Dalvik and ART

Jonathan Levin

http://NewAndroidBook.com/

http://www.technologeeks.com

Did you attend part I?

- If you didn't, at least get the presentation
  - http://NewAndroidBook.com/files/Andevcon-DEX.pdf
  - http://NewAndroidBook.com/files/Andevcon-ART.pdf (is part II)

- Maybe you should have.. ART builds over DEX
  - We'll refer back to DEX nomenclature as we go along
  - Feel free to pause for questions at any time.

What we **won't** be discussing

- The nitty-gritty, molecular-level internals of ART
  - Code Generation down to the assembly level
  - LLVM integration
  - Internal memory structures
- Because...
  A) This level has only recently meta-stabilized

  (ART in 5.0 is not compatible with 4.4.x's, **or** the preview releases.

  B) We don't really have time to go that deep (71 Mins to go!)

  C) There's a chapter in the book for that*

  q.v. www.newAndroidBook.com (tip: Follow RSS or @Technologeeks)

*** - Well, at least there will be. Still working on updating that chapter with a massive rewrite, unfortunately..**

What we **will** be discussing

- High level architecture and principles
- ART and OAT file structure
- ART code generation at a high level view
- ART reversing
- Debugging in ART (high-level)

Interlude (Necessary Plug*)

- Me: Jonathan Levin, CTO of http://Technologeeks.com
  - Training and consulting on internals/debugging, networking
  - Follow us on Twitter (@Technologeeks), Etc. Etc. Etc
- My Book: "Android Internals: A Confectioner's Cookbook"
  - http://www.NewAndroidBook.com/ for tools, articles, and more
  - Unofficial sequel to Karim Yaghmour's "Embedded Android"
    - More on the **how** and **why** Android frameworks and services work
  - (presently) only in-depth book on the subject
- Just in case anyone's into iOS (w/41% probability?)
  - http://www.newosxbook.com/
  - 2nd Edition (covers iOS 8, OS X 10.10) due March '15

*** - Keeping it quick for those people who sat through Part I ☺**

Part II - ART

The Android RunTime

- ART was introduced in KitKat (4.4):
  - Available only through developer options
  - Declared to be a “preview” release, use-at-your-own-risk
  - Very little documentation, if any
  - Some performance reviews (e.g. AnandTech), but only for Preview Release
- In Lollipop, ART becomes the RunTime of choice
  - Supersedes (all but buries) Dalvik
  - Breaks compatibility with older DEX, as well as itself (in preview version)
  - And still – very little documentation, if any

Dalvik Disadvantages

- ART was designed to address the shortcomings of Dalvik:
  - Virtual machine maintenance is expensive
    - Interpreter/JIT simply aren't efficient as native code
    - Doing JIT all over again on every execution is wasteful
    - Maintenance threads require significantly more CPU cycles
    - CPU cycles translate to slower performance – and shorter battery life
  - Dalvik garbage collection frequently causes hangs/pauses
  - Virtual machine architecture is 32-bit only
    - Android is following iOS into the 64-bit space

... Become ART Advantages

- ART moves compilation from **Just-In-Time** to **Ahead-Of-Time**
  - Virtual machine maintenance is **not as** expensive
    - Interpreter/JIT simply aren't efficient as native code **ART compiles to native**
    - Doing JIT all over again on every execution is wasteful **Just ONCE, AOT**
    - Maintenance threads require significantly more CPU cycles **Less threads**
    - ~~CPU cycles translate to slo~~wer performance – and shorter battery life

      Less overhead cycles
  - Dalvik garbage collection ~~frequently causes hangs/pauses~~

    GC Parellizable (foreground/background),

    Non-blocking (i.e. less GC_FOR_ALLOC)
  - Virtual machine architecture is 32-bit only
    - Android is following iOS into the 64-bit space

      (Some issues still exist here)

Main Idea of ART - AOT

- Actually, compilation can be to one of two types:
  - QUICK: Native Code
  - Portable: LLVM Code
- In practice, preference is to compile to Native Code
  - Portable implies another layer of IR (LLVM's BitCode)

The Android RunTime

- ART uses not one, but two file formats:
  - .art:
    - Only one file, **boot.art**, in /system/framework/[arch] (arm, arm64, x86_64)
  - .oat:
    - Master file, **boot.oat**, in /system/framework/[arch] (arm, arm64, x86_64)
    - .odex files: **NO LONGER Optimized DEX, but OAT!**
      - alongside APK for system apps/frameworks
      - /data/dalvik-cache for 3rd-party apps

ART files

- The ART file is a proprietary format
  - Poorly documented (which is why I wrote the book)
  - changed format internally repeatedly (which is why book was so delayed)
  - Not really understood by oatdump, either.. (which is why I wrote dextra)
  - And.. changed again in L (ART009 vs. 005).. (which is why I'm rewriting the tool)
- ART file maps in memory right before OAT, which links with it.
- Contains pre-initialized classes, objects, and support structures

Creating ART (and OAT)

