13. Mobile Security and Malicious Apps

Computer Security Courses @ POLIMI

Prof. Carminati & Prof. Zanero

Smartphone as a Target

- Always online
- Ample computing resources available
- Handles sensitive data
  - Email
  - Social networks
  - Online banking
  - Current location

Mobile vs. Desktop

	Desktop	**Android**	**iOS**
Operating System	Win/Linux/Mac	Linux based	Darwin based
Processor Architecture	x86	ARM/x86/...	ARM
Programming Language	arbitrary	Java/native code	Objective C
Accessibility	open	open	closed

Security Model in a Nutshell

iOS

- code signing
- sandboxing with predefined permissions
- closed ecosystem (App Store is the CA)

Android

- code signing checked only at installation
- sandboxing with explicit permissions
- open ecosystem (no PKI, no CA)

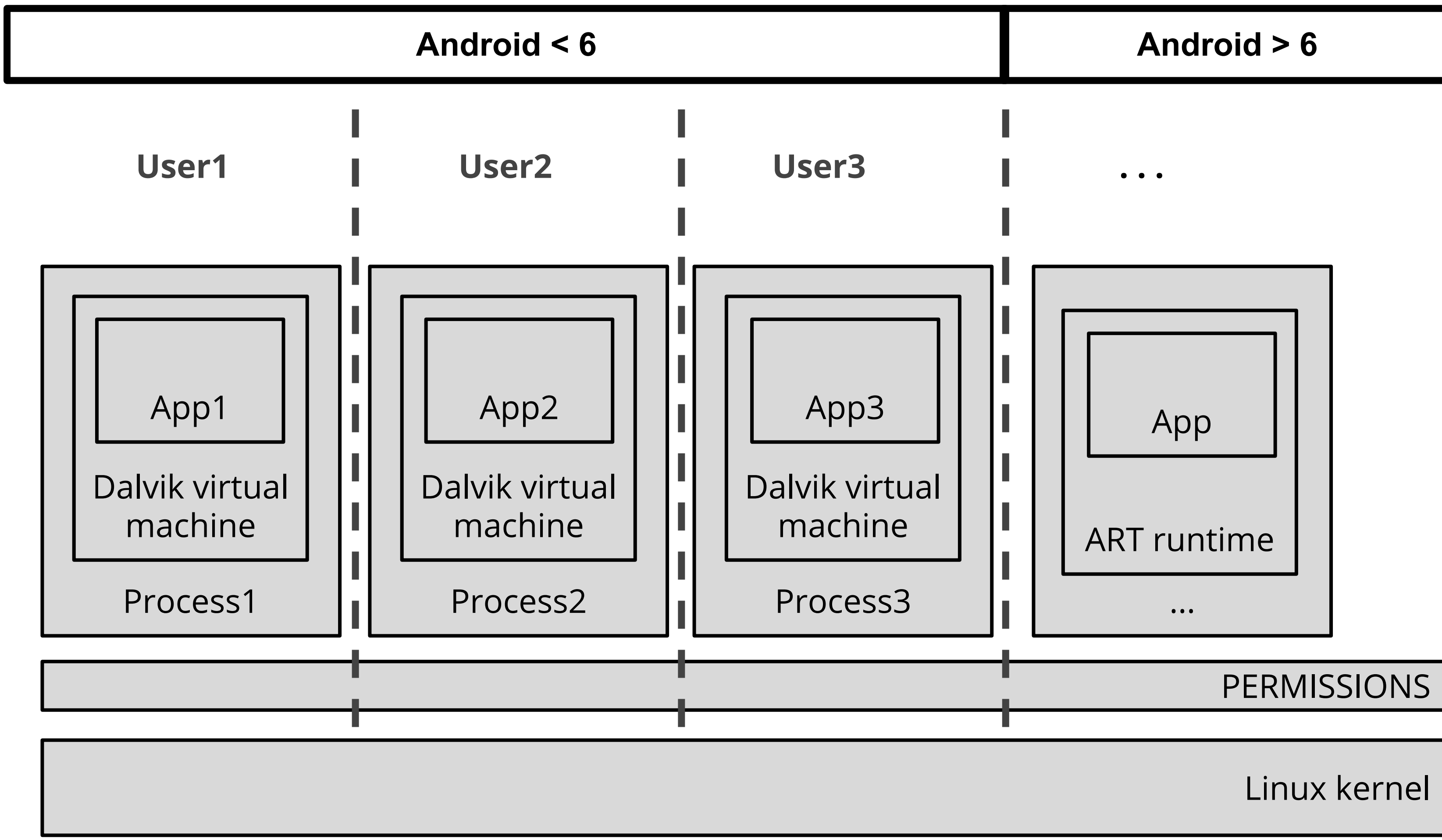
Sandboxing

iOS: the kernel uses MAC to restrict the files and memory space that each app can access.

Android: each app is assigned a distinct user and Linux takes care of the privilege separation automatically.

Result: in both cases, apps cannot interact with each other unless authorized by the kernel.

Android Sandboxing



Permission-based Authorization

iOS: use "privacy profiles" to define access control rules (e.g., Safari can access the Address Book). User must confirm at first use. Default deny.

