Dalvík and ART

Jonathan Levin

http://NewAndroidBook.com/

http://www.technologeeks.com/

Wait.. Isn't Android all ART now?

- Well.. Yes, and no.. The actual runtime **is** ART, but..
  - Your applications still compile into Dalvik (DEX) code
  - Final compilation to ART occurs on the device, during install
  - Even ART binaries have Dalvik embedded in them
  - Some methods may be left as DEX, to be interpreted
  - Dalvik is *much* easier to debug than ART.

What we **won't** be discussing

- Dalvik VM runtime architecture*
  - Mostly replaced by ART, prominent features removed
  - No talk about JIT (ART does AOT)
  - No JNI

- Dalvik specific debug settings
  - Not really relevant anymore, either

*** - We discuss these aspects later on, in the context of ART – but that's part II**

What we **will** be discussing

- DEX file structure
- DEX code generation
- DEX verification and optimization
- DEX decompilation and reverse engineering

The Book

“Android Internals: A Confectioner’s Cookbook”

- **深入解析Android 操作系统 - Coming in Chinese (by end of 2016)**
- Volume I (Available now): Power User’s view
- Volume II (Available once N is out, and ART is final!): Developer’s View
- http://NewAndroidBook.com/TOC.html for detailed Table of Contents

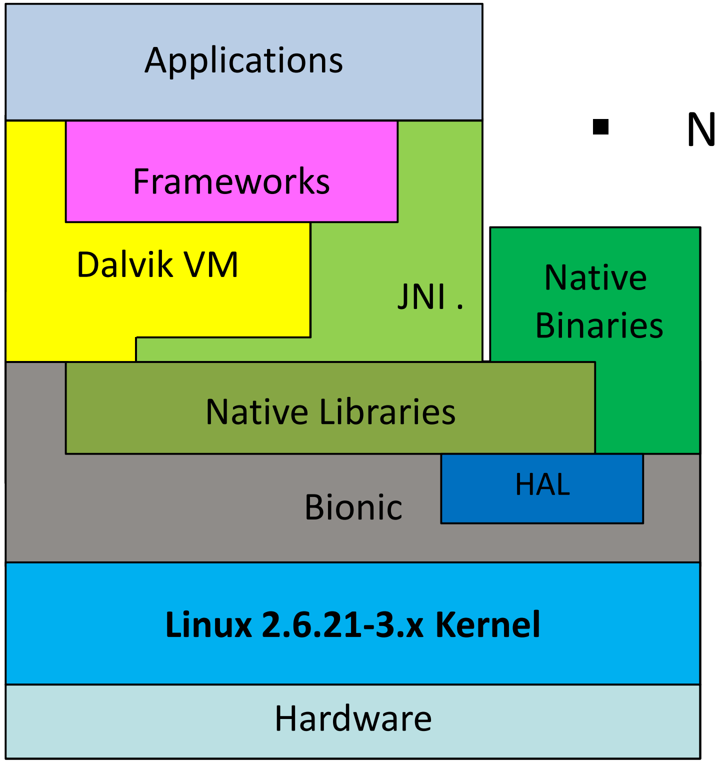
- Unofficial sequel to Karim Yaghmour’s “Embedded Android”, different focus:
  - More on the **how** and **why** Android frameworks and services work
  - More on DEX and ART (this talk is an excerpt from Volume II)
  - (presently) only in-depth book on the subject

- http://www.NewAndroidBook.com/ :
  - **Free** and **powerful** tools
  - Articles and bonus materials from Books

- Android Internals & Reverse Engineering: Feb 8th-12th, 2016, NYC
  - http://Technologeeks.com/AIRE

Part I - Dalvík

Dalvík and the Android Architecture



The Dalvík Virtual Machine* is:

- Customized, optimized JVM
  - Based on Apache "Harmony" JVM
- Not fully J2SE or J2ME compatible
  - Java compiles into DEX code
    - 16-bit opcodes
    - Register, rather than stack-based

*** - Android L replaces Dalvik by the Android RunTime – but does not get rid of it fully (more later)**

A Brief History of Dalvík

- Dalvík was introduced along with Android
  - Created by Dan Bornstein
  - Named after an Icelandic town

- 2.2 (Froyo) brought Just-in-Time compilation

- 4.4 (KitKat) previews ART

- 5.0 (Lollipop) ART supersedes.
  - DEX is still alive and well, thank you for asking

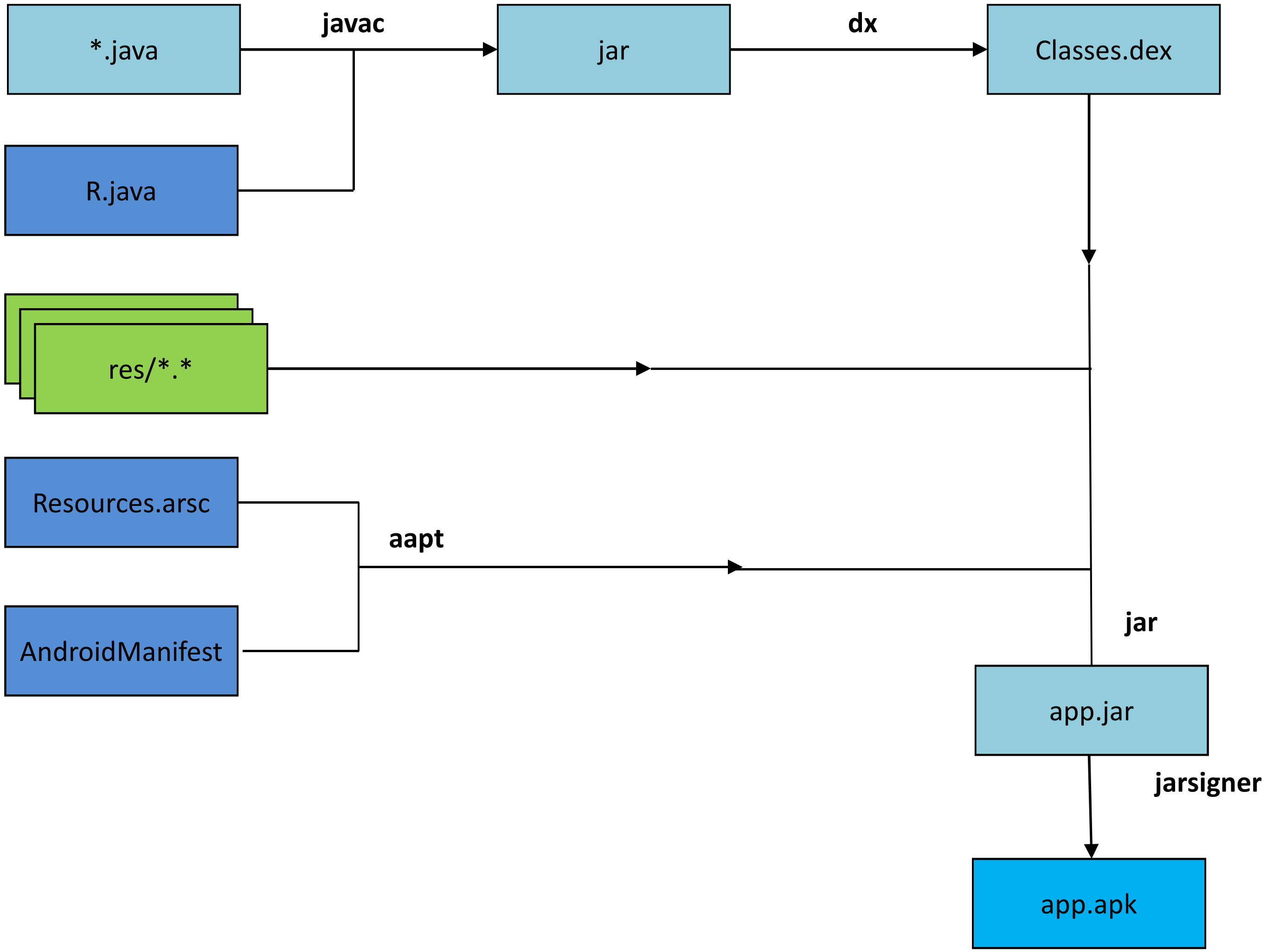
Dalvik, Iceland (photo by the author)

Dalvik VM vs. Java

- Dalvík is a virtual machine implementation
  - Based on Apache Harmony
  - Borrows heavily from Java*
- Brings significant improvements over Java, in particular J2ME:
  - Virtual Machine architecture is optimized for memory sharing
    - Reference counts/bitmaps stored separately from objects
    - Dalvik VM startup is optimized through Zygote
- Java .class files are further compiled into DEX.

*** - So heavily, in fact, that Oracle still carries Sun’s grudge against Google**

Reminder: Creating an APK



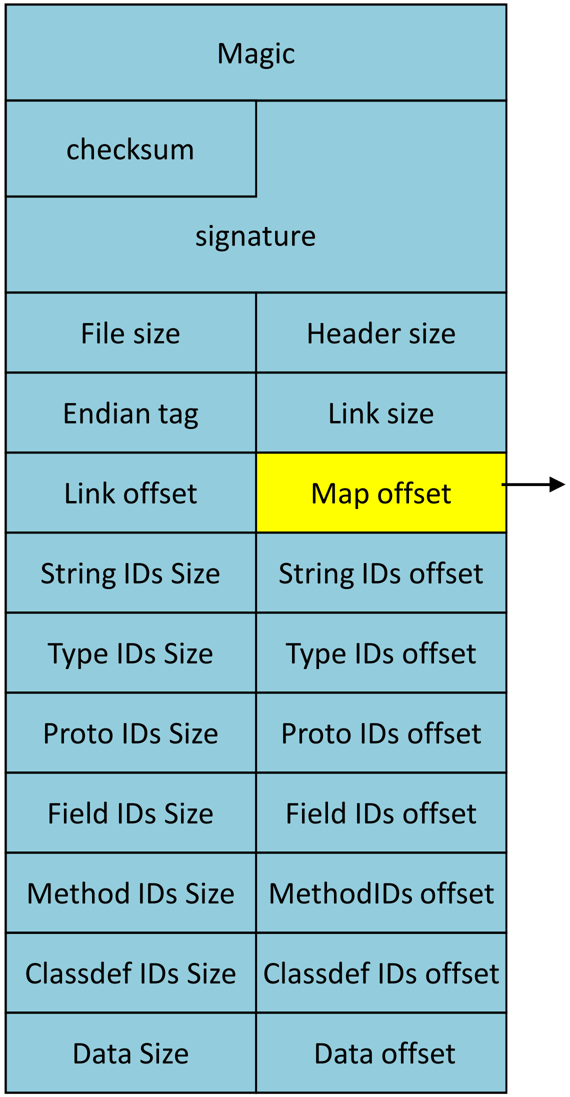
The DEX file format

- The “dx” utility converts multiple .class files to classes.dex
  - Script wrapper over `java -Xmx1024M -jar ${SDK_ROOT}.../lib/dx.jar`
  - Java byte code is converted to DEX bytecode
    - DEX instructions are 16-bit multiples, as opposed to Java’s 8-bit
  - Constant, String, Type and Method pools can be merged
    - Significant savings for strings, types, and methods in multiple classes
- Overall memory footprint diminished by about 50%
- DEX file format fully specified in Android Documentation

