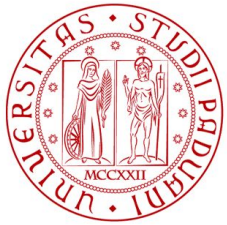
Mobile Security

Dr. Eleonora Losiouk
Department of Mathematics
University of Padua
elosiouk@math.unipd.it
https://www.math.unipd.it/~elosiouk/

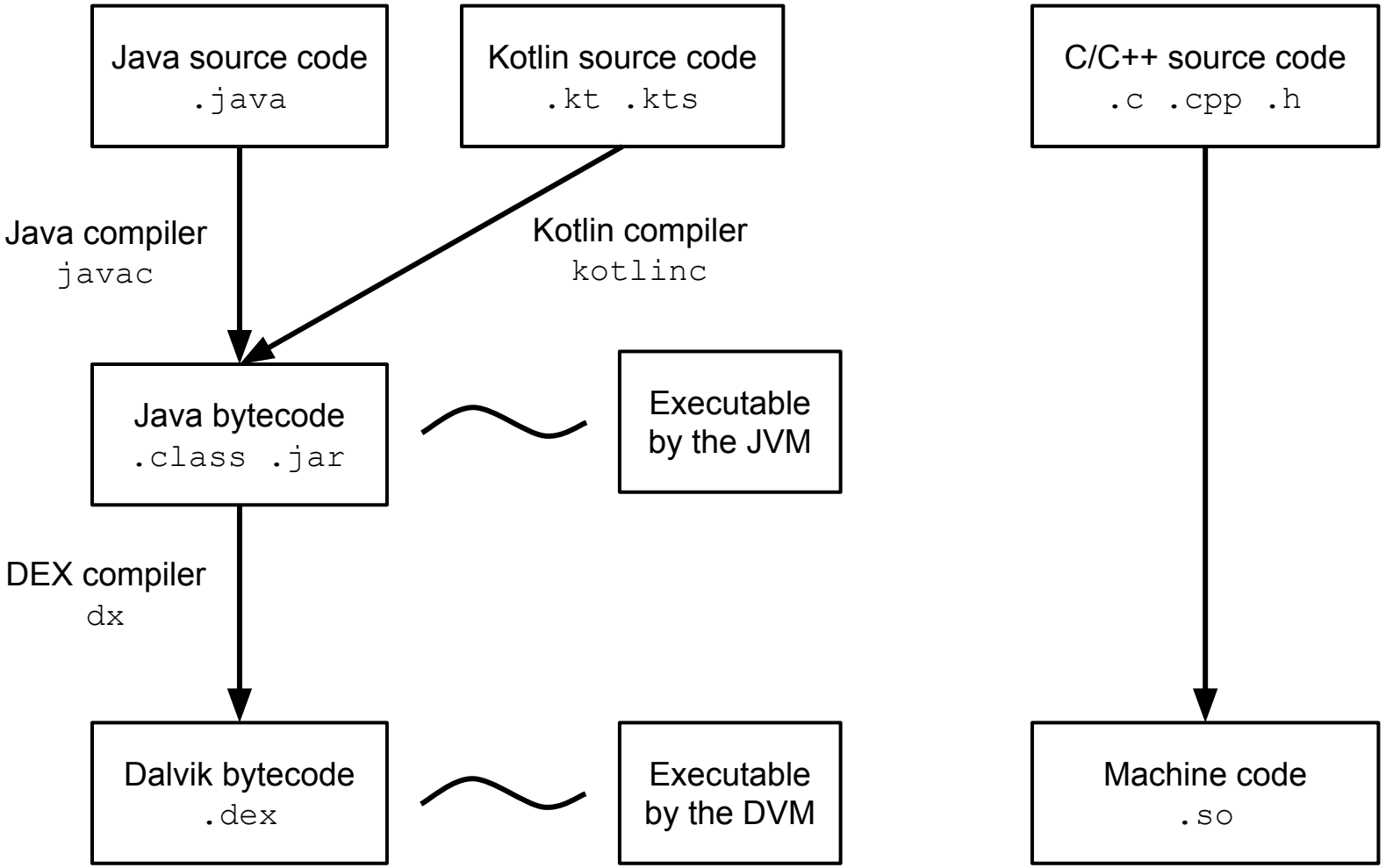


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

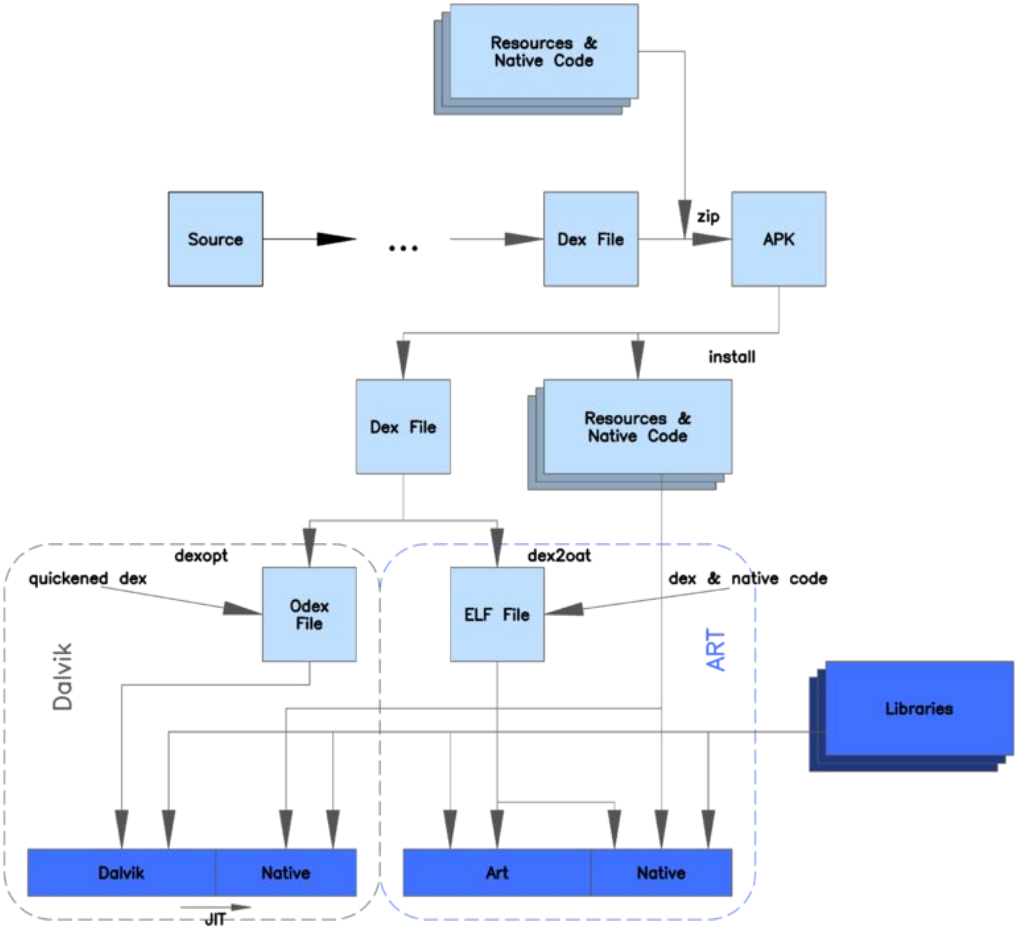
SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

DIPARTIMENTO
MATEMATICA

The Compilation Process



The Big Picture



Taken from [stackoverflow]

Dalvik bytecode, it looks like this:


```
.method foo(ILcom/mobisec/Peppa;)I
    .registers 4

    invoke-virtual {v4, v3}, Lcom/mobisec/Peppa;->pig(I)I
    move-result v0

    add-int v1, v3, v0
    return v1
.end method
```

Dalvik Bytecode

- Dalvik knows about OO concepts
  - Classes, methods, fields, "object instances"
- Dest-to-src syntax
  - E.g., "move r3, r2" means r2 → r3
- Types
  - Built-in: V (void), B (byte), S (short), C (char), I (int), Z (boolean), ...
  - Actual Classes (syntax: L<fullyqualifiedclassname>;)
    - Landroid.content.Intent;
    - Lcom.mobiotsec.Peppa;