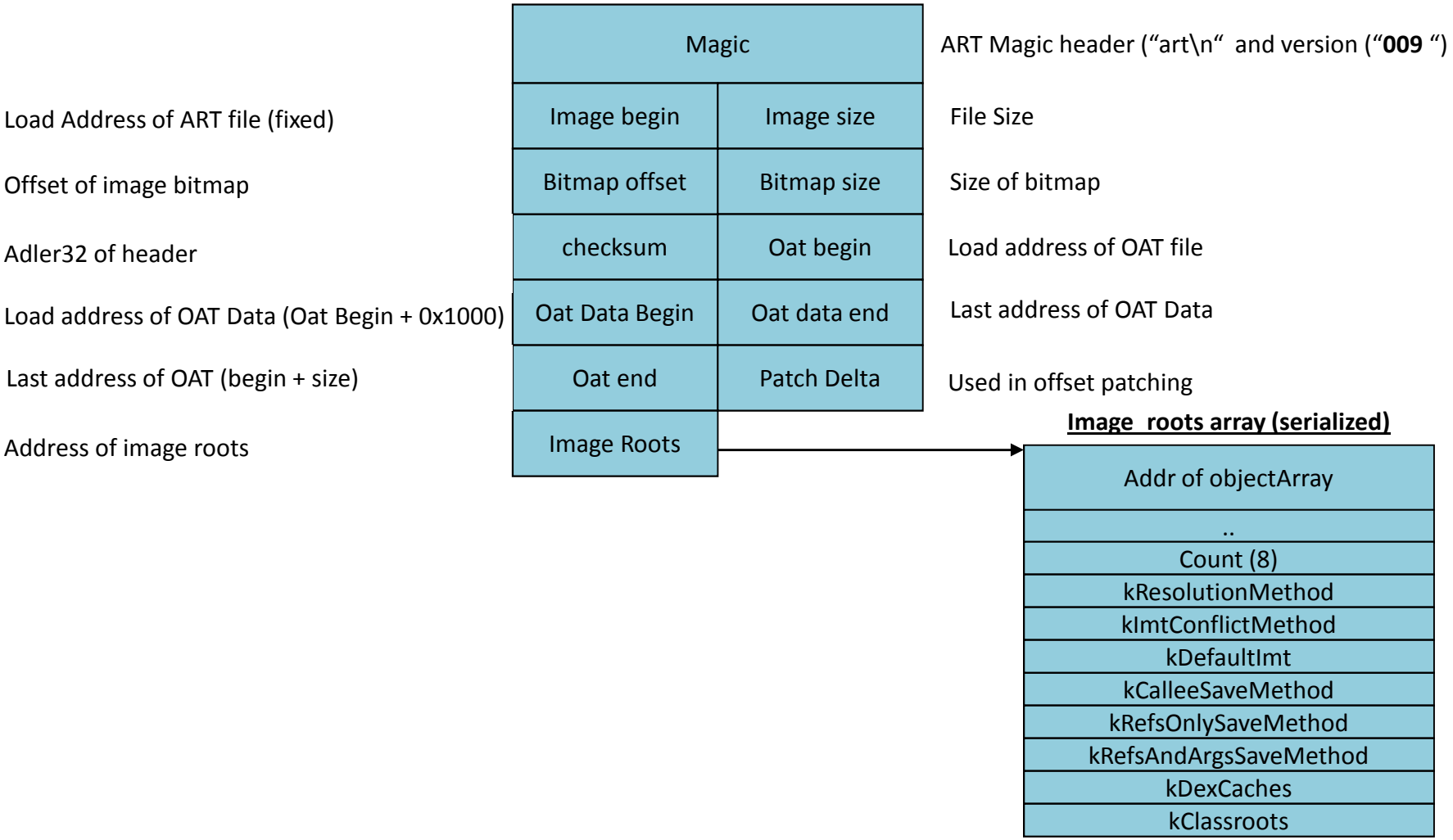
- In KK (ART is optional) you can see ART and OAT file creation:

Experiment: Behind the scenes of Dalvik to ART conversion

You can see what happens behind the scenes when the runtime is converted from Dalvik to ART by toggling the change, and then using `adb logcat` as the device reboot. The `adb` command will wait for the device to become online again, and then spit out the log statements made by GenerateImage:

```
D/AndroidRuntime(   51): >>>>>> AndroidRuntime START com.android.internal.os.ZygoteInit <<<<<<
D/AndroidRuntime(   51): CheckJNI is ON
I/art     (   51): option[0]=-Xzygote
I/art     (   51): option[1]=exit
# .. Command line arguments to Zygote ..
I/art     (   51): option[14]=-Djava.io.tmpdir=/sdcard
# Command line used by GenerateImage:
I/art     (   51): GenerateImage: /system/bin/dex2oat --image=/data/dalvik-cache/system@framework@boot.art
  --runtime-arg -Xms64m --runtime-arg -Xmx64m --dex-file=/system/framework/core-libart.jar
  --dex-file=/system/framework/conscrypt.jar --dex-file=/system/framework/okhttp.jar
  --dex-file=/system/framework/core-junit.jar --dex-file=/system/framework/bouncycastle.jar
  --dex-file=/system/framework/ext.jar --dex-file=/system/framework/framework.jar
  --dex-file=/system/framework/framework2.jar --dex-file=/system/framework/telephony-common.jar
  --dex-file=/system/framework/voip-common.jar --dex-file=/system/framework/mms-common.jar
  --dex-file=/system/framework/android.policy.jar --dex-file=/system/framework/services.jar
  --dex-file=/system/framework/apache-xml.jar --dex-file=/system/framework/webviewchromium.jar
  --oat-file=/data/dalvik-cache/system@framework@boot.oat --base=0x60000000
  --image-classes-zip=/system/framework/framework.jar --image-classes=preloaded-classes
..
```

