

# Android Malware Reverse Engineering

# Axelle Apvrille

# Insomni'hack, March 2017



And

*I'll cure you, don't worry.*



## Get started

Lab 1: Basics - Contents of an APK

Lab 2: Static Analysis

Labs 3 and 4: Dynamic Analysis

Lab 5: Using Androguard

Lab 6: Working with Radare2

Lab 7: De-obfuscation

Labs 8 and 9: Unpacking Pangxie and LIAPP

Demo: Debugging an APK

Lab 10: Counter anti-emulator tricks (BONUS)

Conclusion



## Who am I? Axelle Apvrille

- ▶ Senior security researcher at Fortinet
- ▶ Topic: malware for smart devices (phones, IoT...)
- ▶ Email: aapvrille at fortinet dot com
- ▶ Twitter: @cryptax
- ▶ GPG: 5CE9 C366 AFB5 4556 E981 020F 9EAA 42A0 37EC 490C



Copy the contents of the USB key  
and pass to your neighbour!  
Thanks!

Please **bring the USB keys back**  
when finished

## Contents of the USB key

- ▶ **instructions:** slides, labs and a summary of commands/tools.
- ▶ **samples:** malicious Android samples we'll analyze in the lab.  
**Real viruses. Do not distribute, do not install on your phones!!!**
- ▶ **scripts-solutions:** spoilers ;)
- ▶ **vm(big):** VirtualBox images. **If your VM/Docker is already up and running, you don't need to copy this directory.**

## Two solutions: choose

**Requirements:** install either Docker or VirtualBox

Lab in a **Docker** container



[https://www.docker.com/  
products/overview](https://www.docker.com/products/overview)

You also need either **ssh** or  
**vncviewer**

Lab in a **VirtualBox** image



[https://www.virtualbox.org/  
wiki/Downloads](https://www.virtualbox.org/wiki/Downloads)

# Test your environment

- ▶ Check you can login with password **rootpass**
- ▶ Check you can view the contents of the USB key from within the container/image. Mount it on /data.
- ▶ Check you have many pre-installed tools in /opt
- ▶ Launch an Android emulator in the container/image:

In Docker

```
$ emulator5 &
```

In VirtualBox

```
$ emulator &
```



Get started

**Lab 1: Basics - Contents of an APK**

Lab 2: Static Analysis

Labs 3 and 4: Dynamic Analysis

Lab 5: Using Androguard

Lab 6: Working with Radare2

Lab 7: De-obfuscation

Labs 8 and 9: Unpacking Pangxie and LIAPP

Demo: Debugging an APK

Lab 10: Counter anti-emulator tricks (BONUS)

Conclusion

# What's an Android Package (APK)?

It is a Zip !

Taken from Android/Spitmo.C!tr.spy

```
$ unzip criptomovil.apk
Archive:  criptomovil.apk
  inflating: res/layout/main.xml
  inflating: AndroidManifest.xml
 extracting: resources.arsc
 extracting: res/drawable-hdpi/icon.png
 extracting: res/drawable-ldpi/icon.png
 extracting: res/drawable-mdpi/icon.png
  inflating: classes.dex
  inflating: META-INF/MANIFEST.MF
  inflating: META-INF/CERT.SF
  inflating: META-INF/CERT.RSA
```





- ▶ Dalvik executable: `classes.dex`.



- ▶ Dalvik executable: `classes.dex`.
- ▶ Resources: images, layouts, localized strings: `./res/*`



- ▶ Dalvik executable: `classes.dex`.
- ▶ Resources: images, layouts, localized strings: `./res/*`
- ▶ Assets: more or less the same as raw resources, but not accessed with the same API. `./assets`



- ▶ Dalvik executable: `classes.dex`.
- ▶ Resources: images, layouts, localized strings: `./res/*`
- ▶ Assets: more or less the same as raw resources, but not accessed with the same API. `./assets`
- ▶ Lib: external libraries.

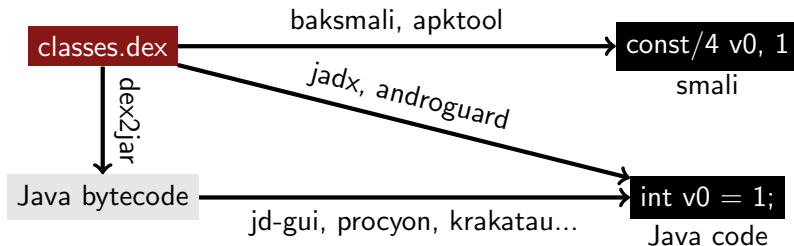


- ▶ Dalvik executable: `classes.dex`.
- ▶ Resources: images, layouts, localized strings: `./res/*`
- ▶ Assets: more or less the same as raw resources, but not accessed with the same API. `./assets`
- ▶ Lib: external libraries.
- ▶ `AndroidManifest.xml`: info about the application.