Example (Java)


```
class Peppa {
    int pig(int x) {
        return 2*x;
    }

    static int foo(int a, Peppa p) {
        int b = p.pig(a);
        return a+b;
    }
}
```

Example (Dalvik bytecode)


```
.method pig(I)I
    .registers 3

    mul-int/lit8 v0, v2, 0x2

    return v0
.end method
```

```
int pig(int x) {
    return 2*x;
}
```

Example (Dalvik bytecode)


```
.method foo(ILcom/mobisec/Peppa;)I
    .registers 4

    invoke-virtual {v4, v3}, Lcom/mobisec/Peppa;->pig(I)I
    move-result v0

    add-int v1, v3, v0
    return v1
.end method
```

```
int foo(int a, Peppa p)
{
    int b = p.pig(a);
    return a+b;
}
```

Example of Dalvik instructions (doc)

- Moving constants/immediates/registers into registers
  - `const v5, 0x123`
  - `move v4, v5`

- Math-related operations (many, many variants)
  - `add-int v1, v3, v0`
  - `mul-int/lit8 v0, v2, 0x2`

Example of Dalvik instructions

- Method invocation
  - `invoke-virtual {v4, v3}, Lcom/mobisec/Peppa;->pig(I)I`
  - `invoke-static ...`
  - `invoke-{direct, super, interface} ...`

- Getting return value
  - `invoke-virtual {v4, v3}, Lcom/mobisec/Peppa;->pig(I)I`
  - `move-result v5`

Example of Dalvik instructions

- Set/get values from fields
  - `iget, iget-object, ...`
  - `iput, iput-object, ...`
  - `sget, sput ...` (for static fields)

- Instantiate new object
  - `new-instance v2, Lcom/mobisec/Peppa;`

Example of Dalvik instructions

- Conditionals / control flow redirection
  - `if-ne v0, v1, :label_a`
    `...`
    `:label_a`
    `...`
  - `goto :label_b`

Which component is actually executing Dalvik?

- In the past (up to Android 4.4)
  - DVM, libdvm.so
  - When about to execute a method, compile it and run
    - Compile process: "Dalvik bytecode -> machine code"
  - Rephrasing: compilation is done "on demand"
  - We refer to this as Just-In-Time compilation (JIT)

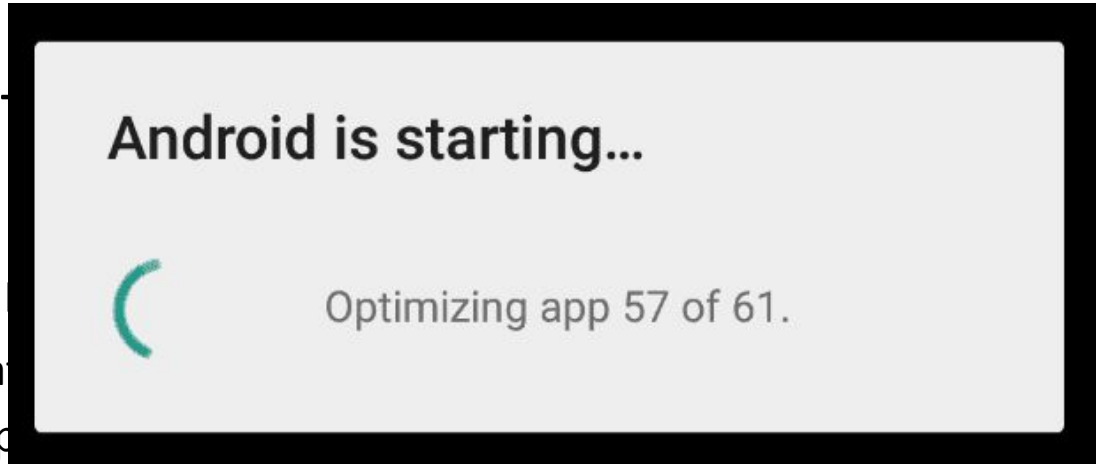
- Compiled code is stored in a cache

Then, Android ART

- ART stands for Android Run-Time
  - It replaced the old DVM
  - It was introduced in Android 4.4 as optional, mandatory in Android 5
- Ahead-Of-Time compilation
  - Compilation happens at app installation time