The DEX file format

	Magic		DEX Magic header ("dex\n“ and version (“035 “)
Adler32 of header (from offset +12)	checksum	signature	
	signature		SHA-1 hash of file (20 bytes)
Total file size	File size	Header size	Header size (0x70)
0x12345678, in little or big endian form	Endian tag	Link size	Unused (0x0)
Unused (0x0)	Link offset	Map offset	Location of file map
Number of String entries	String IDs Size	String IDs offset	
Number of Type definition entries	Type IDs Size	Type IDs offset	
Number of prototype (signature) entries)	Proto IDs Size	Proto IDs offset	
Number of field ID entries	Field IDs Size	Field IDs offset	
Number of method ID entries	Method IDs Size	MethodIDs offset	
Number of Class Definition entries	Classdef IDs Size	Classdef IDs offset	
Data (map + rest of file)	Data Size	Data offset	

The DEX file format



Magic

checksum

signature

File size	Header size
Endian tag	Link size
Link offset	Map offset
String IDs Size	String IDs offset
Type IDs Size	Type IDs offset
Proto IDs Size	Proto IDs offset
Field IDs Size	Field IDs offset
Method IDs Size	MethodIDs offset
Classdef IDs Size	Classdef IDs offset
Data Size	Data offset

Type	Implies	Size	Offset
0x0	DEX Header	1 (implies Header Size)	0x0
0x1	String ID Pool	Same as String IDs size	Same as String IDs offset
0x2	Type ID Pool	Same as Type IDs size	Same as String IDs offset
0x3	Prototype ID Pool	Same as Proto IDs size	Same as ProtoIDs offset
0x4	Field ID Pool	Same as Field IDs size	Same as Field IDs offset
0x5	Method ID Pool	Same as Method IDs size	Same as Method IDs offset
0x6	Class Defs	Same as ClassDef IDs size	Same as ClassDef IDs offset
0x1000	Map List	1	Same as Map offset
0x1001	Type List	List of type indexes (from Type ID Pool)	
0x1002 0x1003	Annotation set Annotation Ref	Used by Class, method and field annotations	
0x2000	Class Data Item	For each class def, class/instance methods and fields	
0x2001	Code	DexCodeItems – contains the actual byte code	
0x2002	String Data	Pointers to actual string data	
0x2003	Debug Information	Debug_info_items containing line no and variable data)	
0x2004	Annotation	Field and Method annotations	
0x2005	Encoded Array	Used by static values	
0x2006	Annotations Directory	Annotations referenced from individual classdefs	

Looking up classes, methods, etc.

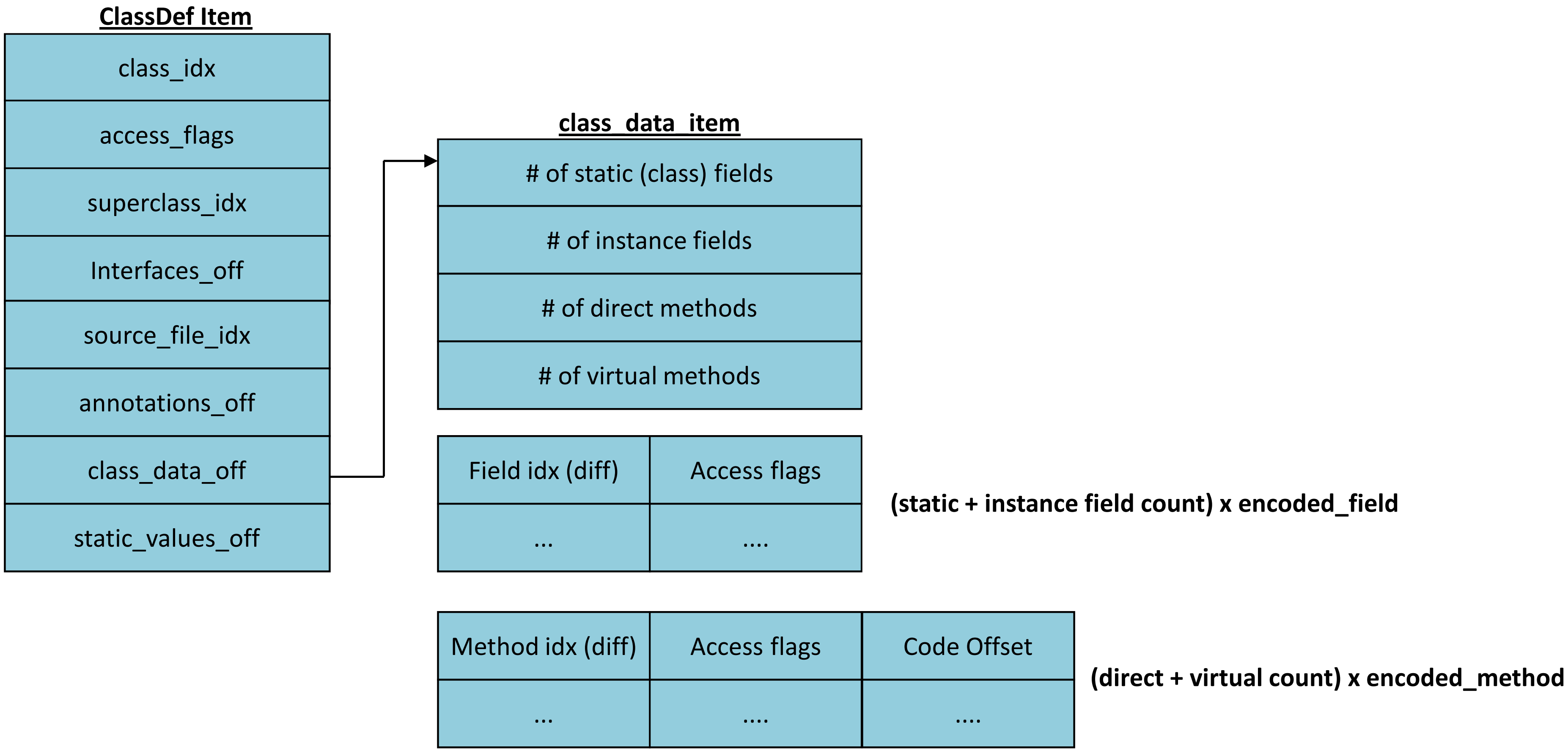
- Internally, DEX instructions refer to Indexes (in pools)
- To find a method:
  - DexHeader’s Method IDs offset points to an array of MethodIDs
  - Each method ID points to a class index, prototype index and method name
- To find a field:
  - DexHeader’s Field Ids offset points to an array of FieldIDs
  - Each Field ID points to a class index, type index, and the field name
- To get a class:
  - DexHeader’s Class Defs Ids offset points to an array of ClassDefs
  - Each ClassDef points to superclass, interface, and class_data_item
  - Class_data_item shows # of static/instance fields, direct/virtual methods
  - Class_data_item is followed by DexField[], DexMethod[] arrays
    - DexField, DexMethod point to respective indexes, as well as class specific access flags

Finding a class’s method code

class_idx	**Index of the class’ type id, from Type ID pool**
access_flags	**ACC_PUBLIC, _PRIVATE, _PROTECTED, _STATIC, _FINAL, etc. Etc..**
superclass_idx	**Index of the superclass’ type id, from Type ID pool**
Interfaces_off	**Offset of type_list containing this class’ implemented interface, if any**
source_file_idx	**Index of the source file name, in String pool**
annotations_off	**Offset of an annotations_directory_item for this class**
class_data_off	**Offset of this class’s class_data_item**
static_values_off	**Offset to initial values of any fields defined as static (i.e. Class)**

access_flags and **static_values_off** particulary useful for fuzzing/patching classes

Finding a class's method code (II)



ClassDef Item

- class_idx
- access_flags
- superclass_idx
- Interfaces_off
- source_file_idx
- annotations_off
- class_data_off
- static_values_off

class_data_item

- # of static (class) fields
- # of instance fields
- # of direct methods
- # of virtual methods

Field idx (diff)	Access flags
...	....

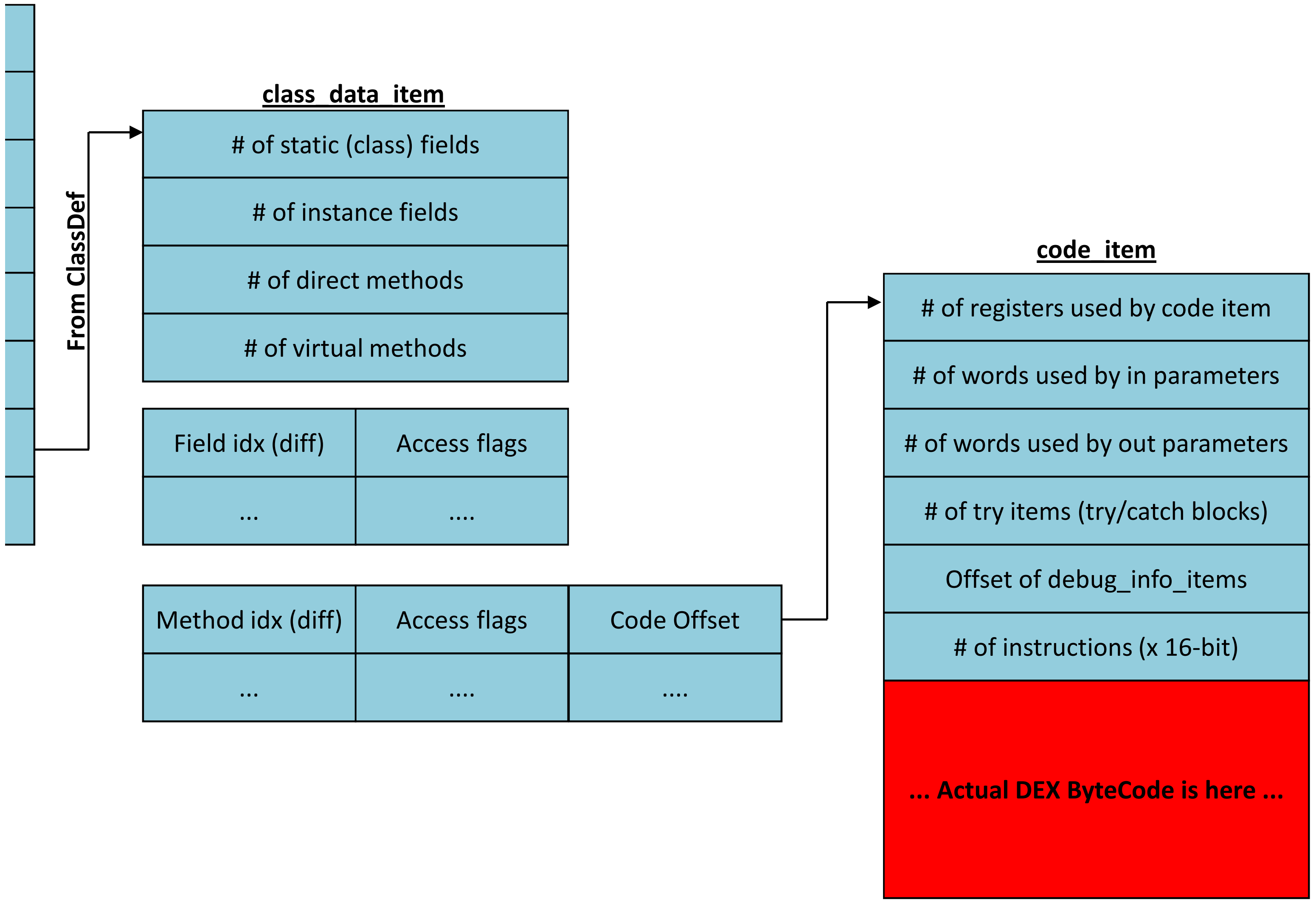
(static + instance field count) x encoded_field

Method idx (diff)	Access flags	Code Offset
...	....	....

