

Android – Static Analysis 2

REVERSE ENGINEERING

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

João Paulo Barraca

Native Applications

Native Applications

- Apps developed with OS provider's language and frameworks.
 - Java, Kotlin, Objective-C, Swift
- Android applications are compiled and packaged with resources
 - Reversing such app can be done to Java (JADX) or Smali (apktool)
- Can access all API's made available by OS vendor.
- But...
 - SDK's are platform-specific.
 - Each mobile OS comes with its own unique tools and GUI toolkit.
 - Developing a world wide app requires multiple implementations

Java Native Interface

- Java applications can call functions from external libraries
 - Libraries can be implemented in Java, and packaged as classes
 - Libraries can also be implemented in any other language
 - Providing that an interfaces allows serialization and name resolution
- **JNI: allows the definition of Java methods, whose implementation is present in native code**
 - When a method is invoked, the objects are serialized, and the respective native symbol is loaded and the code executed.
 - There is a penalty due to serialization, but also a performance boost due to native code execution.
 - References:
 - [JNI Functions \(oracle.com\)](#)
 - [Contents \(oracle.com\)](#)
- Standard mechanism for Java (not specific for android)

Native code

Android
NDK
apps

Dalvik runtime

Android
SDK
apps

JNI

Android app framework

Standard libraries

Linux kernel + Android extensions

■ Android Native Development Kit (NDK)

- Provides a Dev. Kit allowing C/C++ applications to access Android resources
 - Similar to the standard SDK available to Java applications
- Developers may choose how to develop application code
 - Java: faster development and richer API
 - Native: faster execution, access to Linux subsystem, and more complex reverse engineering
 - Sometimes binary blobs are the only method to access a cryptographic method, DRM or hardware device
 - Sometimes the developer wishes to further obfuscate the code by compiling it to native code
- As libraries are native, an application must include multiple implementations
 - One for each architecture
 - A new device may not use applications that lack an implementation for that architecture
 - Implies using portable code that works in multiple architectures (arm, armv7, arm64, x86, x64, ...)

■ Android binary libraries – Mediacode.apk

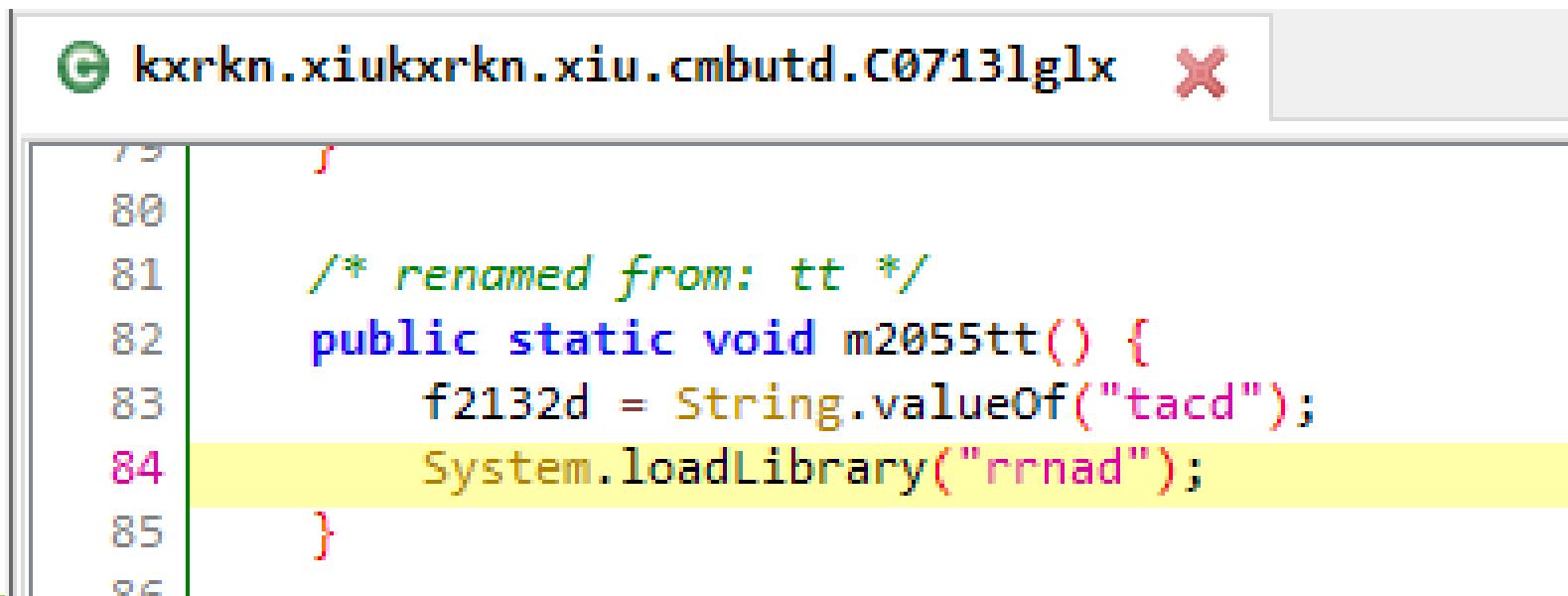
- Application contains DEX code and binary blobs
- One version for each architecture
 - armeabi: ARM 32bits no Floating Point
 - mips: MIPS
 - x86: intel X86 32bits
- Libraries export symbols to be used through JNI
 - nm -gD lib/x86/librrnad.so | grep JNI

```
lib
lib/armeabi
lib/armeabi/librrnad.so
lib/armeabi-v7a
lib/armeabi-v7a/librrnad.so
lib/mips
lib/mips/librrnad.so
lib/x86
lib/x86/librrnad.so
```