The ART file format



Loading the ART file

The ART file mapping in memory is fixed (as art the .OAT)

```
root@generic:/ # cat /proc/1088/maps | grep boot
70dbd000-718db000 rw-p 00000000 1f:01 7053       .../system@framework@boot.art
718db000-7338c000 r--p 00000000 1f:01 7054       .../system@framework@boot.oat
7338c000-74844000 r-xp 01ab1000 1f:01 7054       .../system@framework@boot.oat
74844000-74845000 rw-p 02f69000 1f:01 7054       .../system@framework@boot.oat
b5242000-b5243000 r--p 00000000 1f:01 7054       .../system@framework@boot.oat
b5244000-b5271000 r--p 00b1e000 1f:01 7053       .../system@framework@boot.art
```

```
morpheus@Forge (~) # dextra ~/Tests/system@framework@boot.art
ART version 0x393030 header detected (header size: ox34, File Size 0xb4b000)
Image Begin: 70dbd000
Image Bitmap: 2d000 @0xb1e000-0xb4b000 (relocated separately from image base)
Patch Delta: 0xdbd000
Checksum: 0x5eae278
OAT file: 0x718db000-0x74845000 (not part of this image)
OAT data: 0x718dc000-0x74843690 (not part of this image)
```

Defeats the whole purpose of ASLR*, may be (eventually) patched

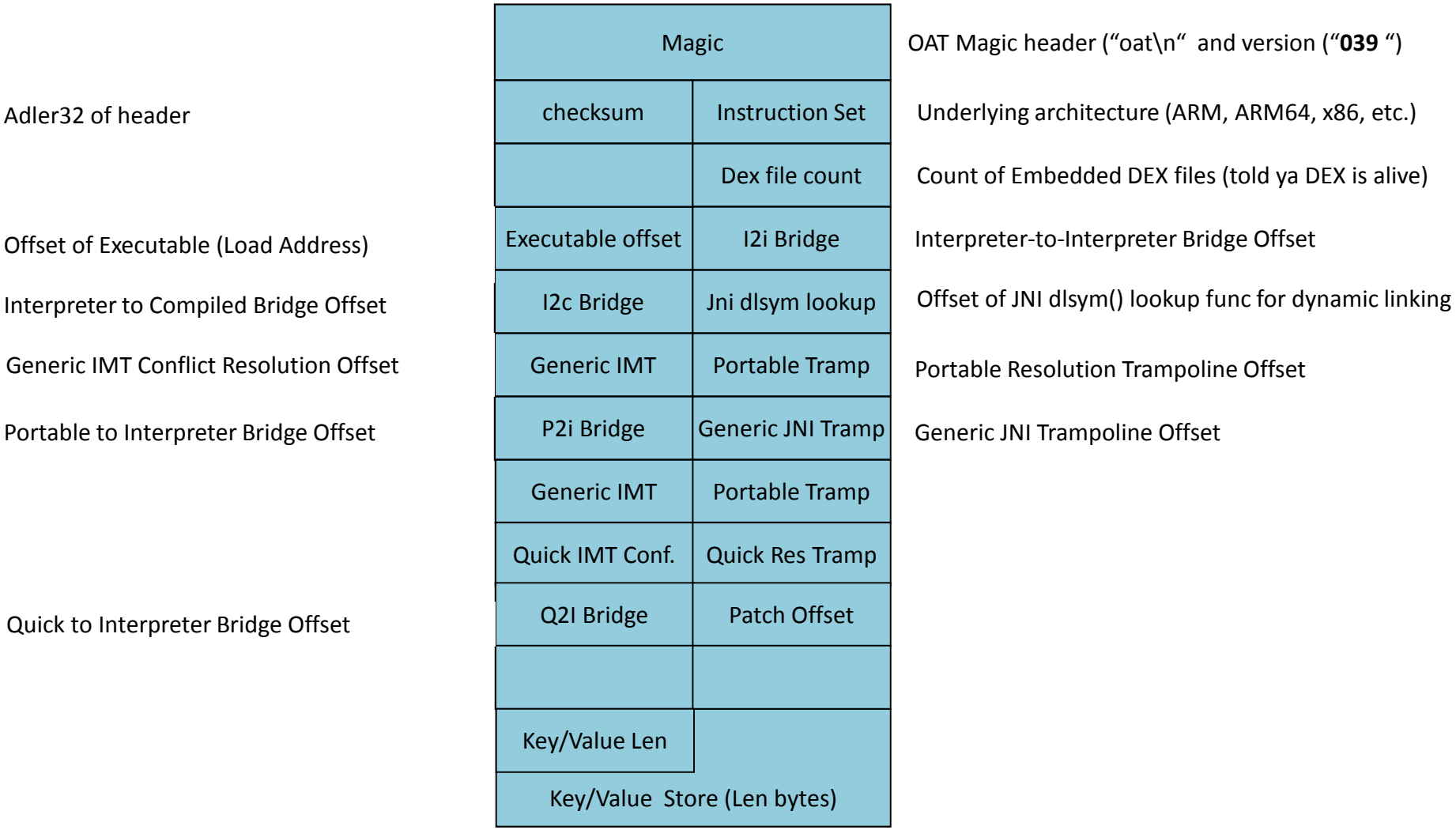
*** - the boot.oat is also pretty big – and executable (ROP gadgets, anyone?)**