(direct + virtual count) x encoded_method

Class_data_item fields are all ULEB128 encoded (*sigh*)

Finding a class’s method code (III)



class_data_item

# of static (class) fields

# of instance fields

# of direct methods

# of virtual methods

Field idx (diff)	Access flags
…	….

Method idx (diff)	Access flags	Code Offset
…	….	….

code_item

# of registers used by code item

# of words used by in parameters

# of words used by out parameters

# of try items (try/catch blocks)

Offset of debug_info_items

# of instructions (x 16-bit)

… Actual DEX ByteCode is here …

The DEX Bytecode

- The Android Documentation is good, but lacking
  - Bytecode instruction set
  - Instruction formats

- No documentation on optimized code
  - ODEX codes (used in 0xE3-0xFF) are simply marked as “unused”

- Not yet updated to reflect ART DEX changes (still undocumented)
  - DEX opcode 0x73 claimed by return-void-barrier
  - ODEX codes 0xF2-0xFA are moved to 0xE3-0xEB. 0xEC-0xFF now unused

The DEX Bytecode

- VM Architecture allows for up to 64k registers
  - In practice, less than 16 are actively used
- Bytecode is method, field, type and string aware
  - Operands in specific instructions are IDs from corresponding pools
- Bytecode is also primitive type-aware
  - Instructions support casting, as well as specific primitive types
- DEX bytecode is strikingly similar to Java bytecode
  - Allows for easy de/re-compilation back and forth to/from java

DEX vs. Java

- Java VM is stack based, DEX is register based
  - Operations in JVM use stack and r0-r3; Dalvik uses v0-v65535
  - Stack based operations have direct register-base parallels
  - Not using the stack (= RAM, via L1/L2 caches) makes DEX somewhat faster.

- Java Bytecode is actually more compact than DEX
  - Java instructions take 1-5 bytes, DEX take 2-10 bytes (in 2-byte multiples)

- DEX bytecode is more suited to ARM architectures
  - Straightforward mapping from DEX registers to ARM registers

- DEX supports bytecode optimizations, whereas Java doesn't
  - APK's classes.dex are optimized before install, on device (more later)

DEX vs. Java Bytecode

Class, Method and Field operators

DEX Opcode	Java Bytecode	Purpose
60-66:sget-* 52-58:iget-*	b2:getstatic b4:getfield	Read a static or instance variable
67-6d:sput 59-5f:iput	b3:putstatic b5:putfield	Write a static or instance variable
6e: invoke-virtual 6f: invoke-super 70: invoke-direct 71: invoke-static 72: invoke-interface	b6: Invokevirtual ba: invokedynamic b7: invokespecial b8: Invokestatic b9: Invokeinterface	Call a method
20: instance-of	c1: instanceof	Return true if obj is of class
1f: check-cast	c0: checkcast	Check if a type cast can be performed
bb:new	22: new-instance	New (unconstructed) instance of object

DEX vs. Java Bytecode

Flow Control instructions

DEX Opcode	Java Bytecode	Purpose
32..37: if-* 38..3d: if-*z	a0-a6: if_icmp* 99-9e: if*	Branch on logical
2b: packed-switch	ab: lookupswitch	Switch statement,
2c: sparse-switch	aa: tableswitch	Switch statement
28: goto 29: goto/16 30: goto/32	a7: goto c8: goto_w	Jump to offset in code
27: throw	bf:athrow	Throw exception

DEX vs. Java Bytecode

Data Instructions

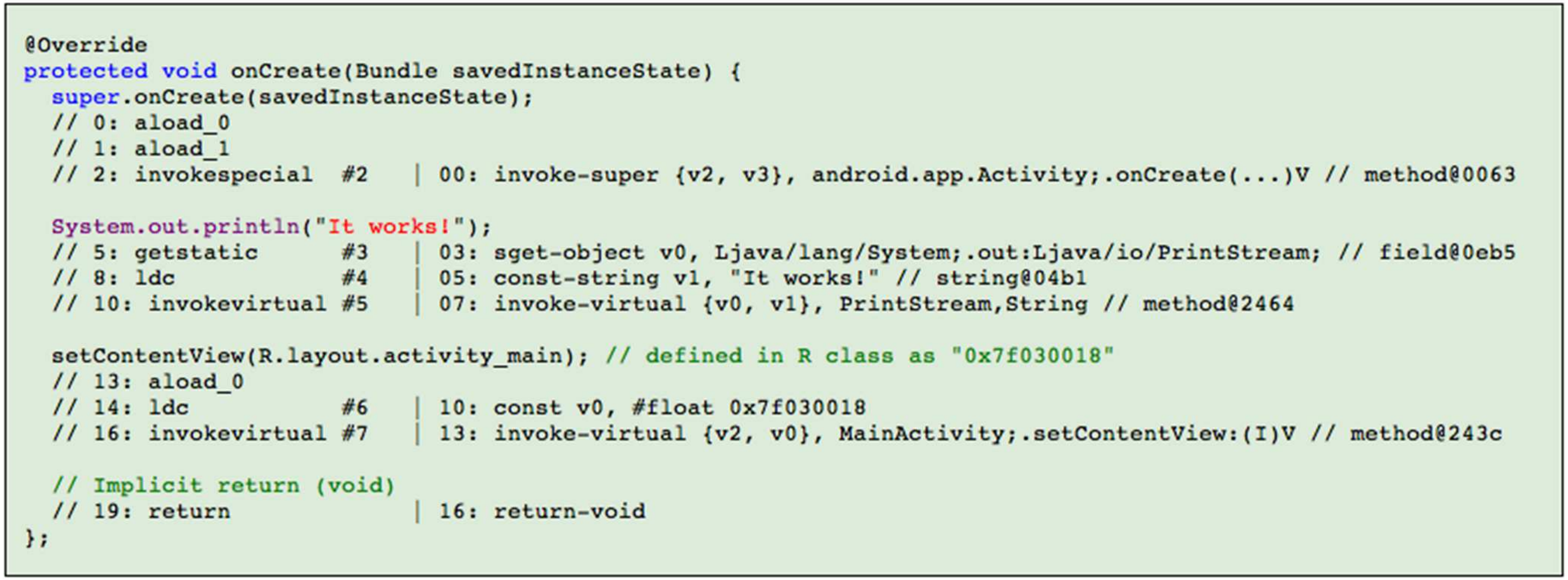
DEX Opcode	Java Bytecode	Purpose
12-1c: const*	12:ldc 13: ldc_w 14: ldc2_w	Define Constant
21: array-length	be: arraylength	Get length of an array
23: new-array	bd: anewarray	Instantiate an array
24-25: filled-new-array[/range] 26: fill-array-data	N/A	Populate an array

Arithmetic instructions are, likewise, highly similar

DEX vs. Java Bytecode

- Example: A “Hello World” activity:

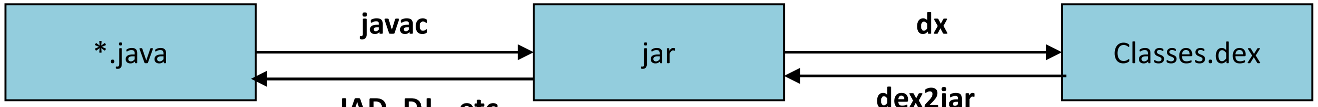
Listing d-dec: Demonstrating Java source, class and DEX bytecode



```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  // 0: aload_0
  // 1: aload_1
  // 2: invokespecial  #2   | 00: invoke-super {v2, v3}, android.app.Activity;.onCreate(...)V // method@0063

  System.out.println("It works!");
  // 5: getstatic      #3   | 03: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@0eb5
  // 8: ldc            #4   | 05: const-string v1, "It works!" // string@04b1
  // 10: invokevirtual #5   | 07: invoke-virtual {v0, v1}, PrintStream,String // method@2464

  setContentView(R.layout.activity_main); // defined in R class as "0x7f030018"
  // 13: aload_0
  // 14: ldc           #6   | 10: const v0, #float 0x7f030018
  // 16: invokevirtual #7   | 13: invoke-virtual {v2, v0}, MainActivity;.setContentView:(I)V // method@243c

  // Implicit return (void)
  // 19: return             | 16: return-void
};
```

DEX to Java

- It comes as no surprise that .dex and .class are isomorphic
- DEX debug items map DEX offsets to Java line numbers
- Dex2jar tool can easily “decompile” from .dex back to a .jar
- Standard Practice:


*.java

javac

JAD, DJ-, etc

jar

dx

dex2jar

Classes.dex

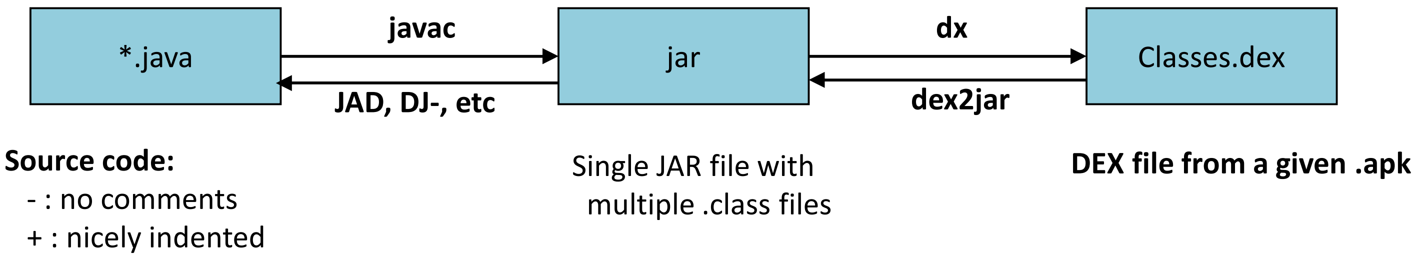
Source code:
- : no comments
+ : nicely indented

Single JAR file with
multiple .class files

DEX file from a given .apk

- Extremely useful for reverse engineering
  - Even more so useful for malice and mischief