Mediacode.apk

■ Android binary libraries – Mediacode.apk

- Before the binary libraries can be used, Java must load them
 - `System.loadLibrary`: argument is the library name (without lib, architecture or .so)
 - `System.load`: generic object load. Argument is the full path to the object
 - `The JNI_OnLoad method is called automatically (in the lib)`
 - Allows automatic setup of data structures and generic initialization
 - May be abused if malware is present
- Without the library, application will crash when external methods are requested



A screenshot of a debugger interface showing Java code. The code is part of a class named `kxrkn.xiukxrkn.xiu.cmbutd.C07131g1x`. The code includes a comment `/* renamed from: tt */` and a method `m2055tt()` which contains a `System.loadLibrary("rrrnad")` call. The line `System.loadLibrary("rrrnad")` is highlighted with a yellow background.

```
79
80
81     /* renamed from: tt */
82     public static void m2055tt() {
83         f2132d = String.valueOf("tacd");
84         System.loadLibrary("rrrnad");
85     }
86 }
```

JNI Arguments

- Native methods support arguments from Java code
 - Arguments are pointers to Java structures
 - Must be processed using specific methods, capable of handling the native Java types
- Native methods can also call Java methods, and classes
 - Mainly achieved by the first argument of any JNI method: `JNIEnv*`
- `JNIEnv*` is a pointer to a structure with a large number of functions.
 - JNI Methods use it to invoke Java methods and handle Java types

■ Android binary libraries – Mediacode.apk

- In the java world native methods are declared:
 - With the keyword native
 - Without implementation
- Easy to spot if we have the java or smali code
 - Java: public native String decryptString(String)
 - Smali: .method public native decryptString(Ljava/lang/String;)Ljava/lang/String

The screenshot shows a search interface with the following fields:

- Search for text: native
- Search definitions of: Class Method Field Code
- Search options: Case insensitive

The results are divided into two sections:

- Node**:
NS com.umeng.common.util.C0708DeltaUpdate.bspatch(String, String,...)
S com.umeng.common.util.C0708DeltaUpdate.m1973b(Context) String
S com.umeng.common.util.C0708DeltaUpdate.m1973b(Context) String
NS kxrkn.xiukxrkn.xiu.cmbtud.C0713lglx.m2054a(String) void
S kxrkn.xiukxrkn.xiu.cmbtud.C0713lglx.tt2(Context) void
S kxrkn.xiukxrkn.xiu.cmbtud.C0713lglx.tt2(Context) void
- Code**:
public static native int bspatch(String str, String str2, String str3);
public static native int bspatch(String str, String str2, String str3);
public static native int bspatch(String str, String str2, String str3);
public static native void m2054a(String str);
public static native void m2054a(String str);
public static native void m2054a(String str);