OAT and ELF

- OAT files are actually embedded in ELF object files

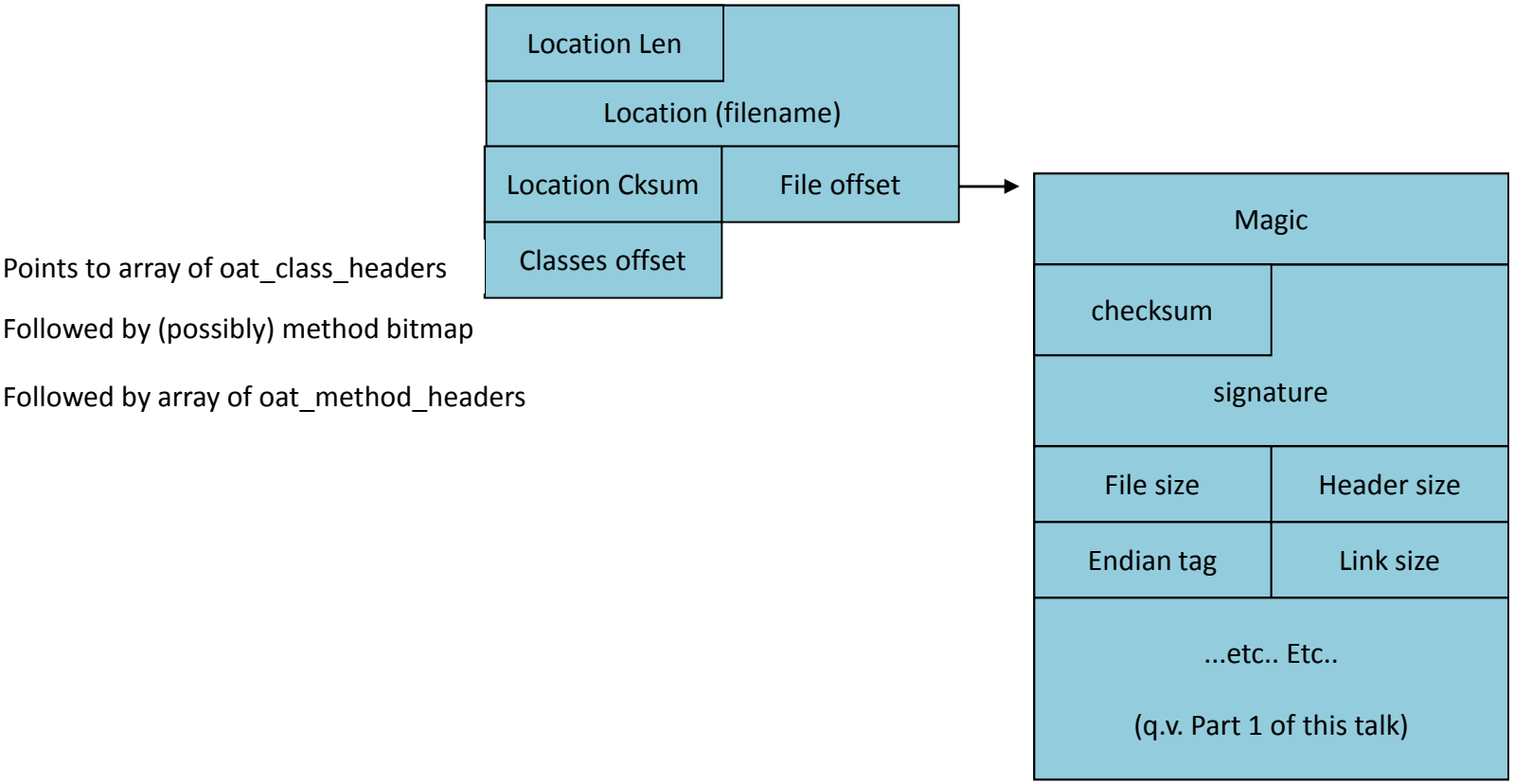
```
morpheus@Forge (~)$ file boot.oat
boot.oat: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (GNU/Linux),
dynamically linked, stripped

morpheus@Forge (~)$ readelf -e boot.oat
...
Section Headers:
  [Nr] Name              Type            Addr     Off      Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000   000000 00      0   0  0
  [ 1] .dynsym           DYNSYM          70b1e0d4 0000d4   000040 10   A  2   0  4
  [ 2] .dynstr           STRTAB          70b1e114 000114   000026 01   A  0   0  1
  [ 3] .hash             HASH            70b1e13c 00013c   000020 04   A  1   0  4
  [ 4] .rodata           PROGBITS        70b1f000 001000  1ab0000 00   A  0   0 4096
  [ 5] .text             PROGBITS        725cf000 1ab1000 14b7690 00  AX  0   0 4096
  [ 6] .dynamic          DYNAMIC         73a87000 2f69000  000038 08   A  1   0 4096
  [ 7] .oat_patches      LOUSER+0        00000000 2f69038  1148b8 04      0   0  4
  [ 8] .shstrtab         STRTAB          00000000 3085388  000045 01      0   0  1
```

The OAT file format



The OAT DexFile Header

- Following the OAT header are.. *surprise* - 1..n DEX files!
  - Actual value given by DexFileCount field in header



Finding DEX in OAT

- ODEX files will usually have only one (=original) DEX embdded
- BOOT.OAT is something else entirely:
  - Some 14 Dex Files – the “Best of” the Android Framework JARs
  - Each DEX contains potentially hundreds of classes