- ▶ Dalvik executable: `classes.dex`.
- ▶ Resources: images, layouts, localized strings: `./res/*`
- ▶ Assets: more or less the same as raw resources, but not accessed with the same API. `./assets`
- ▶ Lib: external libraries.
- ▶ `AndroidManifest.xml`: info about the application.
- ▶ `META-INF`: generated when signing the package.

# Dalvik Executables (.dex)



- ▶ **Dalvik Ex**executable (DEX): similar to .class for Java
- ▶ **smali** means assembler in icelandic



Apktool - directly from apk

```
$ java -jar apktool.jar d YOURPACKAGE.apk -o OUTPUTDIR
```

- ▶ **d** is for **d**ecode
- ▶ Also converts Android manifest and resources to readable form
- ▶ In the VM / container: /opt/apktool/apktool.jar



Androdd

```
$ androdd -i classes.dex -o output
```

Androlyze from classes.dex

```
$ androlyze -s  
d, dx = AnalyzeDex("classes.dex")  
d.create_python_export()
```

Androlyze from apk

```
$ androlyze -s  
a, d, dx = AnalyzeAPK('sample.apk')  
d.CLASS_xxxx.METHOD_yyy.pretty_show()
```

Androlyze is the Androguard interactive Python shell



- ▶ **Baksmali**: `java -jar baksmali.jar -o output-dir classes.dex`
- ▶ **IDA Pro**
- ▶ **Radare2**: `r2 classes.dex`



AdminService class, inheriting from Service.  
Source file name is missing:

Class header

```
.class public AdminService
.super Service
.source ""
```



```
.method static <clinit>()V
    .registers 1
    const/4        v0, 0
    sput-object    v0, AdminService->c0Ic00o:Thread
    return-void
.end method
```

- ▶ Dalvik is **register** based, not *stack* based
- ▶ ()V: Java signatures for methods: V for void, B for byte, Z for boolean...
- ▶ Dalvik instructions: const/4, sput-object...

```
.method public constructor <init>(I)V  
    .registers 4  
    .param p1, "initialCapacity"
```

- ▶ p0 is for *this*, p1 is first argument of method
- ▶ naming is not always provided!

## Calls

```
invoke-virtual {v0, v1, p1}, L.../TinyDB;  
    ->putInt(Ljava/lang/String;I)V
```

Means: `this.putInt(v1, p1);`

# Want to read Java source code? Use a decompiler!

- ▶ [Androguard](#) embeds a good decompiler.

```
a, d, dx = AnalyzeAPK('sample.apk', decompiler='dad')  
d.CLASS_xxxx.METHOD_yyy.source()
```

- ▶ [JADX](#): `jadx -d output-dir classes.dex`
- ▶ [JEB Decompiler](#): not free - but excellent. Trial version exists.
- ▶ Two step solution:
  1. Convert to **jar** using [dex2jar](#): `d2j-dex2jar.sh classes.dex`
  2. Then use a Java decompiler e.g [JD](#)...

# Decompiled Java source code - at a glance

**Classes, fields, and methods**

**Dummy variable names**

**Bad class: localObject2 is a TelephonyManager**

```
File Edit Navigate Search Help
com.antivirus.kav
├── KavService
├── MainActivity
├── R
└── SmsReceiver
    ├── SmsReceiver
    ├── AppContext
    ├── FirstSchedule
    ├── SettingHideS
    ├── UriToReport
    ├── FireGetReque
    ├── GetBoolValue
    ├── GetRequest
    ├── GetStaticDat
    ├── LinkAntivirus
    ├── SaveBoolValu
    └── onReceiveCc

SmsReceiver.class

public static String GetStaticDataString(Context paramContext)
{
    Object localObject2 = ((TelephonyManager)paramContext.getSystemService("phone")).getLine1Number();
    Object localObject1 = ((TelephonyManager)localObject2).getLine1Number();
    String str1 = ((TelephonyManager)localObject2).getDeviceId();
    String str2 = ((TelephonyManager)localObject2).getDeviceId();
    String str3 = "empty";
    if (str2 != null)
    {
        str3 = Integer.toString(2 * Integer.parseInt(str2.substring(8)));
        str3 = "1" + str3 + "3";
    }
    else
    {
        str2 = "empty";
    }
    if (localObject1 != null)
        localObject2 = ((String)localObject1).replace("+", "");
    else
        localObject2 = "empty";
    if (str1 == null)
        str1 = "empty";
    int i = 0;
    if (GetBoolValue(paramContext, "AntivirusEnabled"))
        i = 1;
    localObject1 = new Object[5];
    localObject1[0] = localObject2;
    localObject1[1] = str1;
    localObject1[2] = str2;
    localObject1[3] = str3;
    localObject1[4] = Integer.valueOf(i);
    return (String)(String)String.format("?to=%s&l=%s&n=%s&id=%s&h=%s", localObject1);
}

public static String LinkAntivirus()
{
}
```