JNI Dynamic Linking

- Dynamic linking is done “automagically” as long as the names of the methods in the library follow a fixed template
 - The library is loaded into the JVM and the methods are linked automatically
 - Implies that symbols are present in the library (not stripped)
- Assuming that our hello world app had a class named Worker, loading a method named doWork, the method in the function would be named:

java_pt_ua_deti_hello_Worker_dowork()

magic

Package name

Class name

method

JNI Static Linking

- Linking must be done “manually”, by the binary code, before the methods are used
 - Allows methods to have any name (read obfuscation!)
 - A fixed method (`JNI_OnLoad`) is called after the lib is loaded
 - Library registers the mapping between java methods and native methods using `RegisterNatives`.
 - Must do this once for each method called.

```
jint RegisterNatives(JNIEnv *env, jclass clazz, const JNINativeMethod *methods, jint nMethods);

typedef struct {
    char *name;
    char *signature;
    void *fnPtr;
} JNINativeMethod;
```

JNI Static Linking

```
typedef struct {  
    char *name;  
    char *signature;  
    void *fnPtr;  
} JNINativeMethod;
```

Java method name

Address of native method

Signature using the following specifiers:

- Z: boolean
- B: byte
- C: char
- S: short
- I: int
- J: long
- F: float
- D: double
- L fully-qualified-class ; :fully-qualified-class
- [type: type[]]
- (arg-types) ret-type: method type
- V: void

**String foo(Int, Boolean) would result in:
(IZ)Ljava/lang/String**

[JNI Types and Data Structures \(oracle.com\)](http://oracle.com)

JNI Static Linking

- Reverse engineering of the library blob is the most viable alternative
 - Some symbols must always be available: `JNI_OnLoad`
 - Remaining symbols usually are available, although they may have obfuscated names
- Process
 - Load the library in a tool: ghidra, IDA, BinaryNinja, R2, etc...
 - Find the `JNI_OnLoad` method
 - Determine when `RegisterNatives` is called
 - Determine the arguments passed to the function
 - Will allow determining the method mapping and the arguments of each function
 - Actually, the arguments may also help identifying the method

Exercises 1 and 2

Determine which method are actually loaded from the MediaCodec.apk shared libraries.

strings

- Do we have interfaces matching the functions we know to be native?
 - int bspatch(String str, String str2, String str3)
 - void m2054a(String s) /* or void a(String s) with jadx */

```
strings lib/x86/librrnad.so |grep "(Ljava/lang/String"
```

```
(Ljava/lang/String;)V
(Ljava/lang/String;I)I
(Ljava/lang/String;)I
(Ljava/lang/String;)Ljava/lang/Object;
(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/ClassLoader;)V
(Ljava/lang/String;)Ljava/lang/Class;
(Ljava/lang/String;ZLjava/lang/ClassLoader;)Ljava/lang/Class;
(Ljava/lang/String;Ljava/lang/String;)Landroid/content/Intent;
(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
(Ljava/lang/String;)Ljava/lang/String;
(Ljava/lang/String;)Ljava/crypto/SecretKeyFactory;
(Ljava/lang/String;)Ljava/crypto/Cipher;
```