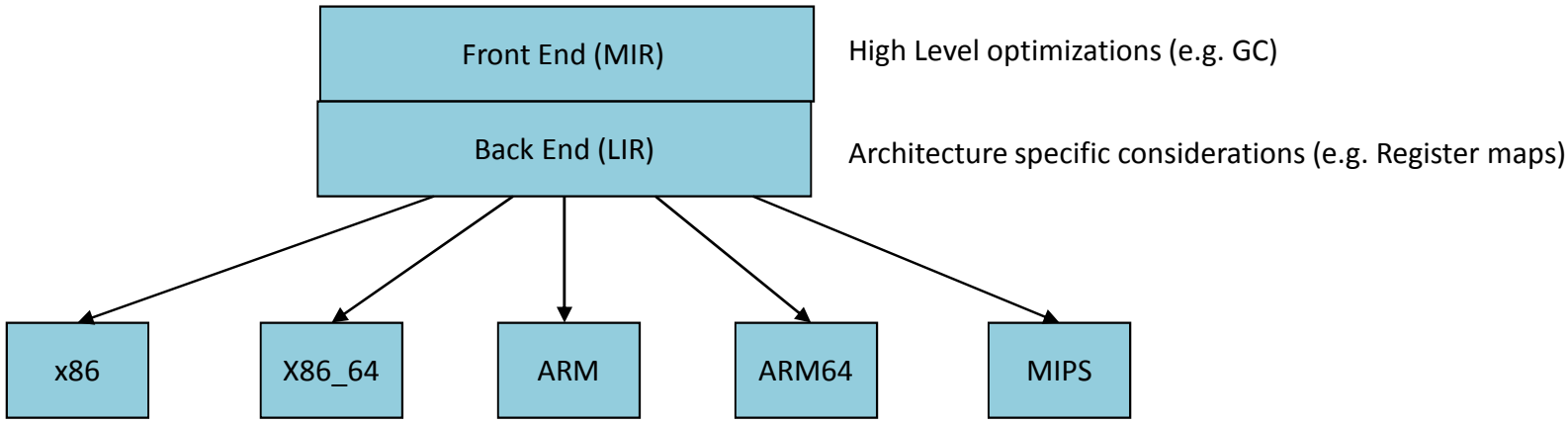
```
morpheus@Forge (~) % dextra Tests/boot.oat | grep DEX
DEX files: 14
DEX FILE 0: /system/framework/core-libart.jar @0xda10 (2132 classes)
DEX FILE 1: /system/framework/conscrypt.jar @0x2cfea8 (166 classes)
DEX FILE 2: /system/framework/okhttp.jar @0x311c14 (179 classes)
DEX FILE 3: /system/framework/core-junit.jar @0x3573f8 (19 classes)
DEX FILE 4: /system/framework/bouncycastle.jar @0x35d36c (824 classes)
DEX FILE 5: /system/framework/ext.jar @0x45dc40 (1017 classes)
DEX FILE 6: /system/framework/framework.jar @0x5a9508 (5858 classes)
DEX FILE 7: /system/framework/framework.jar:classes2.dex @0xef3c34 (1547 classes)
DEX FILE 8: /system/framework/telephony-common.jar @0x11e1b14 (551 classes)
DEX FILE 9: /system/framework/voip-common.jar @0x1369050 (76 classes)
DEX FILE 10: /system/framework/ims-common.jar @0x138e614 (42 classes)
DEX FILE 11: /system/framework/mms-common.jar @0x13a26e8 (1 classes)
DEX FILE 12: /system/framework/android.policy.jar @0x13a28a4 (117 classes)
DEX FILE 13: /system/framework/apache-xml.jar @0x13e4030 (658 classes)
```

ART Code Generation

- OAT Method headers point to offset of native code
- Each method has a Quick or Portable Method Header
  - Contains mapping from virtual register to underlying machine registers
- Each method also has a Quick or Portable Frame Info
  - Provides frame size in bytes
  - Core register spill mask
  - FP register spill mask (largely unused)
- Generated code uses unusual registers
  - Especially fond of using LR as call register
  - Still saves/restores registers so as not to violate ARM conventions

ART Code Generation

- ART supports multiple architectures (x86, ARM/64, MIPS)
- Compiler is a layered architecture*:



*** - Using Portable (LLVM) adds another level, with LLVM BitCode – which is outside the scope of this presentation**

Example: AM.ODEX

- For a practical example, we consider am.odex
  - Simple class, providing basic ActivityManager Command Line Interface
- We pick a simple method – runKillAll()
  - One line method, demonstrating botch instance field access and method invocation

frameworks/base/cmds/am/src/com/android/commands/am/Am.java

```
private void runKillAll() throws Exception {
        mAm.killAllBackgroundProcesses();
    }
```