## Cross references: who's using this method/field?

- ▶ Good news: smali are text files. You can **grep** etc.
- ▶ **Androguard**: `show_xref()`, `show_dref()`
- ▶ **JEB**: Ctrl-X
- ▶ **Radare**: `axt`, `axf`

### Beware

Inheritance, interfaces, events “break” the call tree :(



Taken from Android/Spitmo.C!tr.spy

- Identify the main entry point

```
<activity android:label="@7F040001" android:name=".MainActivity">  
  
    <intent-filter>  
  
        <action android:name="android.intent.action.MAIN">  
  
        </action>  
  
        <category android:name="android.intent.category.LAUNCHER">  
  
        </category>
```



Taken from Android/Spitmo.C!tr.spy

- ▶ Identify the main entry point
- ▶ Background services

```
<service android:enabled="true" android:name=".KavService">  
  
</service>
```

# Understanding the Android manifest

## Taken from Android/Spitmo.C!tr.spy

- ▶ Identify the main entry point
- ▶ Background services
- ▶ Receivers: called when events occur

```
<receiver android:name=".SmsReceiver">

    <intent-filter android:priority="999999">

        <action android:name="android.provider.Telephony.SMS_RECEIVED">

        </action>

        <action android:name="android.intent.action.NEW_OUTGOING_CALL">

        </action>

        <action android:name="android.intent.action.BOOT_COMPLETED">

        </action>

    </intent-filter>

</receiver>
```



Taken from Android/Spitmo.C!tr.spy

- ▶ Identify the main entry point
- ▶ Background services
- ▶ Receivers: called when events occur
- ▶ Permissions

```
<uses-permission android:name="android.permission.READ_SMS">
```

```
</uses-permission>
```

```
<uses-permission android:name="android.permission.RECEIVE_SMS">
```

## How to reverse Android malware ?



① First glance matters

- Are they trying to hide something?
- What's the name of the package?
- What does the certificate say?
- Where did I find it?

② Disassemble it

const-v  
get i0

The code says it all!  
Don't be lazy  and read it in Android depth.

③ Still don't understand?

Run it in an emulator, display logs and capture network traffic.



Never use your own phone.  
Do not provide any personal data  
(name, IMEI, phone number...)



*THE CODE DOES NOT  
MAKE SENSE ?*

Maybe it's heavily obfuscated or packed.

*THERE'S NOTHING SUSPICIOUS ?!*

Good  Check the assets and resources directory for Javascript or ARM executables.

@cryptax  
2016



- ▶ **Androguard**: in the path
- ▶ **Apktool**: /opt/apktool/apktool.jar
- ▶ **AXMLPrinter from rednaga**: java -jar axmlprinter-0.1.7.jar
- ▶ **Baksmali/smali**: /opt/baksmali.jar, /opt/smali.jar
- ▶ **CFR**: /opt/cfr\_0\_118.jar
- ▶ **ClassyShark**: /opt/ClassyShark.jar
- ▶ **Dedexer** produces .ddx files  $\approx$  **Jasmin** w/ Dalvik opcodes



- ▶ [DED](#) Decompiler or [Dare](#)
- ▶ [dex2jar](#): in the path
- ▶ [DroidSec Links](#)
- ▶ [JEB Decompiler](#)
- ▶ [Krakatau](#): `/opt/Krakatau/disassemble.py`
- ▶ [Procyon](#): `/opt/procyon-decompiler.jar`
- ▶ [JD](#): `/opt/jd-gui.jar`

# Lab 1: Time to Work!!!



It's a training, time for **you** to work :=)

Samples are located in **/data**

Tools are located in **/opt** (and subdirectories)