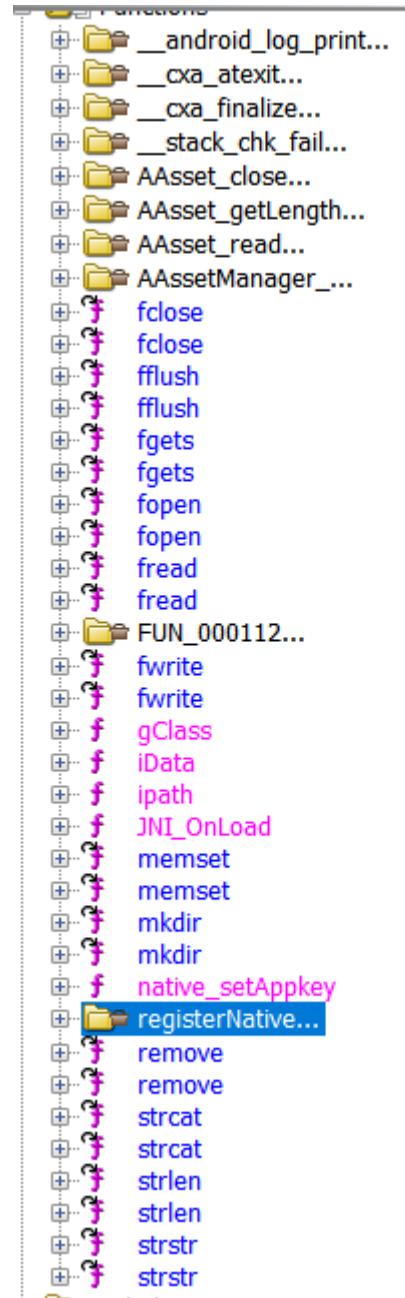
nm

libraries are not exactly contained on the apk
they are on device or installed on
runtime

- Do we have dynamic linking?
- Let's look for methods following the known pattern
- nm -gD lib/x86/librrnad.so | grep java_
 - None...
- Conclusion
 - We have artifacts pointing to Java types
 - We do not have indication of Dynamic Linking

ghidra

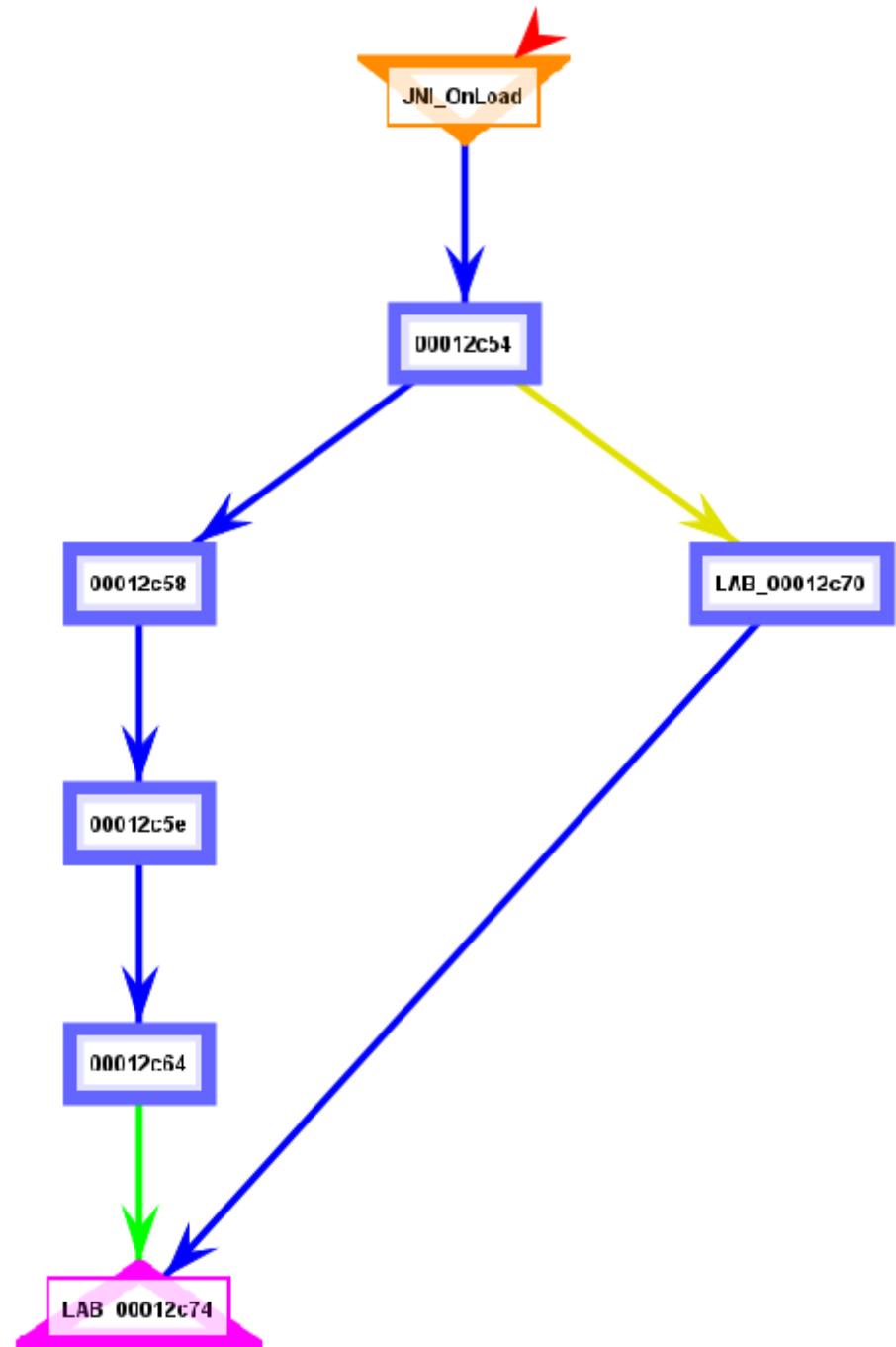
- Open ghidra
- Create a new project
- Load a library
 - I loaded all and selected ARMEABI
- Several interesting functions discovered
 - **JNI_OnLoad**
 - `registerNatives`, `registerNativeMethods`
 - `FUN_00011230`, `FUN_000270`, `FUN_11290`, `FUN_112b4`
 - `native_setAppKey`
- Coherent with Static Linking
- Explore the functions, exports, Classes, etc... lots of info