Android: each app requests explicit permissions that the user must approve, or the app won't install. Meh…

- Now it's a little different, more like iOS

Android Permissions

Declared in a manifest file

```
<uses-permission ="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission ="android.permission.READ_LOGS" />
<uses-permission ="android.permission.WAKE_LOCK" />
<uses-permission ="android.permission.READ_PHONE_STATE" />
<uses-permission ="android.permission.PROCESS_OUTGOING_CALLS" />
<uses-permission ="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission ="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission ="android.permission.ACCESS_WIFI_STATE" />
<uses-permission ="android.permission.CHANGE_WIFI_STATE" />
<uses-permission ="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission ="android.permission.CHANGE_NETWORK_STATE" />
<uses-permission ="android.permission.MODIFY_PHONE_STATE" />
<uses-permission ="android.permission.WRITE_SECURE_SETTINGS" />
<uses-permission ="android.permission.WRITE_SETTINGS" />
<uses-permission ="android.permission.INTERNET" />
<uses-permission ="android.permission.BLUETOOTH" />
```

Code Signing: iOS

- The kernel enforces that only signed code is executed.
  - app code is signed with the developer's key
  - the developer's certificate is signed by Apple's CA
  - no external code is allowed in the ecosystem
- No dynamically generated code, no JIT (except for JavaScript).
- No traditional shellcode.
  - more advanced techniques (e.g., return-oriented programming) are needed for exploitation.
- <u>Sandbox ensures last line of protection.</u>

Code "Signing": Android

- The installer checks that every new application is cryptographically valid.
  - Apps are verified only upon installation
    - not each time they are run
  - Apps can be signed with self-signed certificate
  - There is no PKI
- Dynamically-generated code is allowed.
  - e.g., **DexClassLoad** API to load external JAR files
  - actually useful (silent upgrades)
- Sandbox ensures last line of protection.

Breaking the Rules for Extra Apps

iOS "jailbreaking"

- exploit a kernel- or driver-level vulnerability
- modify the OS to allow "other" apps
- install extra store managers (e.g., Cydia)
- not straightforward for regular users

Android ~~"rooting"~~ (not necessary)

- enable "Allow from unknown sources" setting
- done.

Malicious Code: Requirements

iOS: needs jailbroken target OR App Store approval (i.e., manual checks).

Android: needs "Allow from external sources" enabled OR Google Play Store approval (i.e., automatic checks). Ask permissions.

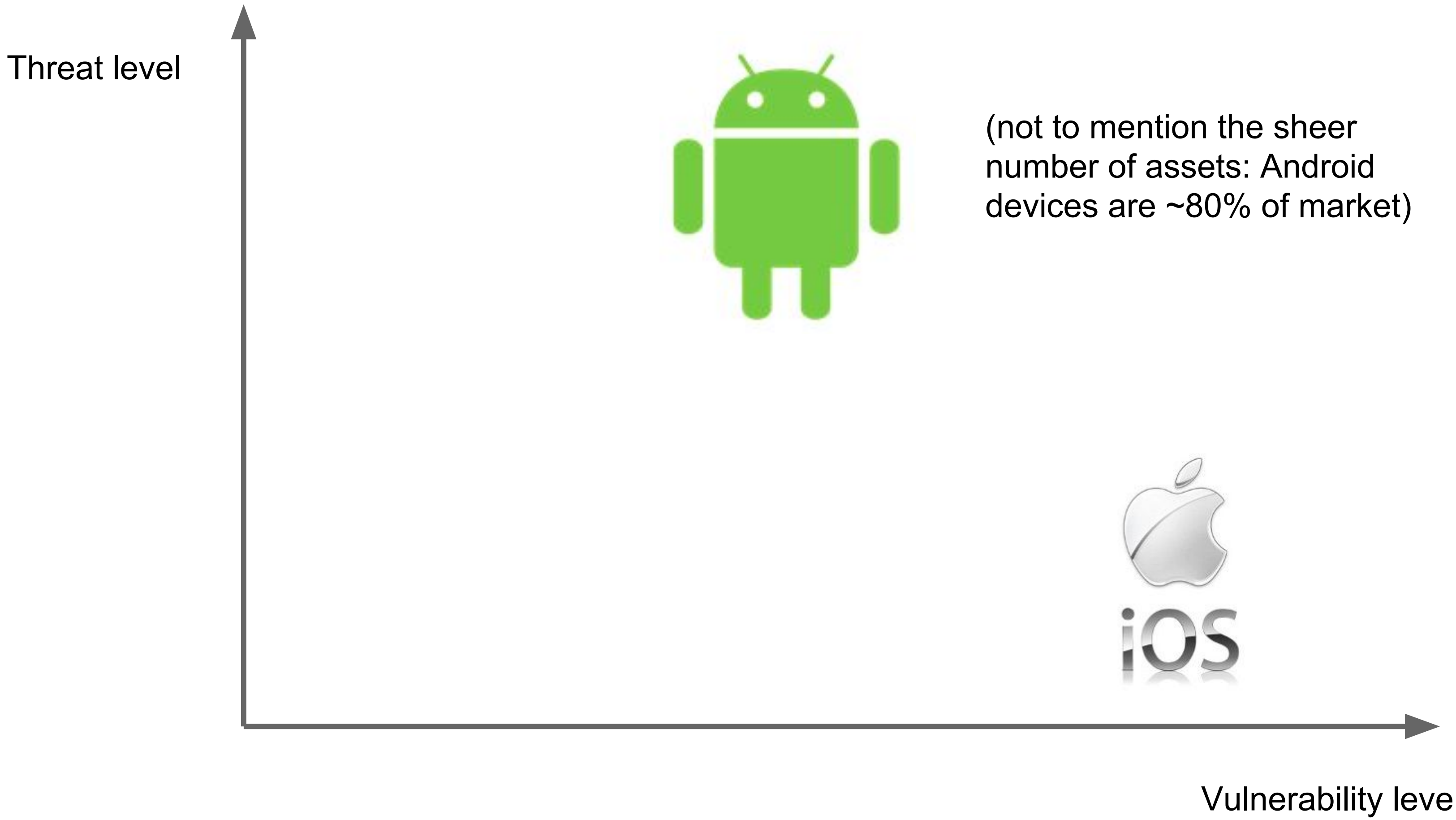
Guess what?