You have a work dir in **/workshop**

Password: **rootpass**





Get started

Lab 1: Basics - Contents of an APK

**Lab 2: Static Analysis**

Labs 3 and 4: Dynamic Analysis

Lab 5: Using Androguard

Lab 6: Working with Radare2

Lab 7: De-obfuscation

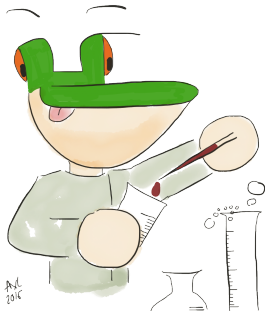
Labs 8 and 9: Unpacking Pangxie and LIAPP

Demo: Debugging an APK

Lab 10: Counter anti-emulator tricks (BONUS)

Conclusion

## Lab 2: Static analysis of Android/SpyBanker





Get started

Lab 1: Basics - Contents of an APK

Lab 2: Static Analysis

**Labs 3 and 4: Dynamic Analysis**

Lab 5: Using Androguard

Lab 6: Working with Radare2

Lab 7: De-obfuscation

Labs 8 and 9: Unpacking Pangxie and LIAPP

Demo: Debugging an APK

Lab 10: Counter anti-emulator tricks (BONUS)

Conclusion

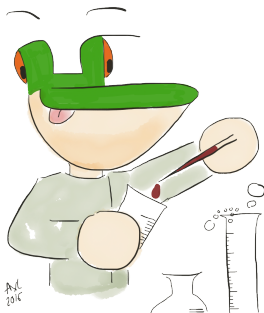
## Modify the smali code

1. Get the smali e.g. with Baksmali
2. Modify the smali
3. Compile the smali to DEX: `java -jar /opt/smali.jar a ./smali/`
4. Zip the DEX with resources: `zip -r ...`
5. Sign it (if necessary create keys before): `jarsigner -keystore test.ks repackaged.apk test`

## Patch to insert logs

```
const-string v0, "Hello there"
const-string v6, "MY TAG: "
invoke-static {v6, v0},
    Landroid/util/Log;->v(Ljava/lang/String;
    Ljava/lang/String;)I
```

## Lab 3: Patching a package



Many AV vendors prohibit malware patching because this creates another malware.  
Do not distribute!



### General advice for Dynamic Analysis

- ▶ Make sure you won't be sending data to the malware authors
- ▶ Some malware perform anti-emulator tricks



Get started

Lab 1: Basics - Contents of an APK

Lab 2: Static Analysis

Labs 3 and 4: Dynamic Analysis

**Lab 5: Using Androguard**

Lab 6: Working with Radare2

Lab 7: De-obfuscation

Labs 8 and 9: Unpacking Pangxie and LIAPP

Demo: Debugging an APK

Lab 10: Counter anti-emulator tricks (BONUS)

Conclusion



## Androguard Home

*Already installed in your Docker container / VirtualBox image*

### RE with Androguard

```
$ androlyze -s  
In [2]: a, d, dx = AnalyzeAPK('your.apk', decompiler='dad')
```

It's a Python interactive shell. The usual Python tricks work:

- ▶ Use **the Tab key** for completion
- ▶ Documentation: `print xxxx.__doc__`
- ▶ History





```
a, d, dx = AnalyzeAPK('your.apk', decompiler='dad')
```

```
a - androguard.core.bytecodes.apk.APK
```

- ▶ a.get\_main\_activity()
- ▶ a.get\_receivers()
- ▶ a.get\_services()
- ▶ a.get\_certificate()
- ▶ ...



`d - androguard.core.bytecodes.dvm.DalvikVMFormat`

- ▶ All classes are named `d.CLASS_foo`
- ▶ All methods are named `d.CLASS_foo.METHOD_bar`
- ▶ All fields are named `d.CLASS_foo.FIELD_blah`
- ▶ Smali: `d.CLASS_foo.METHOD_bar.pretty_show()`
- ▶ Decompiled code: `d.CLASS_foo.METHOD_bar.source()`
- ▶ Method cross references:  
`d.CLASS_foo.METHOD_bar.show_xref()`
- ▶ Field cross references:  
`d.CLASS_foo.FIELD_blah.show_dref()`



## Complex operations - dx

Class name:

`androguard.core.analysis.analysis.uVMAnalysis`

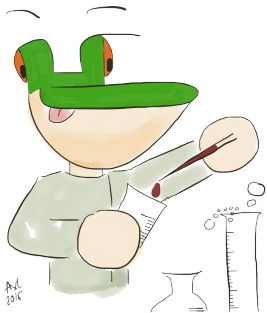
- ▶ List used permissions: `show_Permissions(dx)`
- ▶ Show where dynamic code is used: `show_DynCode(dx)`