DEX code

```
15: void com.android.commands.am.Am.runKillAll() (dex_method_idx=164)
  0x0000: iget-object v0, v1,
    Landroid/app/IActivityManager; com.android.commands.am.Am.mAm
  0x0002: invoke-interface {v0},
        void android.app.IActivityManager.killAllBackgroundProcesses()
  0x0005: return-void
```

oatdump –-oat-file=/system/frameworks/arm/am.odex

AM.ODEX (arm)

```
 0x00018d28: f5bd5c00      subs    r12, sp, #8192
 0x00018d2c: f8dcc000      ldr.w   r12, [r12, #0]
 suspend point dex PC: 0x0000
 GC map objects:  v1 (r6)
// Prolog: Stack setup, save registers
 0x00018d30: e92d40e0      push    {r5, r6, r7, lr}
 0x00018d34: b084          sub     sp, sp, #16
 0x00018d36: 1c07          mov     r7, r0
 0x00018d38: 9000          str     r0, [sp, #0]
 0x00018d3a: 1c0e          mov     r6, r1
 0x00018d3c: 6975          ldr     r5, [r6, #20]
 0x00018d3e: f04f0c11      mov.w   r12, #17        // Note - 17
 0x00018d42: 1c29          mov     r1, r5
 0x00018d44: 6808          ldr     r0, [r1, #0]
 suspend point dex PC: 0x0002 // invoke-interface {v0}, ...killAllBackground..
 GC map objects:  v0 (r5), v1 (r6)
 0x00018d46: f8d000f4      ldr.w   r0, [r0, #244]
 0x00018d4a: f8d0e028      ldr.w   lr, [r0, #40]   ; Method at offset 40
 0x00018d4e: 47f0          blx     lr              ; Execute method (note usage of lr)
 suspend point dex PC: 0x0002
 GC map objects:  v0 (r5), v1 (r6)
 0x00018d50: 3c01          subs    r4, #1          ; Check VM Thread State
 0x00018d52: f0008003      beq.w   +6 (0x00018d5c)
 // Epilog: Stack teardown, restore registers
 0x00018d56: b004          add     sp, sp, #16
 0x00018d58: e8bd80e0      pop     {r5, r6, r7, pc}
 0x00018d5c: f8d9e230      ldr.w   lr, [r9, #560]  ; pTestSuspend
 0x00018d60: 47f0          blx     lr     ; call pTestSuspend
 suspend point dex PC: 0x0005
 0x00018d62: e7f8          b       -16 (0x00018d56)
```

oatdump –-oat-file=/system/frameworks/arm64/am.odex

AM.ODEX (arm64)

```
0x0001c708: d1400be8      sub x8, sp, #0x2000 (8192)
0x0001c70c: f9400108      ldr x8, [x8]
suspend point dex PC: 0x0000 // iget-object v0, v1...
GC map objects:  v1 (r21)
0x0001c710: d100c3ff      sub sp, sp, #0x30 (48)
0x0001c714: a90157f4      stp x20, x21, [sp, #16]
0x0001c718: a9027bf6      stp x22, x30, [sp, #32]
0x0001c71c: aa0003f6      mov x22, x0
0x0001c720: b90003e0      str w0, [sp]
0x0001c724: aa0103f5      mov x21, x1
0x0001c728: b94016b4      ldr w20, [x21, #20]
0x0001c72c: 52800231      movz w17, #0x11    // 0x11 - 17
0x0001c730: aa1403e1      mov x1, x20
0x0001c734: b9400020      ldr w0, [x1]
suspend point dex PC: 0x0002 // invoke-interface {v0}, ...killAllBackground..
GC map objects:  v0 (r20), v1 (r21)
0x0001c738: b9413000      ldr w0, [x0, #304] ; note w0 (32 bit register usage)
0x0001c73c: f940141e      ldr x30, [x0, #40] ; method at offset 40
0x0001c740: d63f03c0      blr x30
suspend point dex PC: 0x0002
GC map objects:  v0 (r20), v1 (r21)
0x0001c744: 71000673      subs w19, w19, #0x1 (1)      // Check VM Thread State
0x0001c748: 540000a0      b.eq #+0x14 (addr 0xbeaf84b4)
0x0001c74c: a94157f4      ldp x20, x21, [sp, #16]
0x0001c750: a9427bf6      ldp x22, x30, [sp, #32]
0x0001c754: 9100c3ff      add sp, sp, #0x30 (48)
0x0001c758: d65f03c0      ret
0x0001c75c: f941f65e      ldr x30, [x18, #1000]
0x0001c760: d63f03c0      blr x30
suspend point dex PC: 0x0005
0x0001c764: 17fffffa      b #-0x18 (addr 0xbeaf84b8)
```

oatdump –-oat-file=/system/frameworks/x86_64/am.odex

AM.ODEX (x86_64)

```
0x0001bb18:         85842400E0FFFF        test    eax, [rsp + -8192]
suspend point dex PC: 0x0000
GC map objects:  v1 (r5)
// Prolog: Stack setup, save registers
0x0001bb1f:               4883EC28        subq    rsp, 40
0x0001bb23:           48895C2410          movq    [rsp + 16], rbx
0x0001bb28:           48896C2418          movq    [rsp + 24], rbp
0x0001bb2d:           4C89642420          movq    [rsp + 32], r12
0x0001bb32:               4C8BE7          movq    r12, rdi
0x0001bb35:               893C24          mov     [rsp], edi
0x0001bb38:               488BEE          movq    rbp, rsi
0x0001bb3b:               8B5D14          mov     ebx, [rbp + 20]
0x0001bb3e:           B811000000          mov     eax, 17    // Again, 17
0x0001bb43:               488BF3          movq    rsi, rbx
0x0001bb46:                 8B3E          mov     edi, [rsi]
suspend point dex PC: 0x0002
GC map objects:  v0 (r3), v1 (r5)
0x0001bb48:           8BBF34010000        mov     edi, [rdi + 308]
0x0001bb4e:               FF5728          call    [rdi + 40] ; Again, offset 40
suspend point dex PC: 0x0002
GC map objects:  v0 (r3), v1 (r5)
0x0001bb51:   6566833C250000000000        cmpw    gs:[0], 0  ; state_and_flags
0x0001bb5b:                 7514          jnz/ne  +20 (0x0001bb71)
// Epilog: Stack teardown, restore registers
0x0001bb5d:           488B5C2410          movq    rbx, [rsp + 16]
0x0001bb62:           488B6C2418          movq    rbp, [rsp + 24]
0x0001bb67:           4C8B642420          movq    r12, [rsp + 32]
0x0001bb6c:             4883C428          addq    rsp, 40
0x0001bb70:                   C3          ret
0x0001bb71:       65FF1425E8030000        call    gs:[1000]  ; pTestSuspend
suspend point dex PC: 0x0005
0x0001bb79:                 EBE2          jmp     -30 (0x0001bb5d)
0x0001bb7b:                 0000          addb    [rax], al ; padding (not executed)
```