DEX to Java



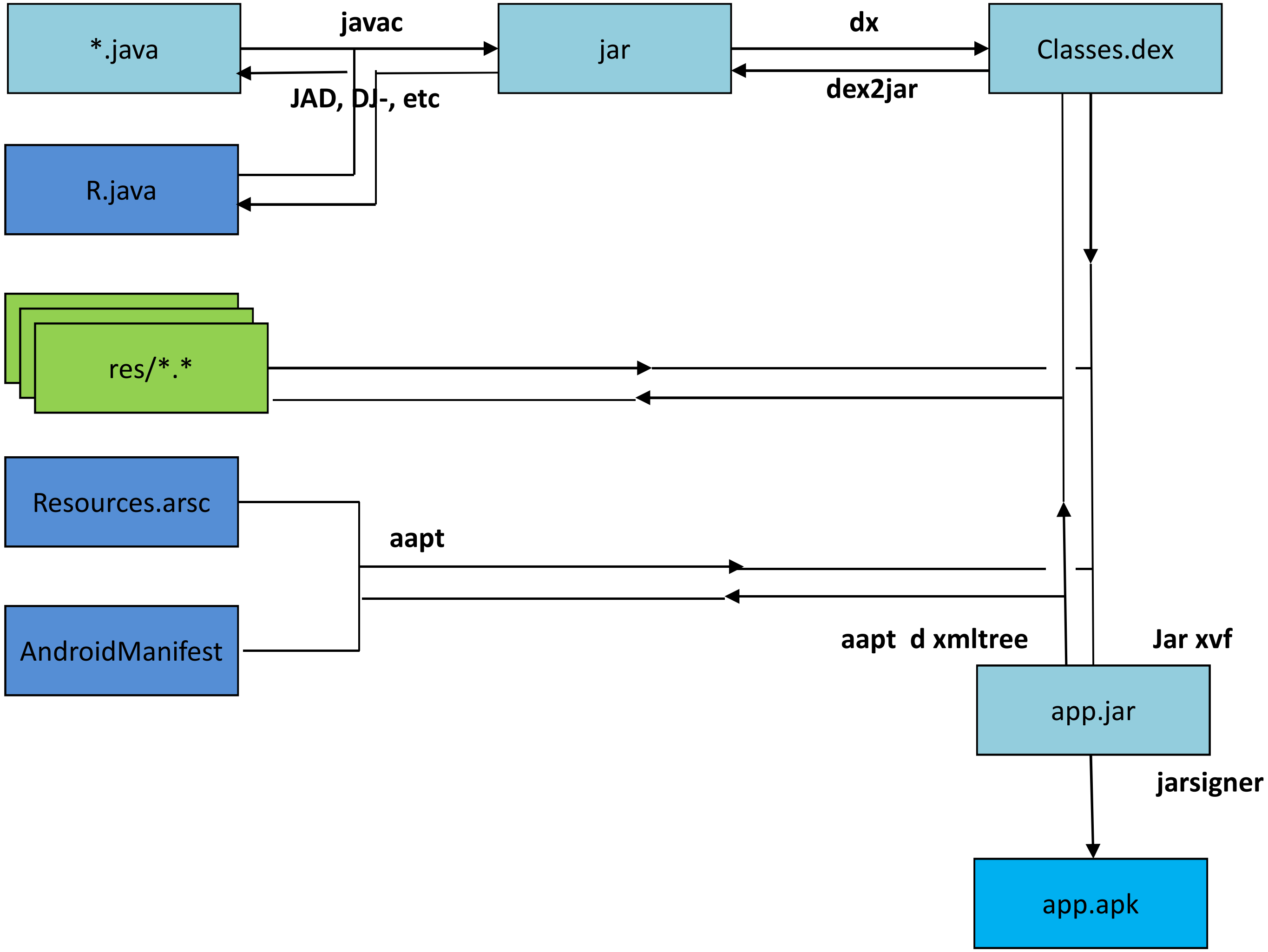
Source code:
- : no comments
+ : nicely indented

Single JAR file with
multiple .class files

DEX file from a given .apk

- Flow from DEX to JAVA is **bidirectional**, meaning that an attacker can:
  - Decompile your code back to Java
  - Remove annoyances like ads, registration
  - Uncover sensitive data (app logic, or poorly guarded secrets)
  - Replace certain classes with others, potentially malicious ones
  - Recompile back to JAR, then DEX
  - Put cloned/trojaned version of your app on Play or another market
- ASEC/OBB “solutions” for this fail miserably when target device is rooted.

Deconstructing an APK



*.java

javac

jar

dx

Classes.dex

JAD, DJ-, etc

dex2jar

R.java

res/*.*

Resources.arsc

aapt

AndroidManifest

aapt d xmltree

Jar xvf

app.jar

jarsigner

app.apk

DEX Obfuscation

- Quite a few DEX "obfuscators" exist, with different approaches:
  - Functionally similar to binutils' `strip`, either java (ProGuard) or sDEX
    - Rename methods, field and class names
    - Break down string operations so as to "chop" hard-coded strings, or encrypt
    - Can use dynamic class loading (DexLoader classes) to impede static analysis
  - Can add dead code and dummy loops (at minor impact to performance)
  - Can also use goto into other instructions (or switches, e.g. DexLABS)
- In practice, quite limited, due to:
  - Reliance on Android Framework APIs (which remain unobfuscated)
  - JDWP and application debuggability at the Java level
  - If Dalvik can execute it, so can a proper analysis tool (e.g. IDA, dextra)
  - Popular enough obfuscators (e.g. DexGuard) have de-obfuscators...
- ... Which is why JNI is so popular

DEX Optimization (dexopt)

- Pre-5.0, installd runs `dexopt` on APK, during installation
  - Extracts the APK’s classes.dex
  - Performs runtime verification and optimization
  - Plops optimized DEX file in /data/dalvik-cache

```
root@android:/data/dalvik-cache # ls -s
total 28547
24 system@app@ApplicationsProvider.apk@classes.dex
1359 system@app@Browser.apk@classes.dex
958 system@app@Contacts.apk@classes.dex
625 system@app@ContactsProvider.apk@classes.dex
99 system@app@DeskClock.apk@classes.dex
795 system@app@DownloadProvider.apk@classes.dex
13 system@app@DrmProvider.apk@classes.dex
...
root@android# file system\@app\@LatinIME.apk\@classes.dex
system@app@LatinIME.apk@classes.dex: Dalvik dex file (optimized for host) version 036
```

- **ART still optimizes DEX**, but uses `dex2oat` instead (q.v. Part II)
  - ODEX files are actually now OAT files (ELF shared objects)
  - Actual DEX payload buried deep inside

DEX Optimization (dexopt)

- `dexopt` is user-friendly ... But only for the right user (`installd`)

```
shell@hammerhead:/ $ dexopt
Usage:

Short version: Don't use this.

Slightly longer version: This system-internal tool is used to
produce optimized dex files. See the source code for details.
```

- The program runs a Dalvik VM with special switches

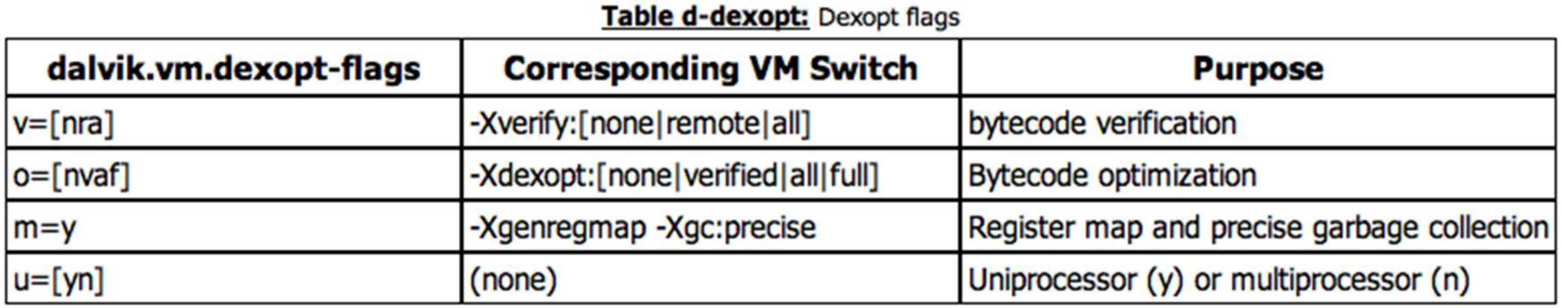


Table d-dexopt: Dexopt flags

dalvik.vm.dexopt-flags	Corresponding VM Switch	Purpose
v=[nra]	-Xverify:[none|remote|all]	bytecode verification
o=[nvaf]	-Xdexopt:[none|verified|all|full]	Bytecode optimization
m=y	-Xgenregmap -Xgc:precise	Register map and precise garbage collection
u=[yn]	(none)	Uniprocessor (y) or multiprocessor (n)

DEX Optimization (dexopt)

- What happens during optimization?
  - Bytecode verification: Deducing code paths, register mapss, and precise GC
  - Wrapping with ODEX header (for optimized data/dependency tables)
  - Opcodes replaced by quick opcode variants*

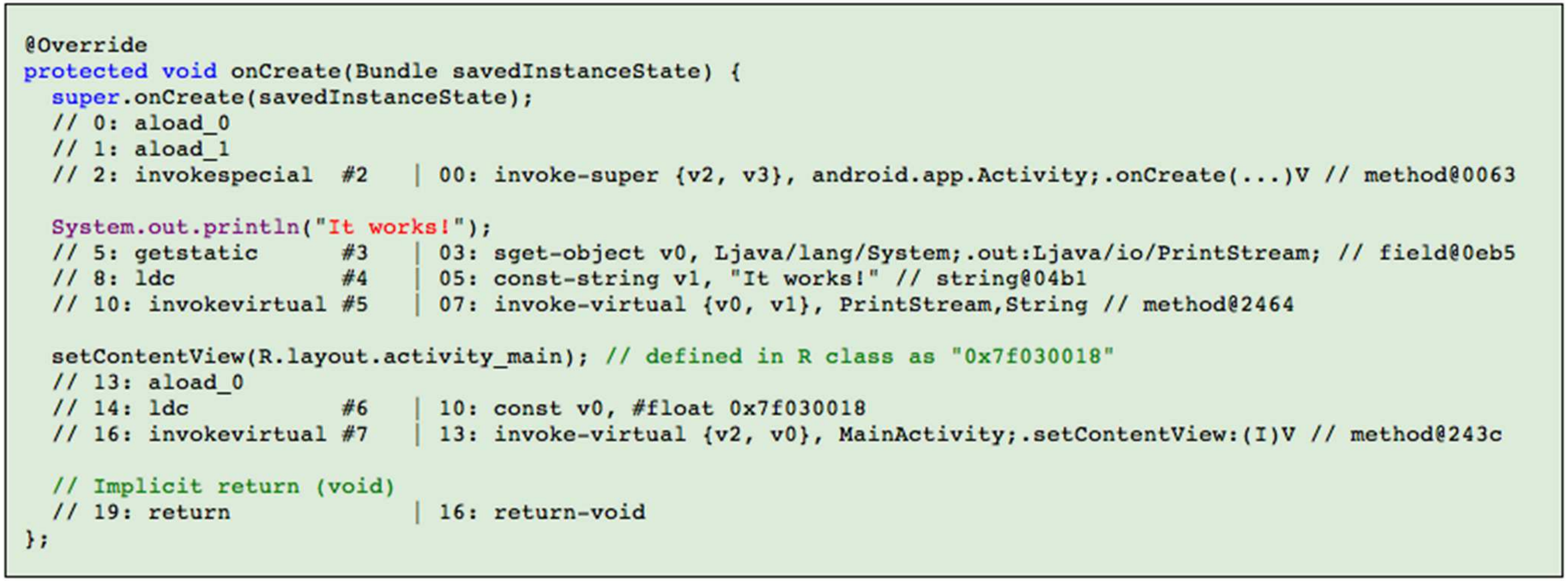
art/compiler/dex/dex_to_dex_compiler.cc

DEX Opcode	ODEX Opcode	Optimization
0e: return-void	73: return-void-barrier	Barrier (in constructors)
52:iget	e3: iget-quick	Use byte offset for field, eliminating costly lookup, and merge primitive datatypes into a 32-bit (wide) instruction, reducing overall code size.
53: iget-wide	e4: iget-wide-quick	
54: iget-object	e5:iget-object-quick	
59: iput	e6: iput-quick	
5a: iput-wide	e7: iput-wide-quick	
5b: iput-object	e8: iput-object-quick	
6e: invoke-virtual	e9/ea: invoke-virtual-quick[/range]	Vtable, eliminating lookup