## Searching

- ▶ Search for a given string: `filter(lambda x: 'YOUR STRING' in x, d.get_strings())`
- ▶ Show where a string is used:  

```
z = dx.tainted_variables.get_string('YOUR STRING')  
z.show_paths(d)
```

Use Androguard on this malware





Get started

Lab 1: Basics - Contents of an APK

Lab 2: Static Analysis

Labs 3 and 4: Dynamic Analysis

Lab 5: Using Androguard

**Lab 6: Working with Radare2**

Lab 7: De-obfuscation

Labs 8 and 9: Unpacking Pangxie and LIAPP

Demo: Debugging an APK

Lab 10: Counter anti-emulator tricks (BONUS)

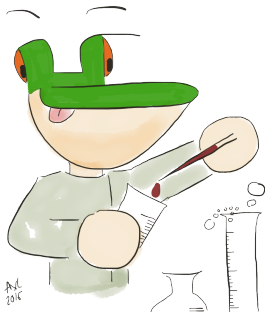
Conclusion



<http://www.radare.org>

- ▶ It works on the classes.dex. Automatic detection of Dalvik. (If not, use `r2 -a dalvik file`).
- ▶ List classes, methods and fields: `ic`, or list functions: `afl`
- ▶ List imports: `ii`
- ▶ List strings: `iz` (method names in there too)
- ▶ Cross references: `axt` (references TO this address) or `axf` (from)
- ▶ Search for string `http`: `f~http` or `/ http`
- ▶ Disassemble: `pd LINES @ ADDR`

## Lab 6: Disassembling Android/Crosate with Radare2





Get started

Lab 1: Basics - Contents of an APK

Lab 2: Static Analysis

Labs 3 and 4: Dynamic Analysis

Lab 5: Using Androguard

Lab 6: Working with Radare2

**Lab 7: De-obfuscation**

Labs 8 and 9: Unpacking Pangxie and LIAPP

Demo: Debugging an APK

Lab 10: Counter anti-emulator tricks (BONUS)

Conclusion



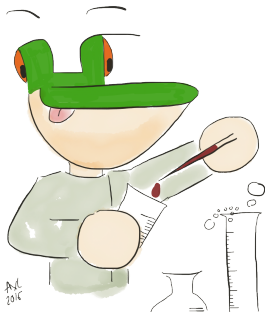


- ▶ **Obfuscators.** Generic term. Proguard, Dexguard, Allatori,
- ▶ **Protectors.** e.g. anti-debugging, anti-emulator techniques  
ApkProtect
- ▶ **Packers.** Executable 'compressor'. Decompression stub  
decompresses sample *in place* (dump memory) or *on disk*  
(inspect /data/data for example). Pangxie, LIAPP, Bangle



- ▶ Identify packers [APKiD](#)
- ▶ Decrypt strings [d2j-decrypt-string.sh](#)
- ▶ Unpacking: [DexHunter](#), [kisskiss](#)
- ▶ [Simplify](#)
- ▶ [JEB](#) or [JEB2](#) scripts
- ▶ Debugging applications: [CodeInspect](#) or [JEB2](#)

## Lab 7: De-obfuscating Obad strings





Get started

Lab 1: Basics - Contents of an APK

Lab 2: Static Analysis

Labs 3 and 4: Dynamic Analysis

Lab 5: Using Androguard

Lab 6: Working with Radare2

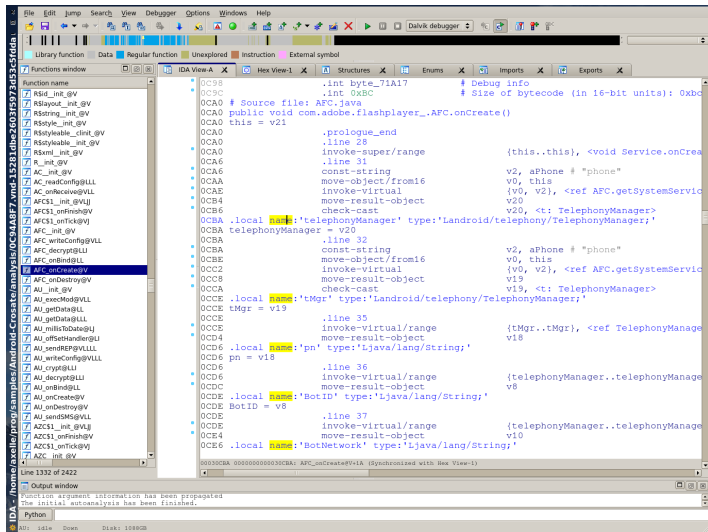
Lab 7: De-obfuscation

**Labs 8 and 9: Unpacking Pangxie and LIAPP**

Demo: Debugging an APK

Lab 10: Counter anti-emulator tricks (BONUS)