Risk = threat x vuln x asset

Consider these 4 empirical facts:

- The *open model* of Android makes it attractive for money-motivated attackers.
- The *closed model* of iOS makes it unattractive for money-motivated attackers.
- Android has *75% of the market*: this makes it attractive for attackers.
- Android has *less vulnerabilities* than iOS.

Threat vs. vulnerability



Result?

~99% of the malware is written to target Android devices.

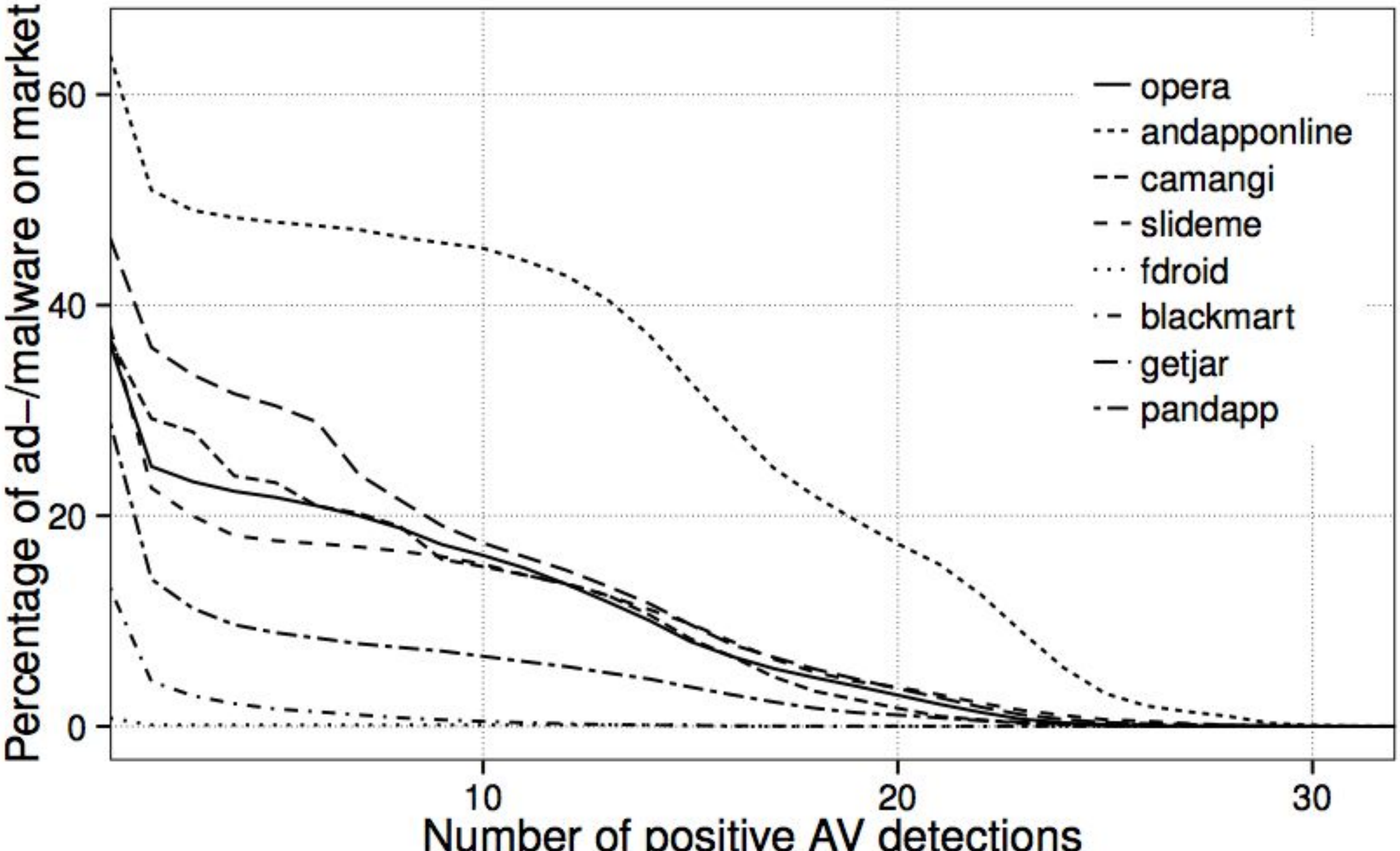
Malicious Code: iOS vs Android

iOS: 10–15 malware families found. Very few samples. Only 2 in the App Store.