- Graph -> Block Flow from `JNI_OnLoad`

- Decompile `JNI_OnLoad`

```
2 undefined8 JNI_OnLoad(int *param_1,undefined4 param_2,undefined4 param_3)
3 {
4     int iVar1;
5     uint uVar2;
6     _JNIEnv *local_c;
7     undefined4 uStack8;
8
9     local_c = (_JNIEnv *)0x0;
10    uStack8 = param_3;
11    iVar1 = (**(code **)(*param_1 + 0x18))(param_1,&local_c,0x10004);
12    if (iVar1 == 0) {
13        iData(local_c);
14        iVar1 = registerNatives(local_c);
15        uVar2 = -(uint)(iVar1 == 0) | 0x10004;
16    }
17    else {
18        uVar2 = 0xffffffff;
19    }
20    return CONCAT44(param_1,uVar2);
21 }
22 }
```



JNI_OnLoad

```
2 undefined8 JNI_OnLoad(int *param_1,undefined4 param_2,undefined4 param_3)
3
4 {
5     int iVar1;
6     uint uVar2;
7     _JNIEnv *local_c;
8     undefined4 uStack8;
9
10    local_c = (_JNIEnv *)0x0;
11    uStack8 = param_3;
12    iVar1 = (**(code **)(*param_1 + 0x18))(param_1,&local_c,0x10004);
13    if (iVar1 == 0) {
14        iData(local_c);
15        iVar1 = registerNatives(local_c); ← Call registerNatives
16        uVar2 = -(uint)(iVar1 == 0) | 0x10004;
17    }
18    else {
19        uVar2 = 0xffffffff;
20    }
21    return CONCAT44(param_1,uVar2);
22 }
23 }
```

JNI_OnLoad: ghidra with a JNI GDT and retyping

- Loading the jni_all.gdt, and retyping the variables, allows resolution of symbols, such as the FindClass.

```
2 undefined8 JNI_OnLoad(JNIEEnv *param_1,undefined4 param_2,undefined4 param_3)
3
4 {
5     jclass p_Var1;
6     int iVar2;
7     uint uVar3;
8     JNIEEnv *local_c;
9     undefined4 uStack8;
10
11    local_c = (JNIEEnv *)0x0;
12    uStack8 = param_3;
13    p_Var1 = (*(*param_1)->FindClass) (param_1, (char *)&local_c);
14    if (p_Var1 == (jclass)0x0) {
15        iData((JNIEEnv *)local_c);
16        iVar2 = registerNatives((JNIEEnv *)local_c);
17        uVar3 = -(uint)(iVar2 == 0) | 0x10004;
18    }
19    else {
20        uVar3 = 0xffffffff;
21    }
22    return CONCAT44(param_1,uVar3);
23 }
24 }
```

registerNatives

```
2 /* registerNatives(_JNIEnv*) */
3
4 void registerNatives(_JNIEnv *param_1)
5
6 {
7     undefined *local_14;
8     char *local_10;
9     code *local_c;
10
11    local_14 = &DAT_0001503b;
12    local_10 = "(Ljava/lang/String;)V";
13    local_c = native_setAppkey + 1;
14    registerNativeMethods(param_1,nativeClassForJni,(JNINativeMethod *)&local_14,1);
15
16    return;
17}
```