*** - Pre-ART optimization also added execute-inline, as well as –volatile variants for iget/iput – but those have been removed**

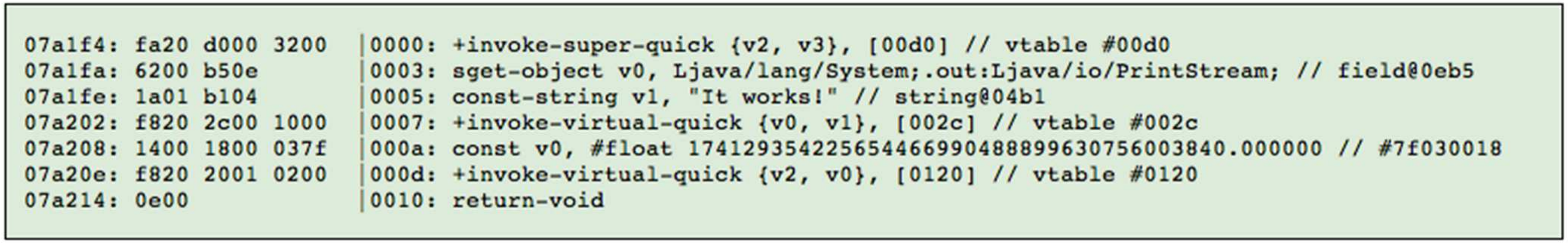
DEX Optimization (dexopt)

Listing d-dec: Demonstrating Java source, class and DEX bytecode



```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  // 0: aload_0
  // 1: aload_1
  // 2: invokespecial  #2   | 00: invoke-super {v2, v3}, android.app.Activity;.onCreate(...)V // method@0063

  System.out.println("It works!");
  // 5: getstatic      #3   | 03: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@0eb5
  // 8: ldc            #4   | 05: const-string v1, "It works!" // string@04b1
  // 10: invokevirtual #5   | 07: invoke-virtual {v0, v1}, PrintStream,String // method@2464

  setContentView(R.layout.activity_main); // defined in R class as "0x7f030018"
  // 13: aload_0
  // 14: ldc           #6   | 10: const v0, #float 0x7f030018
  // 16: invokevirtual #7   | 13: invoke-virtual {v2, v0}, MainActivity;.setContentView:(I)V // method@243c

  // Implicit return (void)
  // 19: return             | 16: return-void
};
```

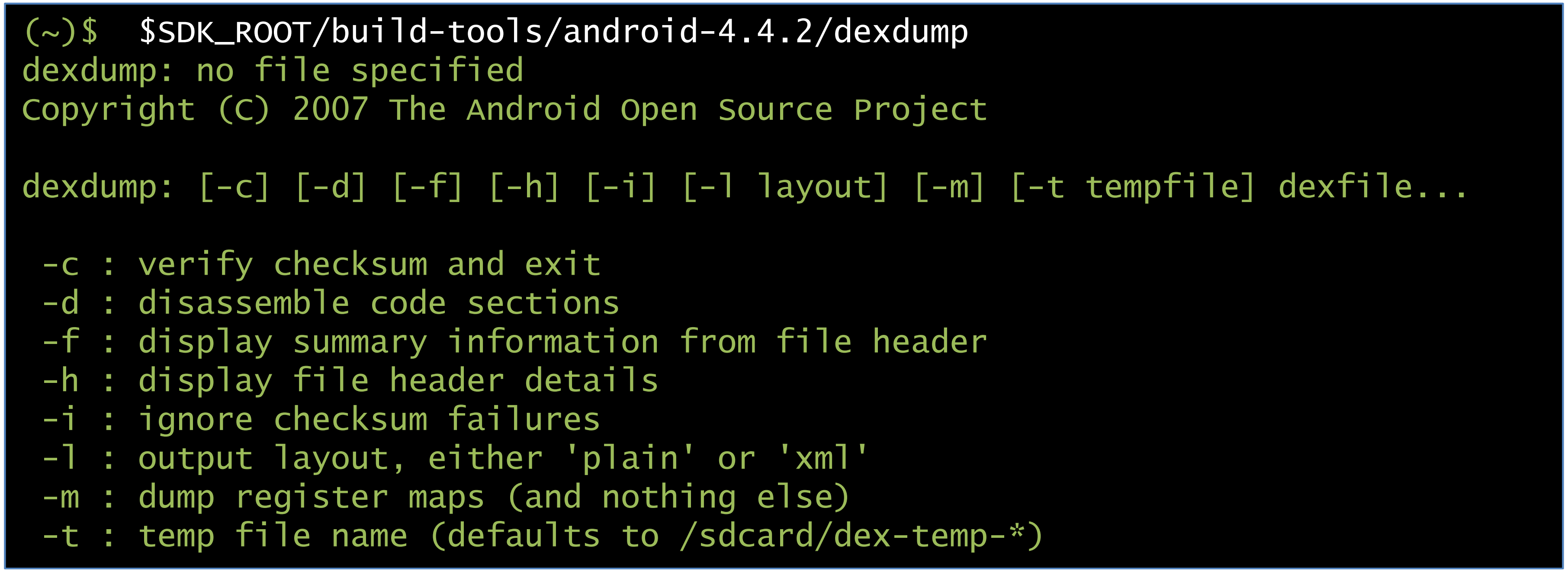
Listing d-optdump: Optimized DEX version of sample App's OnCreate()



```
07a1f4: fa20 d000 3200  |0000: +invoke-super-quick {v2, v3}, [00d0] // vtable #00d0
07a1fa: 6200 b50e       |0003: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@0eb5
07a1fe: 1a01 b104       |0005: const-string v1, "It works!" // string@04b1
07a202: f820 2c00 1000  |0007: +invoke-virtual-quick {v0, v1}, [002c] // vtable #002c
07a208: 1400 1800 037f  |000a: const v0, #float 174129354225654466990488899630756003840.000000 // #7f030018
07a20e: f820 2001 0200  |000d: +invoke-virtual-quick {v2, v0}, [0120] // vtable #0120
07a214: 0e00            |0010: return-void
```

Example: Reversing DEX

- You can use the AOSP-supplied DEXDUMP to disassemble DEX



```
(~)$  $SDK_ROOT/build-tools/android-4.4.2/dexdump
dexdump: no file specified
Copyright (C) 2007 The Android Open Source Project

dexdump: [-c] [-d] [-f] [-h] [-i] [-l layout] [-m] [-t tempfile] dexfile...

 -c : verify checksum and exit
 -d : disassemble code sections
 -f : display summary information from file header
 -h : display file header details
 -i : ignore checksum failures
 -l : output layout, either 'plain' or 'xml'
 -m : dump register maps (and nothing else)
 -t : temp file name (defaults to /sdcard/dex-temp-*)
```

(Interactive Demo)

Example: Reversing DEX

- Alternatively, use DEXTRA (formerly Dexter)

```
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)

Example: Reversing DEX

- Dextra has (for the moment, medium) support for decompilation (working on it)