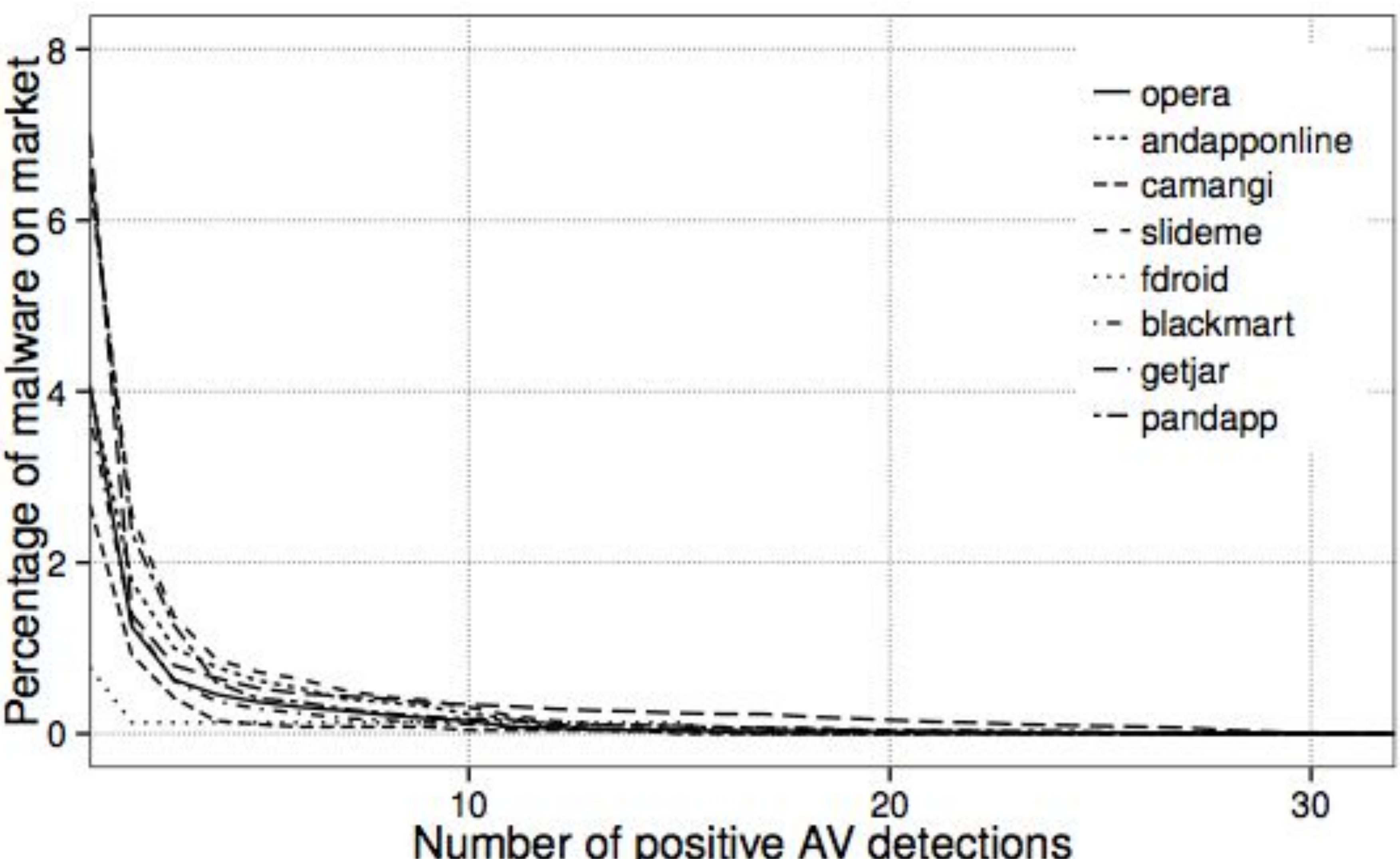
Android: >= 50 malware families.

- 200–500K samples
- 0.28% infected devices
- A dozen cases in the Google Play Store
- about ~100 alternative marketplaces (e.g., apptoide, slideme, andapponline) full of malicious apps.

