvik-jit-code-cache
-002 com.lauterbach.ltest2.MainActivity$SieveThread.run()() @dev/ashmem/dalvik-jit-code-cache
-003 art_quick_invoke_stub(asm) @libart.so
-004 art::ArtMethod::Invoke() @libart.so
-005 art::InvokeWithArgArray() @libart.so
-006 art::InvokeVirtualOrInterfaceWithJValues() @libart.so
-007 art::Thread::CreateCallback() @libart.so
-008 __pthread_start(arg = 0x00000007FA998B450) @libc.so
-009 __start_thread(fn = 0x00000007FB53F5738, arg = 0x00000007FA998B450) @libc.so
— end of frame
  
```

# Thank you!

▪ Khaled JMAL ▪  
2016 / 11 / 17

# Questions?