Name of Java method name

Prototype void foo(String)

Native Function

Call registerNativesMethods
Is this the actual register method?

registerNativeMethods

```
4 jclass registerNativeMethods(JNIEnv *param_1,char *param_2,JNINativeMethod *param_3,int param_4)
5
6 {
7     jclass clazz;
8     uint uVar1;
9     jthrowable p_Var2;
10
11    clazz = (*(*param_1)->FindClass)(param_1,param_2);
12    if (clazz != (jclass)0x0) {
13        uVar1 = (*(*param_1)->RegisterNatives)(param_1,clazz,(JNINativeMethod *)param_3,param_4);
14        p_Var2 = (*(*param_1)->ExceptionOccurred)(param_1);
15        if (p_Var2 == (jthrowable)0x0) {
16            clazz = (jclass)(~uVar1 >> 0x1f);
17        }
18        else {
19            (*(*param_1)->ExceptionClear)(param_1);
20            clazz = (jclass)0x1;
21        }
22    }
23    return clazz;
24 }
```

Actual registration made through JNIEnv method

Web and Hybrid applications

■ Why not Native apps?

- Native Apps are not that good (or not always that good)
 - Have low Code Reusability
 - Require more development and maintenance
 - Requires designers and developers' experts on multiple architectures
 - Have low upgrade flexibility
- Once was the traditional way of developing applications
 - Currently being surpassed by web and hybrid applications
- From a RE perspective, the toolset and languages are very different
 - More complex to analyze
 - Better commonly available obfuscators

Web apps

- Use standard web technologies (HTML, CSS, Javascript)
- Especially since HTML5 allowed:
 - Advanced UI components
 - Access to media types and sources
 - Access to geolocation
 - Access to local storage
- Look like a standard application (present an icon)
- Completely different stack
 - Standalone Mobile Web Browser

Hybrid apps

- Combine both worlds: Native and Web
 - a thin Java application with a Web application
- Most commonly:
 - Web for the interface
 - Java for the application backend
 - Custom Interface connecting both levels
- Installable from the store and indistinguishable from native apps
 - As devices are more powerful, these are becoming very common

Typical frameworks



TCATECHNOLOGIES
IMPLEMENT YOUR THOUGHTS

RE Perspective

- Most frameworks use JS, but sometimes with custom VMs
- Packaging consists of adding the application JS code, HTML, styles and remaining resources
 - May use a bundle, including all resources
 - May leave resources bare in the APK
 - May use binary libs with obfuscated code, but frequently they are just plugins for native functions
- Code is frequently obfuscated
 - An inheritance of the JS obfuscators available
- Code may be compiled to an intermediate representation
 - Decompilers are not that robust as the ones for Java
- RE support is lacking...

- Runs on the Apache Cordova infrastructure
 - Framework implemented in Java
 - Application presented through a Web View
- Actual application is a webpage in the assets/www directory
 - Cordova Plugins in www/plugins
 - Implemented in JS, communicating with main framework through interface
- Framework is event driver with actions activated on interactions

- Every file contains a single line

- Minified code
- Pushing logic, or a handler

- Index.html as the entry point

- Workflow:

- Beautify code and extract information
- Launching Cordova on local PC
- Inspection through browser
- Dynamic Analysis