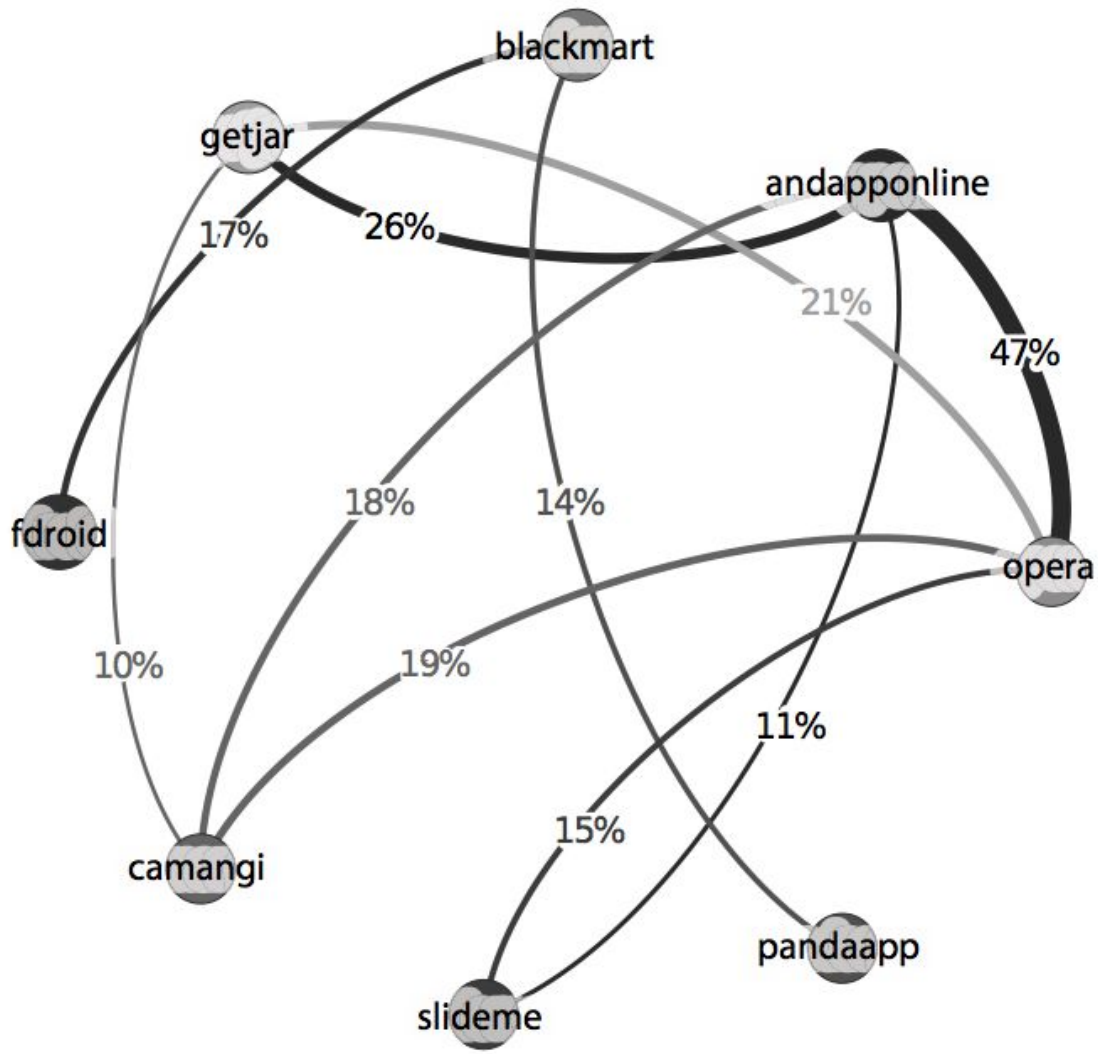
Adware/Malware in Marketplaces



Malware in Marketplaces



Overlapping Apps in Marketplaces

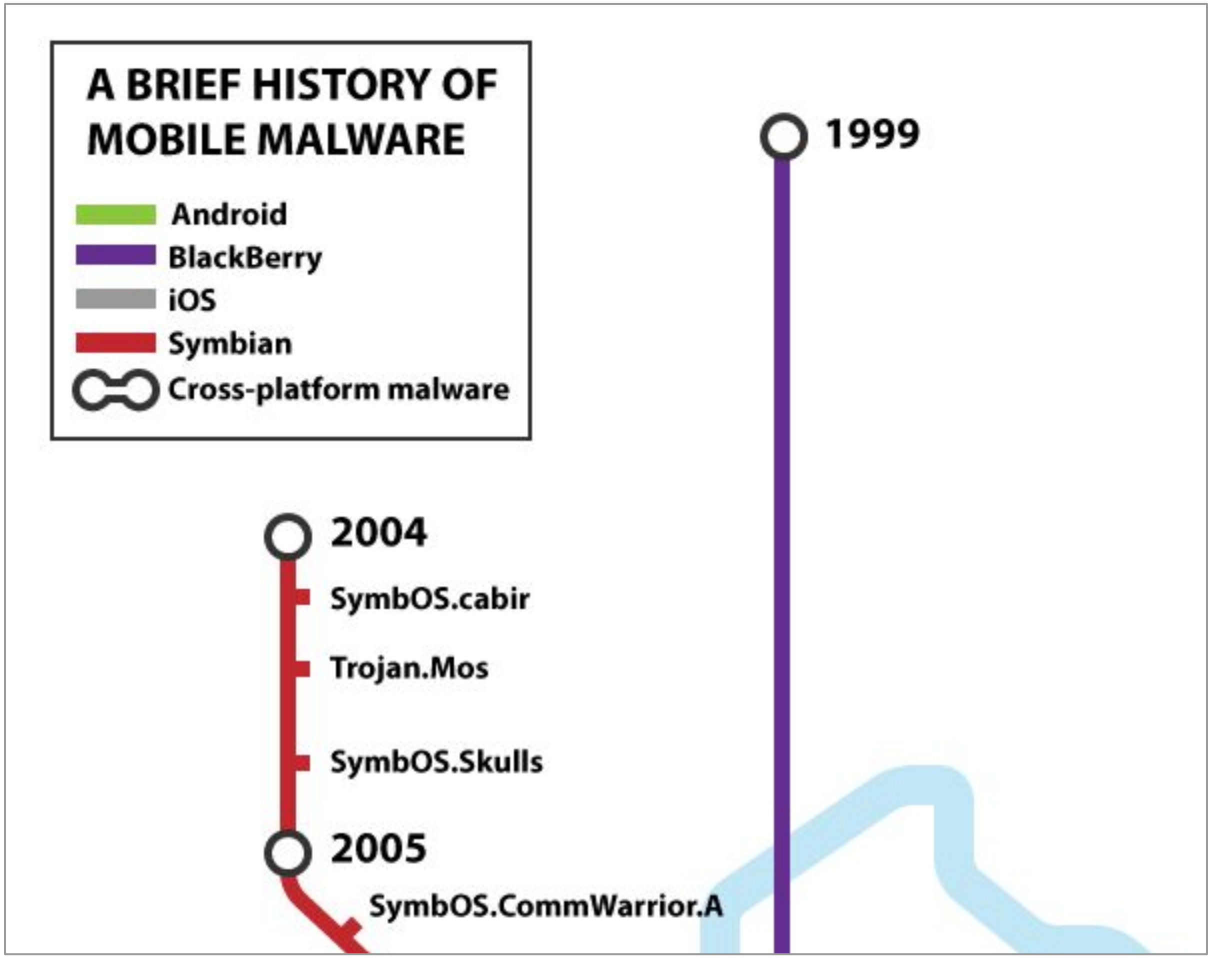


Malicious Code: Actions

Sandbox restrictions ~> creativity

- call or text to premium numbers ($$$)
- silently visit web pages
    - boost ranking ($$$)
- steal sensitive information
    - mobile banking credentials ($$$)
    - contacts, email addresses (re-sell, and $$$)
- root-exploit the device (Android only, so far)
- turn the device into a bot (re-sell, and $$$)
- lock the device and ask for a ransom ($$$)

Cabir: Bluetooth-based Worms



Cabir (2004)
- Propagates to nearby devices using bluetooth
- Several variants
- Steals phonebook

Mos (2004)
- Premium texting: 1st case!