```
(~)$  JCOLOR=1 dextra -d -D Tests/classes.dex
...
         public class   com.technologeeks.BasicApp.MainActivity
           extends android.app.Activity    {
          public  void <init> () // Constructor
                  {
                  result = android.app.Activity.<init>(v0); // (Method@0)
                  }
 public  void onCreate (android.os.Bundle)
 {
   v0 = java.lang.System.out; // (Field@4)
   v1 = “It Works!\n"; // (String@3)
   result = java.io.PrintStream.println(v0, v1); // (Method@11)
   result = android.app.Activity.onCreate(v2, v3); // (Method@1)
   v0 = 0x7f030018;
   result = com.technologeeks.BasicApp.MainActivity.
                   setContentView(v2, v0); // (Method@5)
 }
}  // end class com.technologeeks.BasicApp.MainActivity
```

(Interactive Demo)

So why is Dalvik deprecated?

- JIT is slow, consuming both cycles and battery power
- Garbage collection (esp. GC_FOR_ALLOC) causes hangs/jitter
- Dalvik VM is 32-bit, and can't benefit from 64-bit architecture
    - And everybody* wants 64-bit, now that iOS has it...
- KitKat was the first to introduce ART, And Lollipop adopts it
    - For more details on ART Internals, stick around for Part II..

*** - Well, maybe everybody except Qualcomm... Or .. On second thought, maybe they do, too?**

Take Away

- DEX is a Dangerous Executable format...
  - Risks to app developers are significant, with no clear solutions
  - (And we haven't even mentioned fun with DEX fuzzing)
  - (And if we do mention fuzzing – Check $AOSP_SRC/art/tools/dexfuzz!)

- DEX isn't DEAD yet – even with ART:
  - Still buried deep inside those OAT files
  - FAR easier to reverse engineer embedded DEX, than do so for OAT

Parts we didn't discuss here are in the book(Volume II)

References

- 2014 - Qualcomm Mobile Security Summit – “ Android App “Protection” “ – “diff”/”case”
- 2015 - Defcon XXIII – “Offsensive & Defensive Android Reverse Engineering” – “diff”/”case”/Fenton

Greets

- Jon Sawyer (“justin case”) - @jcase

Dalvik and ART

Jonathan Levin

http://NewAndroidBook.com/

http://www.technologeeks.com

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)

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
- Constantly evolving, through Marshmallow
  - Major caveat: Often changes in between Android minor versions
  - (Android re-“Optimizes Apps” every time you update)

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**

  not as

  - Virtual machine maintenance is 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
      - Still uses “.odex” extension, now file format is ELF/OAT.

ART files

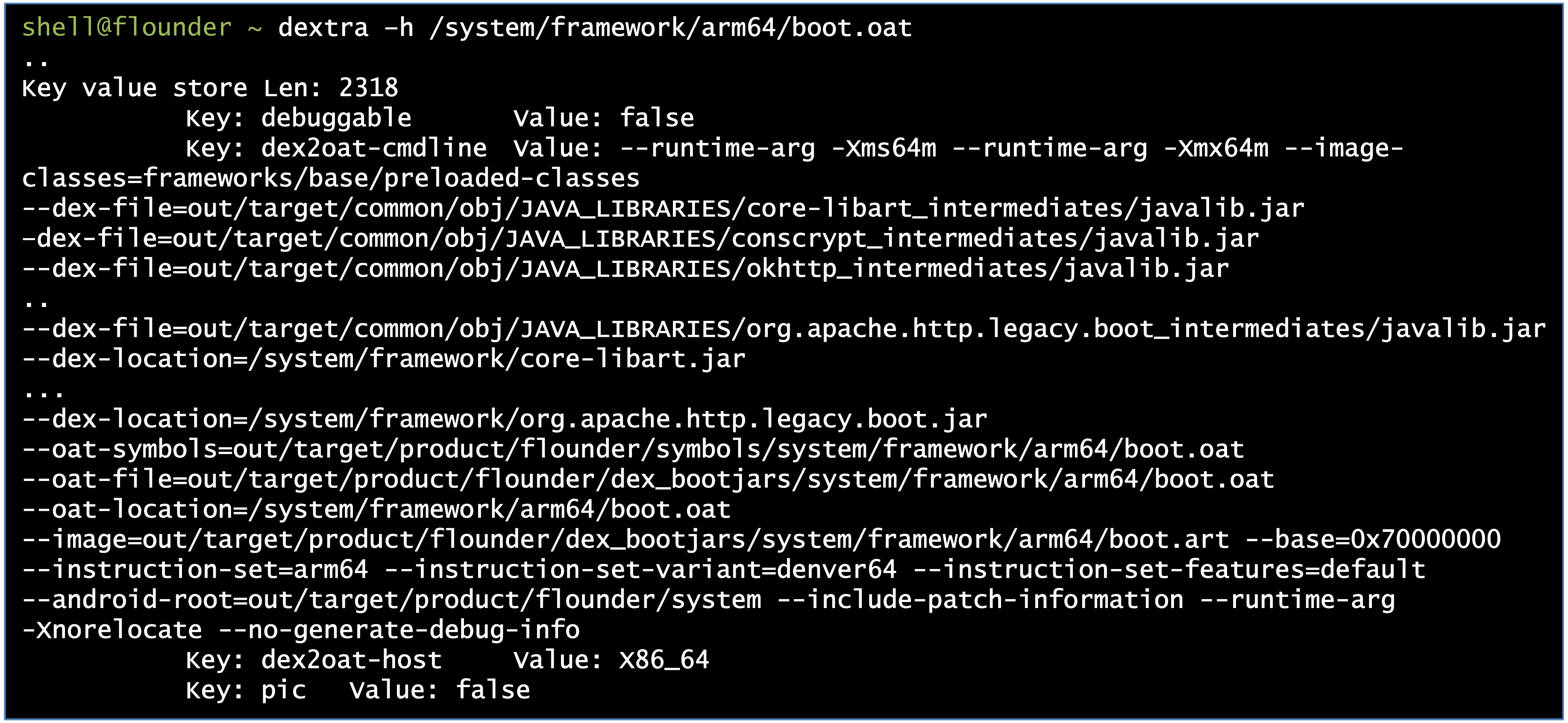
- The ART file is a proprietary format
  - Poorly documented (which is why I wrote the book)
  - changed format internally repeatedly (which is why book is so delayed)
  - Not really understood by oatdump, either.. (which is why I wrote dextra)
  - And.. changed again (from 009 to 017....) (which is why I keep updating it)

- ART file maps in memory right before OAT, which links with it.

- Contains pre-initialized classes, objects, and support structures

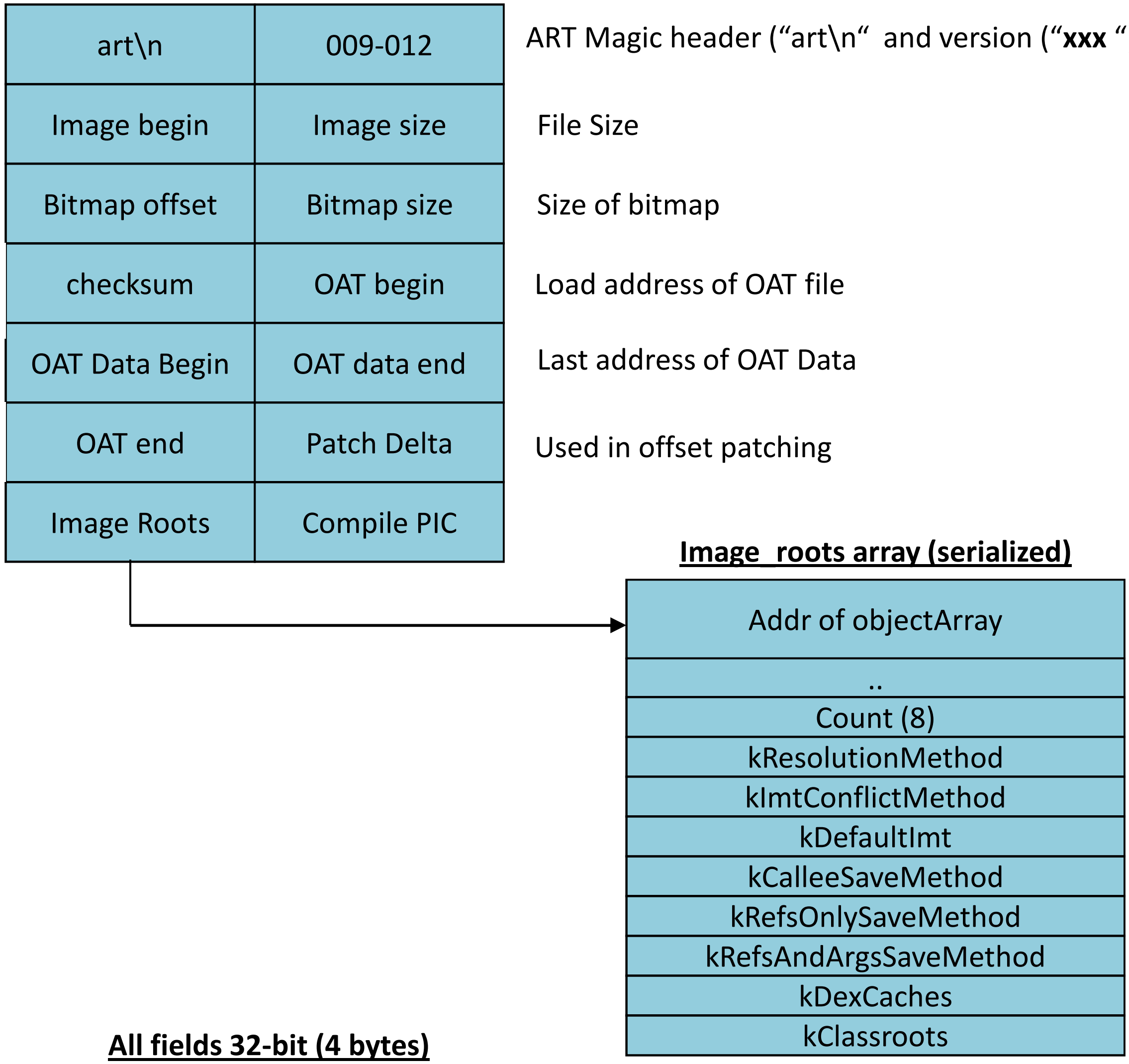
Creating ART (and OAT)

- ART/OAT files are created (on device or on host) by dex2oat
- Command line saved inside OAT file's key value store:



```
shell@flounder ~ dextra -h /system/framework/arm64/boot.oat
..
Key value store Len: 2318
           Key: debuggable      Value: false
           Key: dex2oat-cmdline  Value: --runtime-arg -Xms64m --runtime-arg -Xmx64m --image-
classes=frameworks/base/preloaded-classes
--dex-file=out/target/common/obj/JAVA_LIBRARIES/core-libart_intermediates/javalib.jar
-dex-file=out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermediates/javalib.jar
--dex-file=out/target/common/obj/JAVA_LIBRARIES/okhttp_intermediates/javalib.jar
..
--dex-file=out/target/common/obj/JAVA_LIBRARIES/org.apache.http.legacy.boot_intermediates/javalib.jar
--dex-location=/system/framework/core-libart.jar
...
--dex-location=/system/framework/org.apache.http.legacy.boot.jar
--oat-symbols=out/target/product/flounder/symbols/system/framework/arm64/boot.oat
--oat-file=out/target/product/flounder/dex_bootjars/system/framework/arm64/boot.oat
--oat-location=/system/framework/arm64/boot.oat
--image=out/target/product/flounder/dex_bootjars/system/framework/arm64/boot.art --base=0x70000000
--instruction-set=arm64 --instruction-set-variant=denver64 --instruction-set-features=default
--android-root=out/target/product/flounder/system --include-patch-information --runtime-arg
-Xnorelocate --no-generate-debug-info
           Key: dex2oat-host    Value: X86_64
           Key: pic   Value: false
```

The ART file format



	art\n	009-012	ART Magic header (“art\n“ and version (“**xxx** “)
Load Address of ART file (fixed)	Image begin	Image size	File Size
Offset of image bitmap	Bitmap offset	Bitmap size	Size of bitmap
Adler32 of header	checksum	OAT begin	Load address of OAT file
Load address of OAT Data (Oat Begin + 0x1000)	OAT Data Begin	OAT data end	Last address of OAT Data
Last address of OAT (begin + size)	OAT end	Patch Delta	Used in offset patching
Address of image roots	Image Roots	Compile PIC	

Image_roots array (serialized)

- Addr of objectArray
- ..
- Count (8)
- kResolutionMethod
- kImtConflictMethod
- kDefaultImt
- kCalleeSaveMethod
- kRefsOnlySaveMethod
- kRefsAndArgsSaveMethod
- kDexCaches
- kClassroots

All fields 32-bit (4 bytes)

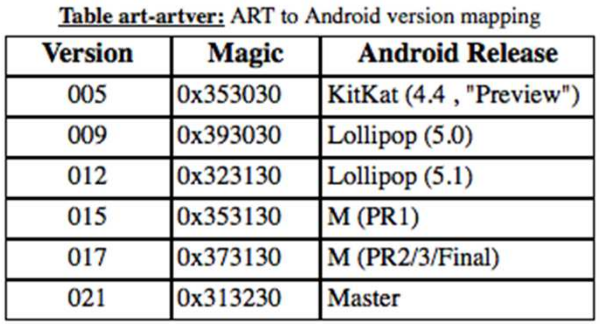


Table art-artver: ART to Android version mapping

Version	Magic	Android Release
005	0x353030	KitKat (4.4 , "Preview")
009	0x393030	Lollipop (5.0)
012	0x323130	Lollipop (5.1)
015	0x353130	M (PR1)
017	0x373130	M (PR2/3/Final)
021	0x313230	Master

Lollipop (5.x)

art\n	009-012
Image begin	Image size
Bitmap offset	Bitmap size
checksum	OAT begin
OAT Data begin	OAT data end
OAT end	Patch Delta
Image Roots	Compile PIC

Marshmallow (PR1)

art\n	015
Image begin	Image size
ART Fields Offset	ART Fields Size
Bitmap offset	Bitmap size
checksum	OAT begin
OAT Data begin	OAT data end
OAT end	Patch Delta
Image Roots	Compile PIC

Marshmallow (PR2-Release)

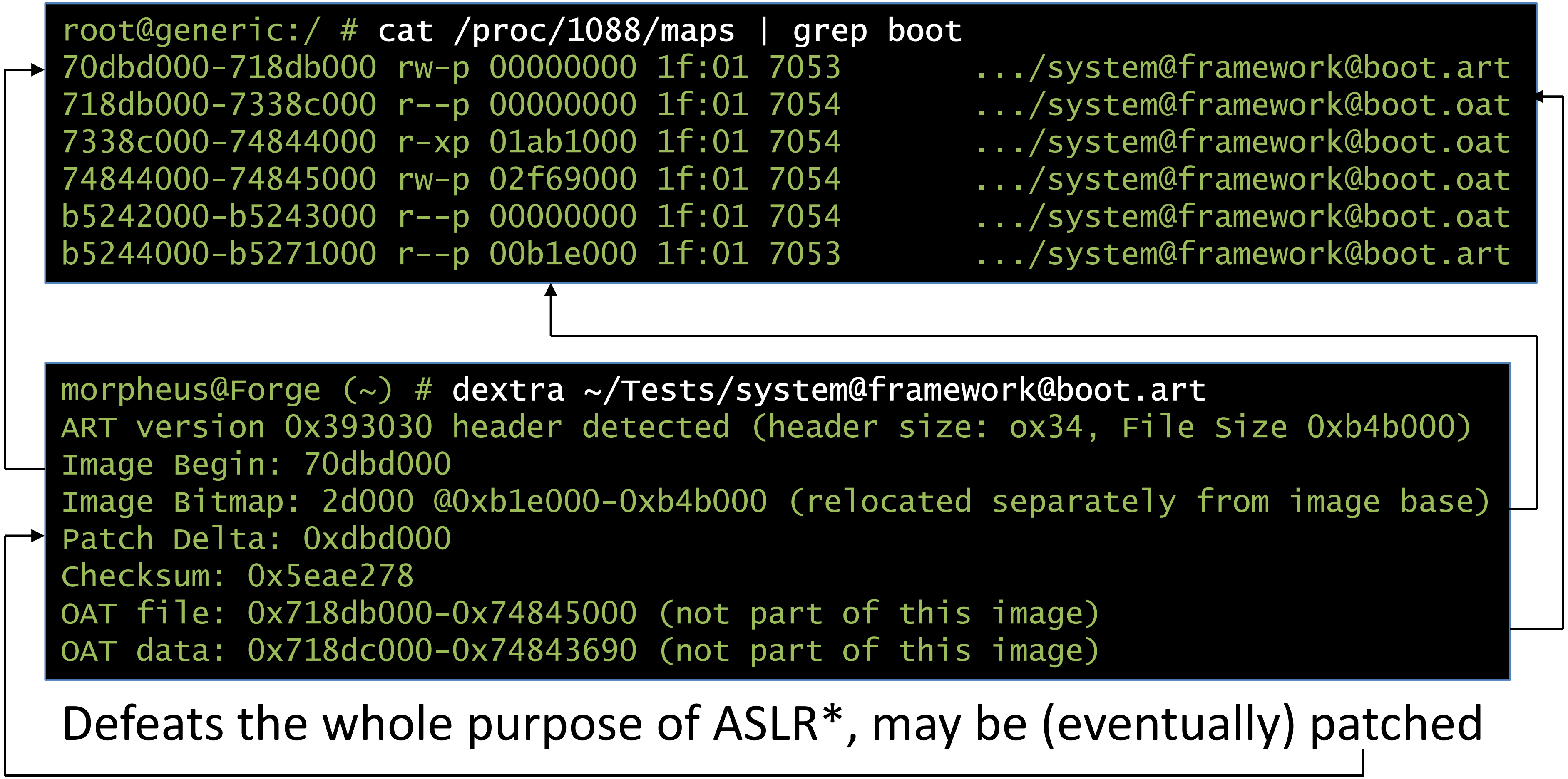
art\n	017-???
Image begin	Image size
OAT checksum	OAT begin
OAT Data begin	OAT Data end
OAT end	Patch Delta
Image Roots	Size of Pointer
Compile_pic	Objects Offset
Objects Size	Fields Offset
Fields Size	Methods offset
Methods size	Strings Offset
Strings size	Bitmap offset
Bitmap size	

... Followed by Image Roots

All fields 32-bit (4 bytes)

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?)**

Example: Inspecting ART

- You can use the AOSP's oatdump to inspect ART (and OAT) files:

```
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: Inspecting ART

- M’s oatdump adds more options:

```
Usage: oatdump [options] ...
    Example: oatdump --image=$ANDROID_PRODUCT_OUT/system/framework/boot.art
    Example: adb shell oatdump --image=/system/framework/boot.art
....
--list-classes may be used to list target file classes (can be used with filters).
      Example: --list-classes
      Example: --list-classes --class-filter=com.example.foo

  --list-methods may be used to list target file methods (can be used with filters).
      Example: --list-methods
      Example: --list-methods --class-filter=com.example --method-filter=foo

  --symbolize=<file.oat>: output a copy of file.oat with elf symbols included.
      Example: --symbolize=/system/framework/boot.oat

  --class-filter=<class name>: only dumps classes that contain the filter.
      Example: --class-filter=com.example.foo

  --method-filter=<method name>: only dumps methods that contain the filter.
      Example: --method-filter=foo

  --export-dex-to=<directory>: may be used to export oat embedded dex files.
      Example: --export-dex-to=/data/local/tmp

  --addr2instr=<address>: output matching method disassembled code from relative
                          address (e.g. PC from crash dump)
      Example: --addr2instr=0x00001a3b
```

Example: Reversing ART

- Better option: http://NewAndroidBook.com/tools/dextra (formerly Dexter)

```
Zephyr:Dextra morpheus$ ./dextra
Usage: ./dextra [...] _file_
Where: _file_ = DEX or ART/OAT file to open
And [...] can be any combination of:
       -l List contents of file (classes is in dex, oat, or ART)
       -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 -j -m)
...
OAT specific options:
       -h: Just dump file header
       -dextract     Extract embedded DEX content from an OAT files
       -o            Display addresses as offsets (useful for file editing/fuzzing)
       -delta value  Apply Patch delta

ART specific options:
       -delta value  Apply Patch delta
       -deep         Deep dump (go into object arrays)

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)

This is DEXTRA, version 1.64.17 (with proper 5.0-6.0(final) .art/.oat), compiled on Nov 30 2015.
For more details: http://NewAndroidBook.com/tools/dextra.html
```

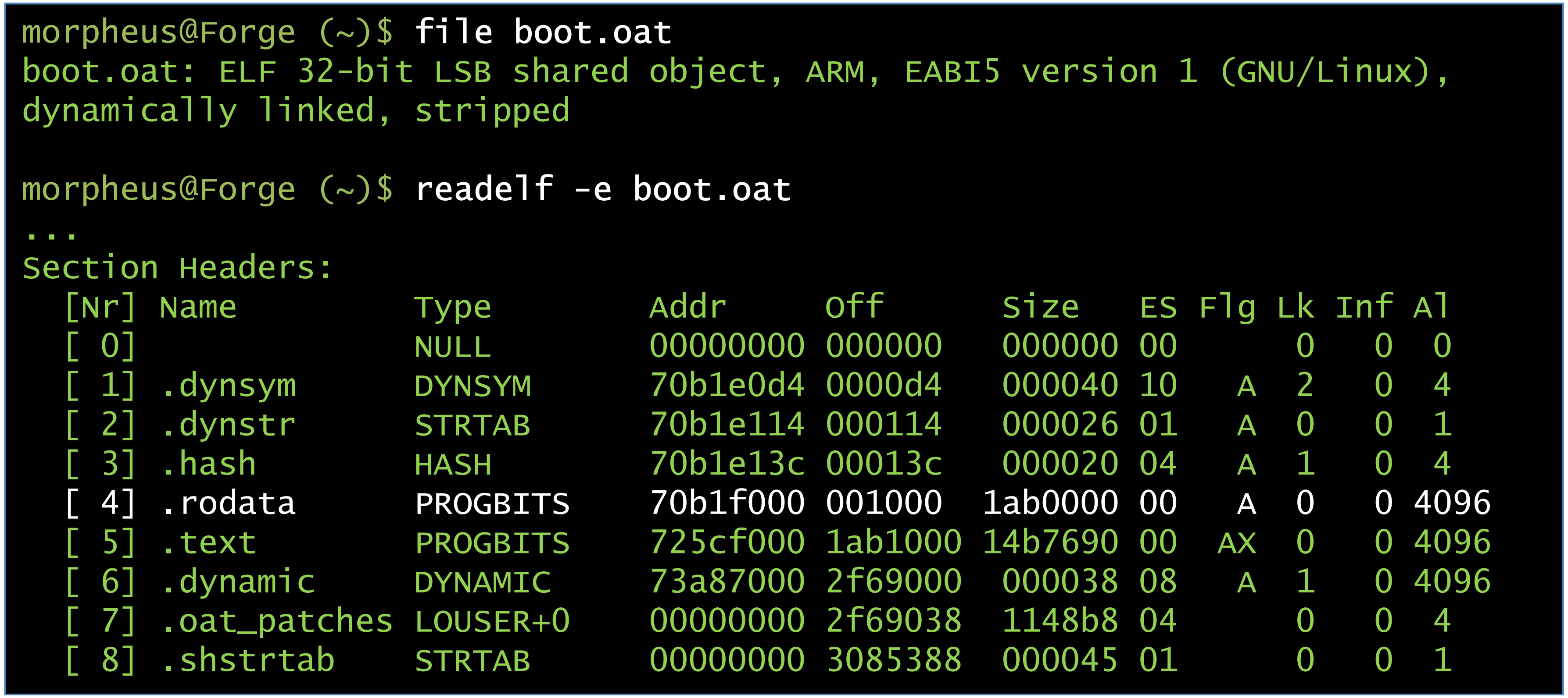
Most of DexTRA's features eventually end up in oatdump (..keep up the good work, Google!)