Some lessons

- Base code is DEX – so VM is still 32-bit
  - No 64-bit registers or operands - so mapping to underlying arch isn't always 64-bit
- Generated code isn't always that efficient
  - Not on same par as an optimizing native code compiler
  - Likely to improve with LLVM optimizations
- Overall code flow (determined by MIR optimization) is same
- Garbage collection, register maps, likewise same
- Caveats:
  - Not all methods guaranteed to be compiled
  - Reversing can be quite a pain...

Example: Reversing OAT

- You can use the AOSP-supplied OATDUMP to disassemble OAT

```
Usage: oatdump [options] ...
   ...
  --oat-file=<file.oat>: specifies an input oat filename.

  --image=<file.art>: specifies an input image filename.

  --boot-image=<file.art>: provide the image file for the boot class path.

  --instruction-set=(arm|arm64|mips|x86|x86_64): for locating the image

  --output=<file> may be used to send the output to a file.

  --dump:raw_mapping_table enables dumping of the mapping table.

  --dump:raw_mapping_table enables dumping of the GC map.

  --no-dump:vmap may be used to disable vmap dumping.

  --no-disassemble may be used to disable disassembly.
```

(Interactive Demo)

Example: Reversing OAT

- In most cases, using DEXTRA (formerly Dexter) may make sense:

```
Usage: dextra [...] _file_
Where: _file_ = DEX or OAT file to open
And [...] can be any combination of:
       -c: Only process this class
       -m: show methods for processed classes (implies -c *)
       -f: show fields for processed classes (implies -c *)
       -p: Only process classes in this package
       -d: Disassemble DEX code sections (like dexdump does - implies -m)
       -D: Decompile to Java (new feature, still working on it. Implies -m)
Or one of:
       -h: Just dump file header
       -M [_index_]: Dump Method at _index_, or dump all methods
       -F [_index_]: Dump Field at _index_, or dump all fields
       -S [_index_]: Dump String at _index_, or dump all strings
       -T [_index_]: Dump Type  at _index_, or dump all types
OAT specific switches:
       -dextract Extract embedded DEX content from an OAT files
And you can always use any of these output Modifiers:
       -j: Java style output (default is JNI, but this is much better)
       -v: verbose output
       -color: Color output (can also set JCOLOR=1 environment variable)
```

(Interactive Demo)

Caveat

- DEXTRA is still a work in progress
  - No disassembly of native/portable code (yet), Just DEX (but with decompilation!)
- Tool MAY Crash – especially on ART files
  - It would help if Google's own oatdump was:
    A) Actually readable code, with C structs instead of C++ serializations!
    B) Actually worked and didn't crash so frequently
- Please use and abuse dextra, and file bug reports
  - Check frequently for updates (current tool version is presently 1.2)
  - http://www.newandroidbook.com/tools/dextra.html

ART Runtime threads

- The runtime uses several worker threads, which it names:

```
# Following the pattern demonstrated to enumerate prctl(2) named threads:
root@generic:/proc/$app_pid/task # for x in *; do grep Name $x/status; done
Name:    android.browser        # Main (UI) thread, last 16 chars of classname
Name:    Signal Catcher         # Intercepts SIGQUIT and SIGUSR1 signals
Name:    JDWP                   # Java Debug Wire Protocol
# Runtime::StartDaemonThreads() calls libcore's java.lang.Daemons for these
Name:    ReferenceQueueD        # Reference Queue Daemon (as in Dalvik)
Name:    FinalizerDaemon        # Finalizer Daemon (as in Dalvik)
Name:    FinalizerWatchd        # Finalizer Watchdog (as in Dalvik)
Name:    HeapTrimmerDaem        # Heap Trimmer
Name:    GCDaemon               # Garbage Collection daemon thread
# Additional Thread Pool Worker threads may be started
...
```

ART Runtime threads

- The Daemon Threads are started in Java, by libcore
  - Daemon class wraps thread class, provides singleton INSTANCE
  - Do same basic operations as they did in “classic” DalvikVm
    - Libart subtree in libcore implementation slightly different

ART Runtime threads

- The Signal Catcher thread responds to SIGQUIT and SIGUSR1:
  - SIGUSR1 forces garbage collection:

runtime/signal_catcher.cc

```
void SignalCatcher::HandleSigUsr1() {
  LOG(INFO) << "SIGUSR1 forcing GC (no HPROF)";
  Runtime::Current()->GetHeap()->CollectGarbage(false);
}
```