Skulls (2004)
- Damage the system
- Remove or replaces apps

CommWarrior (2005)
- Propagates via MMS
- Steals phonebook

http://www.symantec.com/connect/blogs/tenth-anniversary-mobile-malware

History Continued

RedBrowser (2006)

- first J2ME malware
- premium texting

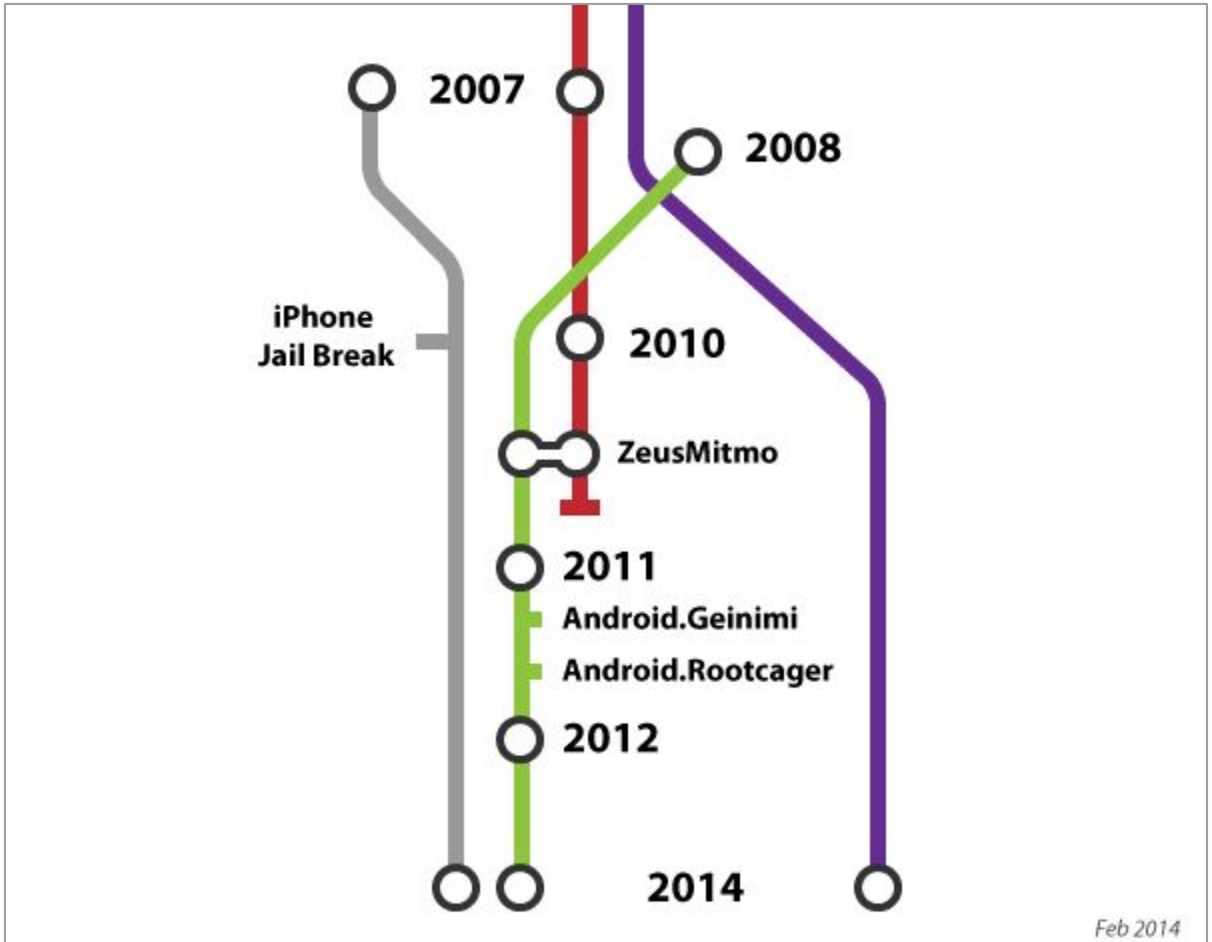
FlexiSpy (2006)

- commercial application
- advertised as "best solution for people who wanted to spy on their spouses"

ZeusMitmo, ZitMo (2010)

- steal second factor of authentication
- works in tandem with desktop ZeuS
- marks the beginning of profit driven mobile malware

Android Malware Outbreak



TapSnake (2010)

- steals location
- embedded in a snake-like game app

Geinimi (2011)

- steals sensitive information

Rootcager (2011)

- 1st case of root-level exploit found in mobile malware

"Recent" Statistics (2012–2014)

[Zhou, 2012]:

- 36.7% leverage root-level exploits
- 90% turn devices into bots
- 45.3% dial/text premium numbers in background
- 51.1% harvest user information

2013–2014:

- 3,492 out of over 380 million (0.0009%) devices in the US infected (lower bound)
- 154 out of over 55,000 (0.26%) worldwide

Mitigating (Mobile) Malware

- Google Play runs automated checks
  - Dissecting the Android Bouncer
- SMS/call blacklisting and quota
  - only in custom ROMs (e.g., CyanogenMod >= 10.4)
- Google App Verify (call home when apps are installed)
  - Google uses VirusTotal to check if known malware
- App sandboxing
  - Limit the privileges of an app
  - Useless against root-level exploits
- SELinux

Counteracting (Mobile) Malware

Ex-post workflow:

1. suspicious app reported by "someone"
2. automatically analyzed
3. manually analyzed
4. app removed from the (official) store
5. antivirus signature developed

Automated App Analysis

Static analysis

- parse the application (binary or byte)code
- pros and cons
  - +code coverage, dormant code
  - -obfuscation, encryption, packing