Tool comparison

Function	Oatdump	Dextra
OS support	Android only	Android Linux Mac OS X Windows (cygwin)
grep(1) friendly	No	Yes
Colorful output	No	Yes
Concise syntax	No	Yes
Open Source	Yes. And very messy	No (but not as messy ☺)

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

	art\n	037-064	OAT Magic header (“oat\n“ and version (“**039** “-”**064”**)
Adler32 of header	checksum	Instruction Set	Underlying architecture (ARM, ARM64, x86, etc.)
	Ins. Set Features	Dex file count	Count of Embedded DEX files (told ya DEX is alive)
Offset of Executable (Load Address)	Executable offset	I2I Bridge	Interpreter-to-Interpreter Bridge Offset
Interpreter to Compiled Bridge Offset	I2C Bridge	Jni dlsym lookup	Offset of JNI dlsym() lookup func for dynamic linking
Portable IMT Conflict Resolution Offset	Portable IMT	Portable Tramp	Portable Resolution Trampoline Offset
Portable to Interpreter Bridge Offset	P2I Bridge	Quick Gen JNI Tramp	Generic JNI Trampoline Offset
Quick IMT Conflict Trampoline Offset	Quick IMT Conf.	Quick Res Tramp	Quick Resolution Trampoline Offset
Quick to Interpreter Bridge Offset	Q2I Bridge	Patch Offset	
	...removed in 062 ...		
	Key/Value Len		
	Key/Value Store (Len bytes)		

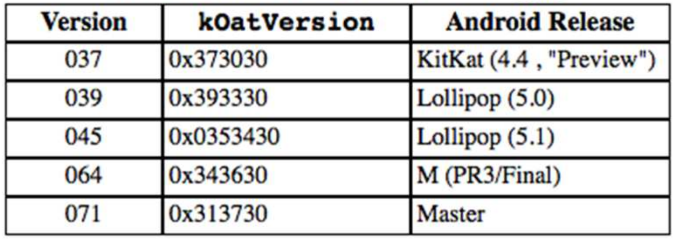
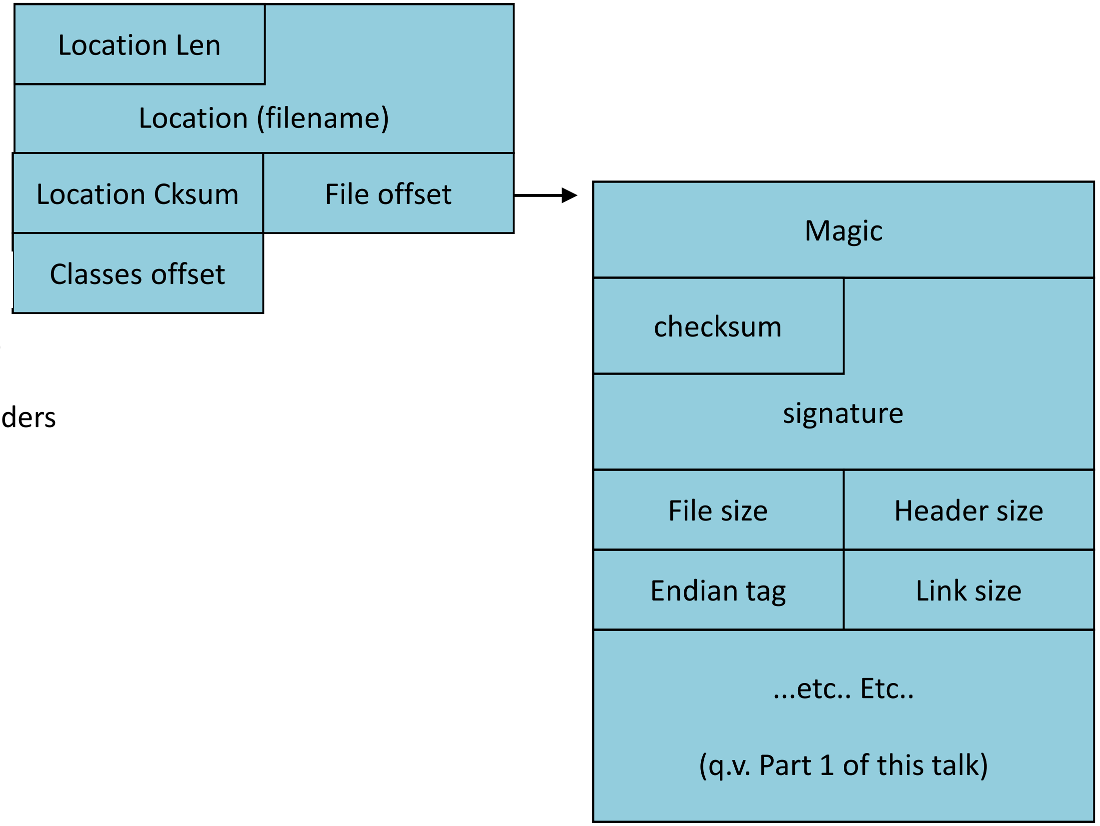


Table art-oatver: OAT to Android version mapping

Version	kOatVersion	Android Release
037	0x373030	KitKat (4.4 , "Preview")
039	0x393330	Lollipop (5.0)
045	0x0353430	Lollipop (5.1)
064	0x343630	M (PR3/Final)
071	0x313730	Master

The OAT DexFile Header

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



Location Len

Location (filename)

Location Cksum | File offset

Classes offset

Points to array of oat_class_headers

Followed by (possibly) method bitmap

Followed by array of oat_method_headers

Magic

checksum

signature

File size | Header size

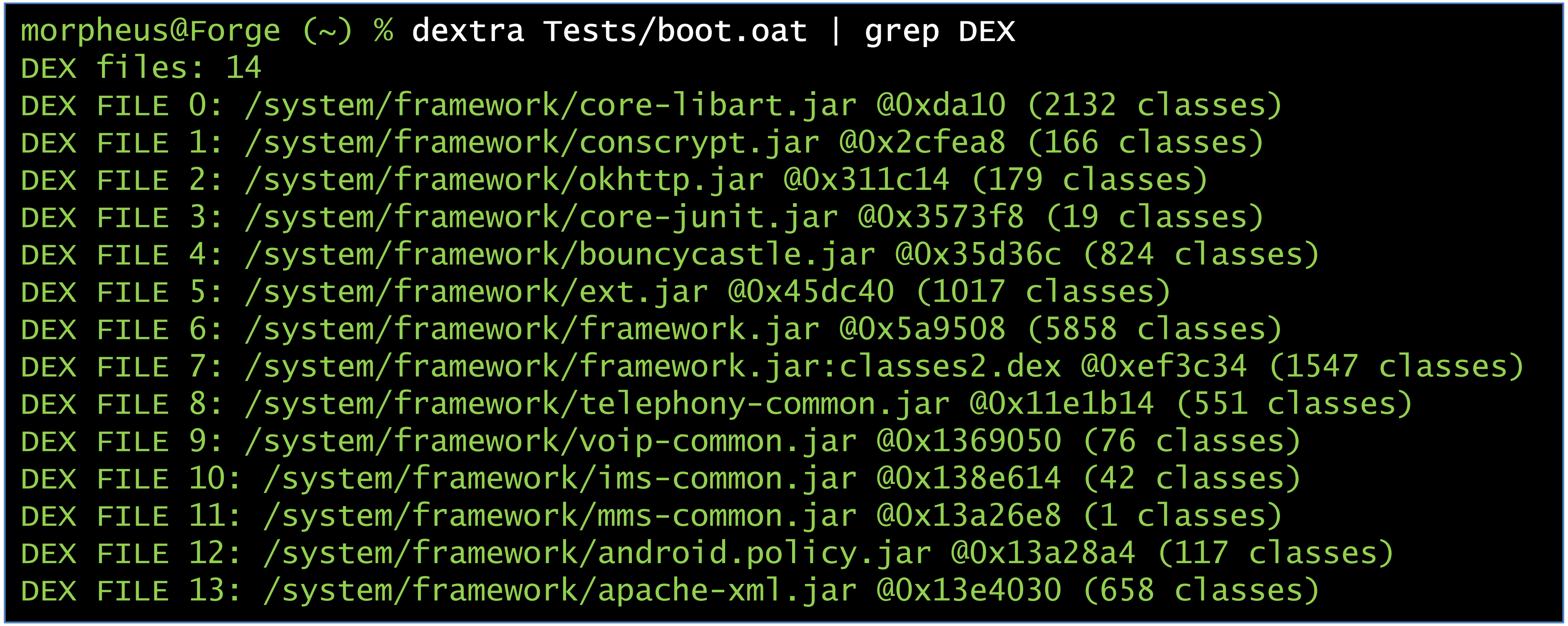
Endian tag | Link size

...etc.. Etc..

(q.v. Part 1 of this talk)

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



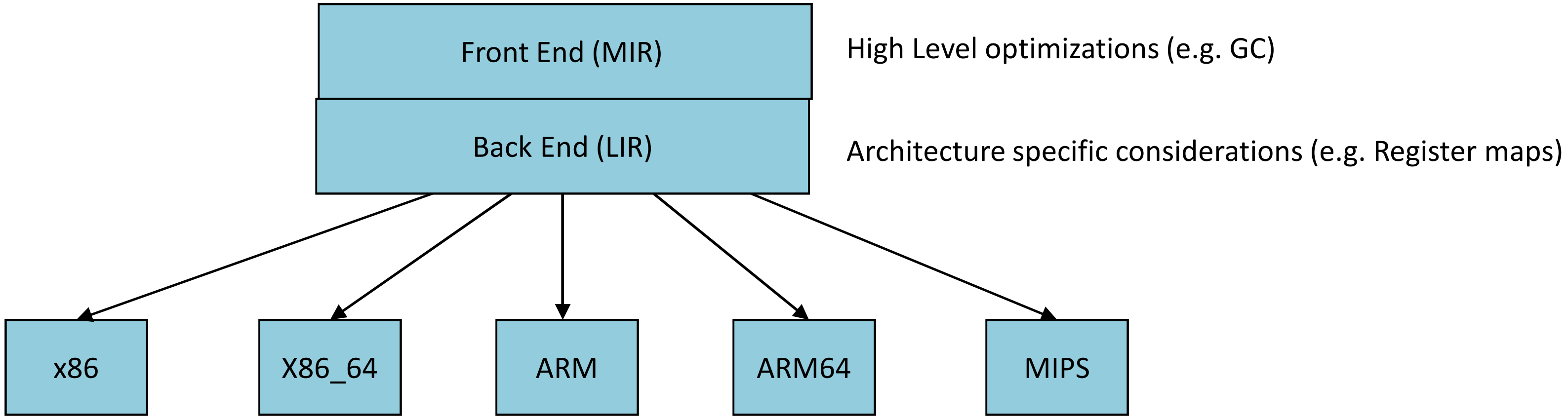
```
morpheus@Forge (~) % dextra Tests/boot.oat | grep DEX
DEX files: 14
DEX FILE 0: /system/framework/core-libart.jar @0xda10 (2132 classes)
DEX FILE 1: /system/framework/conscrypt.jar @0x2cfea8 (166 classes)
DEX FILE 2: /system/framework/okhttp.jar @0x311c14 (179 classes)
DEX FILE 3: /system/framework/core-junit.jar @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*:



Front End (MIR)

High Level optimizations (e.g. GC)

Back End (LIR)

Architecture specific considerations (e.g. Register maps)

*** - 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
    - There are actually a few 64-bit instructions (e.g. Move-wide) but most DEX code doesn't use them)
- 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...

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 1.17.64)
  - 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**

Oh, and…

@Technologeeks Training

- "Android Internals & Reverse Engineering" training discusses all this, and more
  - Native level debugging and tracing
  - Binder internals
  - Native services
  - Frameworks
  - DEX, OAT structure and reversing

- Based on "Android Internals" – (available) Volume I and (Jan 2016) Volume II

- http://Technologeeks.com/AIRE
  - Next training: To Be announced!

- Follow @Technologeeks for updates, training, and more!