  - And outputs to the Android logs as I/art with the PID signaled:

```
I/art     (  806): Thread[2,tid=812,WaitingInMainSignalCatcherLoop,Thread*=0xaee9d400,
                   peer=0x12c00080, "Signal Catcher"]: reacting to signal 10
I/art     (  806): SIGUSR1 forcing GC (no HPROF)
I/art     (  806): Explicit concurrent mark sweep GC freed 16(1088B) AllocSpace objects,
                   0(0B) LOS objects, 63% free, 297KB/809KB, paused 745us total 238.066msss
```

  - SIGQUIT doesn't actually quit, but dumps statistics to /data/anr/traces.txt
    - Statistics are appended, so it's a bad idea to delete the file while system is running

ART Statistics

/data/anr/traces.txt

```
----- pid ... at 2014-11-17 20:22:55 -----
Cmd line: com.android.dialer
ABI: arm                        # 32-bit ARMv7 architecture
Build type: optimized
Loaded classes: 3596 allocated classes
Intern table: 4639 strong; 239 weak
JNI: CheckJNI is on; globals=246
Libraries: ...    # List of native runtime libraries from /system/lib (possibly vendor)
Heap: 63% free, currentKB/maxKB; number objects
Dumping cumulative Gc timings
Start Dumping histograms for 247 iterations for concurrent mark sweep
... Detailed garbage collection histograms
Done Dumping histograms
Total time spent in GC: 31.345s
Mean GC size throughput: 831KB/s
Mean GC object throughput: 3366.85 objects/s
Total number of allocations 142890
Total bytes allocated 25MB
Free memory 512KB
Free memory until GC 512KB
Free memory until OOME 63MB
Total memory 807KB
Max memory 64MB
Total mutator paused time: 625.069ms
Total time waiting for GC to complete: 37.614ms
```

ART Statistics

/data/anr/traces.txt

```
DALVIK THREADS (##):
"main" prio=5 tid=1 Native   # Native, Waiting, or Runnable
  | group="main" sCount=1 dsCount=0 obj=0x7485b970 self=0xb5007800
  | sysTid=806 nice=0 cgrp=apps/bg_non_interactive sched=0/0 handle=0xb6f5fec8
  | state=S schedstat=( 260000000 14200000000 134 ) utm=10 stm=16 core=0 HZ=100
  | stack=0xbe4e4000-0xbe4e6000 stackSize=8MB
  | held mutexes=
  kernel: sys_epoll_wait+0x1d4/0x3a0       #   (wchan)
  kernel: sys_epoll_pwait+0xac/0x13c       #   (system call invoked)   <------+
  kernel: ret_fast_syscall+0x0/0x30        #   (entry point)                  |
  native: #00 pc 00039ed8  /system/lib/libc.so (__epoll_pwait+20)      -------+
  native: #01 pc 00013abb  /system/lib/libc.so (epoll_pwait+26)
  native: #02 pc 00013ac9  /system/lib/libc.so (epoll_wait+6)

  # Managed stack frames (if any) follow (from Java's printStackTrace())
  at android.os.MessageQueue.nativePollOnce(Native method)
  at android.os.MessageQueue.next(MessageQueue.java:143)
  at android.os.Looper.loop(Looper.java:122)
  at android.app.ActivityThread.main(ActivityThread.java:5221)
  at java.lang.reflect.Method.invoke!(Native method)
  at java.lang.reflect.Method.invoke(Method.java:372)
  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:899)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:694)

   ... (for as many as ## threads, above)
```

ART Memory Allocation

- ART has not one, but two underlying allocators:
  - DLMalloc: The traditional libc allocator, from Bionic
    - Not optimized for threads (uses a global memory lock)
    - Inter-thread conflicts arise, as do potential collisions with GC
  - ROSalloc: Runs-of-Slots-Allocator (art/runtime/gc/allocator/rosalloc.h)
    - Allows thread-local-storage region for reasonably small objects
      - Separate Thread Local bit map used, which GC can access with no lock
    - Supports “Bulk Free”:
      - GC first marks slots to free (with no lock)
      - Bulk free operation uses one lock, and frees all slots with indicated bits
    - Larger objects can be locked independently of others

ART Garbage Collection

- ART uses not one, but two Garbage Collectors:
  - The Foreground collector
  - The Background collector
- There are also no less than eight garbage collection algorithms:

Mark/Sweep
Concurrent Mark/Sweep
Semi-Space, Mark/Sweep
Generation Semi-Space
Mark Compact Collector
Heap Trimming Collector
Concurrent Copying Collector
Homogenous Space Compactor

Takeaways

- ART is a far more advanced runtime architecture
  - Brings Android closer to iOS native level performance (think: Objective-C*)
- Vestiges of DEX still remain, to haunt performance
  - DEX Code is still 32-bit
- Very much still a shifting landscape
  - Internal structures keep on changing – Google isn't afraid to break compatibility
  - LLVM integration likely to only increase and improve
- For most users, the change is smooth:
  - Better performance and power consumption
  - Negligible cost of binary size increase (and who cares when you have SD?)
  - Minor limitations on DEX obfuscation remain.
  - For optimal performance (and obfuscation) nothing beats JNI...

*** - Unfortunately, iOS is moving away again with SWIFT and METAL both offering significant performance boosts over OBJ-C**