Dynamic analysis

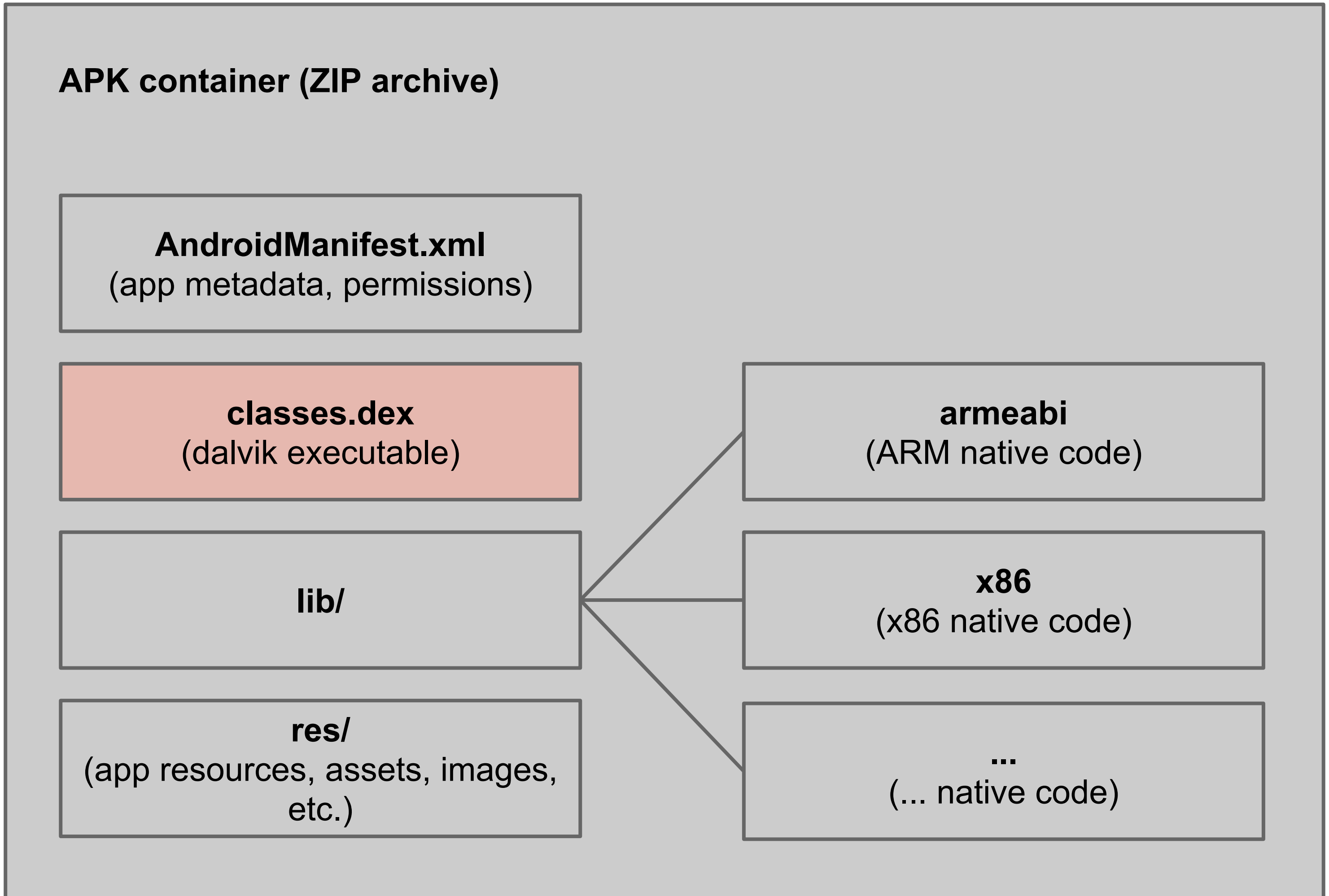
- observe the runtime behavior of an app
- pros and cons
  - -code coverage, dormant code
  - +obfuscation, encryption, packing

Static Analysis of Android Apps

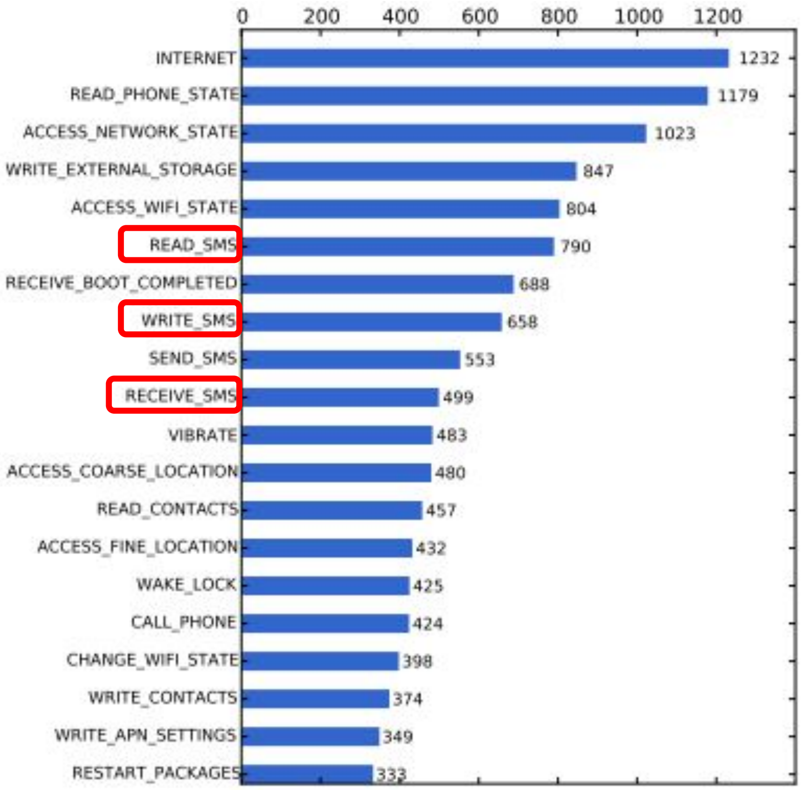
1. Parse the metadata (e.g., permissions)
2. Parse the bytecode and the native code
3. Reconstruct the control-flow graph
4. Statically determine suspicious structural components