ART vs DVM

- Pro: The app boot and execution are MUCH faster
  - Because everything is already compiled
- Cons: ART
- Major co
  - Installa
  - Bad rep



New Version of ART

- Profiled-guided JIT/AOT
  - Introduced in Android 7
- ART profiles an app and precompiles only the "hot" methods, the ones most likely to be used
- Other parts of the app are left uncompiled

New Version of ART

- It is pretty smart...
  - It automatically precompiles methods that are "near to be used"
  - Precompilation only happens when the device is idle and charging
- Biggest Pro: quick path to install / upgrade

DVM JIT vs ART AOT vs ART JIT/AOT

	DVM JIT	ART AOT	ART JIT/AOT
App boot time	slowest	fastest	trade-off
App speed	slowest	fastest	trade-off
App install time	fastest	slowest	trade-off
System upgrade time	fastest	slowest	trade-off
RAM/disk usage	lowest	highest	trade-off

"Dalvik and ART" slides: link

ODEX: Optimized DEX

- DEX → **`dexopt`** → ODEX
- It is optimized DEX: faster to boot and to run
- Most (all?) system apps that start at boot are ODEXed
- Note: ODEX is an *additional* file, next to an APK
- Cons
  - ODEX files take space
  - Device-dependent (note: it is still bytecode)

The analogous of ODEX for ART is tricky...

- The new Android Run-Time uses two formats
- The ART format (.art files)
  - It contains pre-initialized classes / objects
- The OAT files
  - Compiled bytecode to machine code, wrapped in an ELF file
  - It can contain one or more DEX files (the actual Dalvik bytecode)
  - Obtained with **`dex2oat`** (usually run at install time)

The analogous of ODEX for ART is tricky...

- The confusing part: you still have **`.odex`** files!
- Now **`.odex`** files are OAT-formatted files!

When are these two formats used?

- ART format:
  - Only *one* file: **boot.art**
  - It contains the pre-initialized memory for most of the Android framework
    - Huge optimization trick
- OAT format:
  - One important file: **boot.oat**
    - It contains the pre-compiled most important Android framework libraries
  - All the "traditional" ODEX files are OAT files
  - You can inspect them with Android-provided **`oatdump`**

When a new app is starting

- All apps processes are created by forking Zygote
- Zygote can be seen as the "init" of Android
  - A "template" process for each app
- Optimization trick
  - boot.oat is already mapped in memory
  - No need to re-load the framework!

Tools time!

Android Application Package (APK)

- An APK is a zip file (kinda)

- `$ unzip app.apk`

- Content
  - AndroidManifest.xml (compressed)
  - classes.dex (raw Dalvik bytecode)
  - resources.arsc (compressed)
  - res/*.xml (compressed)

Unpacking APKs

- unzip app.apk
  - AndroidManifest.xml (compressed)
  - classes.dex
  - resources (compressed)

smali/baksmali (doc1, doc2)

- $ baksmali disassemble app.apk -o app
  - Disassemble DEX files
  - Output: a .smali file for each class
  - Dalvik bytecode in "smali" format

- $ smali assemble app -o classes.dex
  - Assembler for DEX files

apktool

- apktool is awesome
- It embeds baksmali/smali
- It unpacks / packs APKs, including resources and manifest files
- $ apktool d app.apk -o output
- $ apktool b output -o patched.apk
  - Used for repackaging attacks

Disassembly vs. Decompilation

- Disassembly
  - classes.dex binary file -> Dalvik bytecode "smali" representation
  - machine code bytes -> assembly representation (mov eax, edx)

- Decompilation
  - Go from assembly/bytecode to source code-level representation
  - Dalvik bytecode -> Java source code

How to decompile

- All-in-one tools
  - JEB (top product, commercial, expensive, but free version available!)
  - BytecodeViewer (pretty good one)
  - jadx

- Using a Java decompiler (Java bytecode -> Java)
  - Dalvik bytecode -> Java bytecode
    - dex2jar
  - Java bytecode -> Java source code
    - Jd-GUI

Decompilation

- Decompiling Dalvik bytecode is usually simple
- Packing techniques and obfuscation tricks try to make decompilers' lives very difficult
- When they don't work, you gotta read the bytecode