	102-es2015.51371bd9617aa978dfbf.js
	103-es5.fb98cd61696ba64d028d.js
	103-es2015.fb98cd61696ba64d028d.js
	104-es5.9e7b364d4c5cc3c37878.js
	104-es2015.9e7b364d4c5cc3c37878.js
	105-es5.b56638b78b99f52128b9.js
	105-es2015.b56638b78b99f52128b9.js
	106-es5.77dd521666c08820e751.js
	106-es2015.77dd521666c08820e751.js
	common-es5.1cc40e9932f563dd10cf.js
	common-es2015.1cc40e9932f563dd10cf.js
	cordova.22d0b106f4dfd80eb9b7.js
	cordova.js
	cordova_plugins.js
	index.html
	main-es5.6e96f9da3b0f73cbb0ed.js
	main-es2015.6e96f9da3b0f73cbb0ed.js
	polyfills-es5.121ee1733fe637c198be.js
	polyfills-es2015.4c8fdf27194d9c6ebca4.js
	runtime-es5.8b23a06649f90db185a8.js
	runtime-es2015.8b23a06649f90db185a8.js
	styles.d25dc14f6bfcbbba633d7.css

App ionfits.apk

```
1 (window.webpackJsonp = window.webpackJsonp || []).push([
2     [3], {
3         a4YZ: function(t, e, n) {
4             "use strict";
5             n.r(e), n.d(e, "createSwipeBackGesture", (function() {
6                 return a
7             }));
8             var r = n("AzGJ"),
9                 a = function(t, e, n, a, i) {
10                 var o = t.ownerDocument.defaultView;
11                 return Object(r.createGesture)({
12                     el: t,
13                     gestureName: "goback-swipe",
14                     gesturePriority: 40,
15                     threshold: 10,
16                     canStart: function(t) {
17                         return t.startX <= 50 && e()
18                     },
19                     onStart: n,
20                     onMove: function(t) {
21                         a(t.deltaX / o.innerWidth)
22                     },
23                     onEnd: function(t) {
24                         var e = o.innerWidth,
25                             n = t.deltaX / e,
26                             r = t.velocityX,
27                             a = r >= 0 && (r > .2 || t.deltaX > e / 2),
28                             c = (a ? 1 - n : n) * e,
29                             u = 0;
30                         if (c > 5) {
31                             var s = c / Math.abs(r);
32                             u = Math.min(s, 540)
33                         }
34                         i(a, n <= 0 ? .01 : n, u)
35                     }
36                 })
37             }
38         }
39     ]
40 ]);
```

App ionfits.apk

Flutter

- UI from Google based on the Dart Language
 - Compiled under the scope of the Dart VM (<https://github.com/dart-lang/sdk>)
 - Designed as a dual purpose framework: Web and Mobile
 - With Native and Web components
 - In mobile devices, Flutter is compiled to a native object (libapp.so)
 - As good reference, check: <https://mrale.ph/dartvm/>
- Two deployment flavors
 - AOT: Ahead of Time – the most frequent – as a bytecode for the Dart VM
 - JIT: Just in Time – for debug builds, interpreted from Source Code
- Project structure:
 - Small java shim to load the actual code
 - Framework in libfutter.so
 - Application in another .so libapp.so (yes, an ELF!)
 - Actually, it contains a snapshot of the VM to be loaded

From a RE perspective

- Flutter compiles Dart to native assembly in a single bundle
 - Internal formats are not publicly known in detail
- By default there is no obfuscation or encryption
 - However the formats are not known
- Flutter applications are difficult to reverse engineer
 - Good for intellectual property
- Some tools start to scratch the surface (mostly extract information from libapp.so), extracting information
 - <https://github.com/mildsunrise/darter>
 - <https://github.com/rscloura/Doldrums>

■ Flutter: Flutter-Weather

- Simple application showing weather info
 - <https://github.com/1hanzla100/flutter-weather>
- Follows typical structure
 - 2 .so: Framework and App for multiple archs
- Current tools extract classes from VM snapshot, but there is little similarity with original code

```
lib/x86_64/libapp.so: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),  
dynamically linked, BuildID[md5/uuid]=409a650592e15d744a33d6a1bdbaa652, strip  
ped
```

```
lib  
lib/arm64-v8a  
lib/arm64-v8a/libapp.so  
lib/arm64-v8a/libflutter.so  
lib/armeabi-v7a  
lib/armeabi-v7a/libapp.so  
lib/armeabi-v7a/libflutter.so  
lib/x86_64  
lib/x86_64/libapp.so  
lib/x86_64/libflutter.so
```