Conclusion





Get started

Lab 1: Basics - Contents of an APK

Lab 2: Static Analysis

Labs 3 and 4: Dynamic Analysis

Lab 5: Using Androguard

Lab 6: Working with Radare2

Lab 7: De-obfuscation

Labs 8 and 9: Unpacking Pangxie and LIAPP

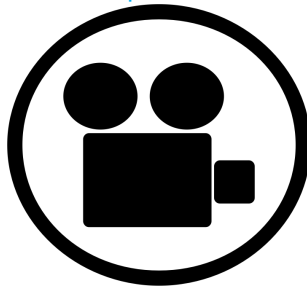
**Demo: Debugging an APK**

Lab 10: Counter anti-emulator tricks (BONUS)

Conclusion



CodeInspect or JEB2





Get started

Lab 1: Basics - Contents of an APK

Lab 2: Static Analysis

Labs 3 and 4: Dynamic Analysis

Lab 5: Using Androguard

Lab 6: Working with Radare2

Lab 7: De-obfuscation

Labs 8 and 9: Unpacking Pangxie and LIAPP

Demo: Debugging an APK

Lab 10: Counter anti-emulator tricks (BONUS)

Conclusion



## IMEI

On emulator, IMEI default value is 0000000000000000.

Very common check in malware.

Get the value:

- ▶ Program: `getDeviceId()`
- ▶ Emulator < Android 5: `adb shell dumpsys iphonesubinfo`
- ▶ Emulator  $\geq$  Android 5: `adb shell service call iphonesubinfo : code`  
5.1.1: `code = 1`

Set the value: search for **+CGSN**

```
00370320 53 3d 00 2b 43 49 4d 49 00 33 31 30 32 36 30 30 |S=+.CIMI.3102600|
00370330 30 30 30 30 30 30 30 30 00 2b 43 47 53 4e 00 30 |00000000+.CGSN.0|
00370340 30 30 30 30 30 30 30 30 30 30 30 30 30 00 2b |0000000000000000.+|
```

## IMSI

Get the value:

- ▶ Program: `getSubscriberId()`
- ▶ Emulator: same as IMEI, except service code is 7 (Android 5.1.1).

Set the value: search for **+CIMI**

## Geographic location

Common especially in Adware e.g. Adware/Feiwo (2016)

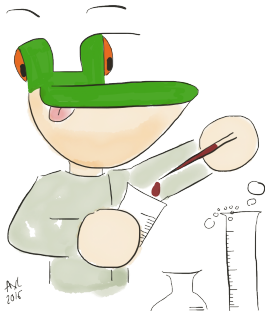
- ▶ Set the value: `adb emu geo fix longitude latitude altitude`
- ▶ Get the value: `adb shell dumpsys location ?` (does not work on emulator)



### Phone number

- ▶ default: 15555215554
- ▶ program: `getLine1Number()`
- ▶ get value on Android 5.1.1: `adb shell service call iphonesubinfo 13`
- ▶ set value: `emulator -port` changes the last 4 numbers (5554 and 5584), or patch source, or use `genymotion`...

## Lab 10: Patching the emulator (BONUS)





Get started

Lab 1: Basics - Contents of an APK

Lab 2: Static Analysis

Labs 3 and 4: Dynamic Analysis

Lab 5: Using Androguard

Lab 6: Working with Radare2

Lab 7: De-obfuscation

Labs 8 and 9: Unpacking Pangxie and LIAPP

Demo: Debugging an APK

Lab 10: Counter anti-emulator tricks (BONUS)

Conclusion



- ▶ Dalvik Opcodes
- ▶ Collection of Android tools
- ▶ Using Androguard for RE
- ▶ Emacs smali mode: Tim Strazzere
- ▶ Obfuscation in Android malware and to fight back
- ▶ Android App “Protection”
- ▶ Fortiguard Research Publications

Thank you for attending!  
Please bring the USB keys back :)



Like the slides? Thanks. This is  $\text{\LaTeX}$ .