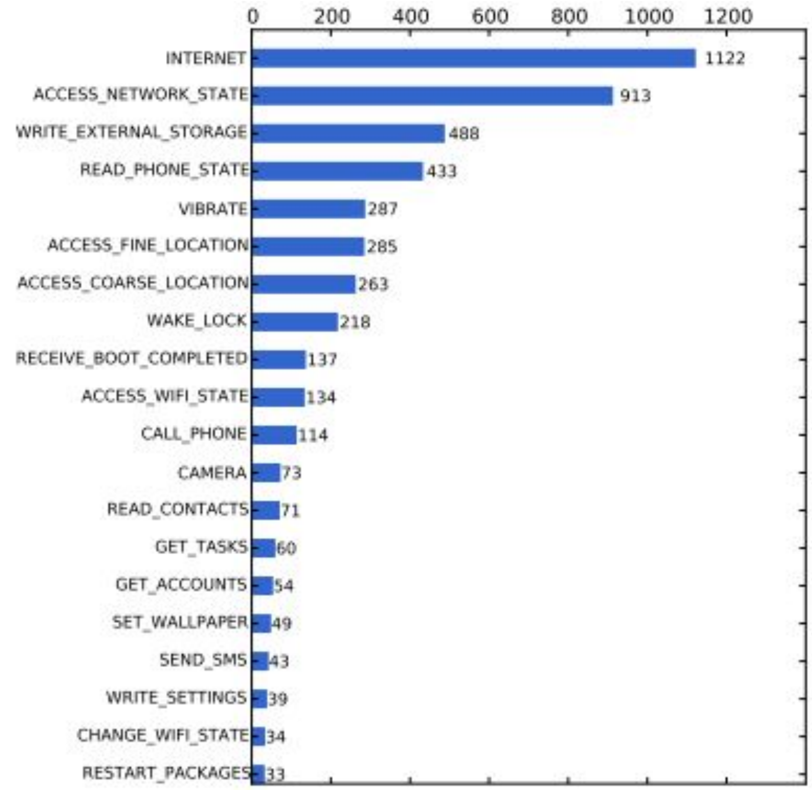
Example: code path from an SMS-related system call to a network socket system call).

Disassembling Android Apps



Example (permission analysis)





Source: Y. Zhou and X. Jiang, “Dissecting Android Malware: Characterization and Evolution,” IEEE SSP, 2012, pp. 95–109.

Example (real malware, simplified)

```
invoke-virtual/range {v21 .. v21}, Utils;->getDeviceId()Ljava/lang/String;

invoke-virtual {v13, v4, v5},
Lorg/json/JSONObject;->put(Ljava/lang/String;Ljava/lang/Object;)

invoke-virtual {v0, v4, v1}, Utils;->sendPostRequest(Ljava/lang/String;Ljava/lang/String;)V
```

Dalvik assembly example

- retrieves the device ID (IMEI)
- put it into a JSON object
- make an HTTP POST request with the JSON object

Result: IMEI stolen.

Further Reading (static analysis)

Arp et al., *"Drebin: Effective and Explainable Detection of Android Malware in Your Pocket"*, NDSS 2014

Gorla et al., *"Checking App Behavior Against App Descriptions"*, ICSE 2014

Andronio et al., *"HelDroid: Dissecting and Detecting Mobile Ransomware"*, RAID 2016

Dynamic Analysis of Android Apps

1. Create emulated environment
2. Launch the (suspicious) app
3. Record the actions performed
   a. filesystem events
   b. network traffic
   c. API function calls
   d. SMS sent
4. Create a report of the activity
5. Develop heuristics to find signs of malicious actions (e.g., send an SMS right after receiving one)

Example (same real malware)

- Network operations

Timestamp	Operation	Host	Port
114.949	open	hosted-by.leaseweb.com	80
115.949	open	hosted-by.leaseweb.com	80
115.949	open	hosted-by.leaseweb.com	80
115.949	open	hosted-by.leaseweb.com	80
115.949	open	hosted-by.leaseweb.com	80
117.949	write	hosted-by.leaseweb.com	80
POST /command/update_incoming_patterns HTTP/1.1 User-Agent: 8b65916051836d5cfd41946e79d14316 Content-Length: 0 Content-Type: text/plain; charset=UTF-8 Host: [redacted] Connection: Keep-Alive			
118.949	write	hosted-by.leaseweb.com	80
POST /command/settings HTTP/1.1 User-Agent: 8b65916051836d5cfd41946e79d14316 Content-Length: 0 Content-Type: text/plain; charset=UTF-8 Host: [redacted] Connection: Keep-Alive			
118.949	write	hosted-by.leaseweb.com	80
POST /command/update_exceptions HTTP/1.1 User-Agent: 8b65916051836d5cfd41946e79d14316 Content-Length: 0 Content-Type: text/plain; charset=UTF-8 Host: [redacted] Connection: Keep-Alive			

http://anubis.iseclab.org/?action=result&task_id=1baf6e4953968c39401c30fe2f1f03089&format=html

Further Reading (dynamic analysis)

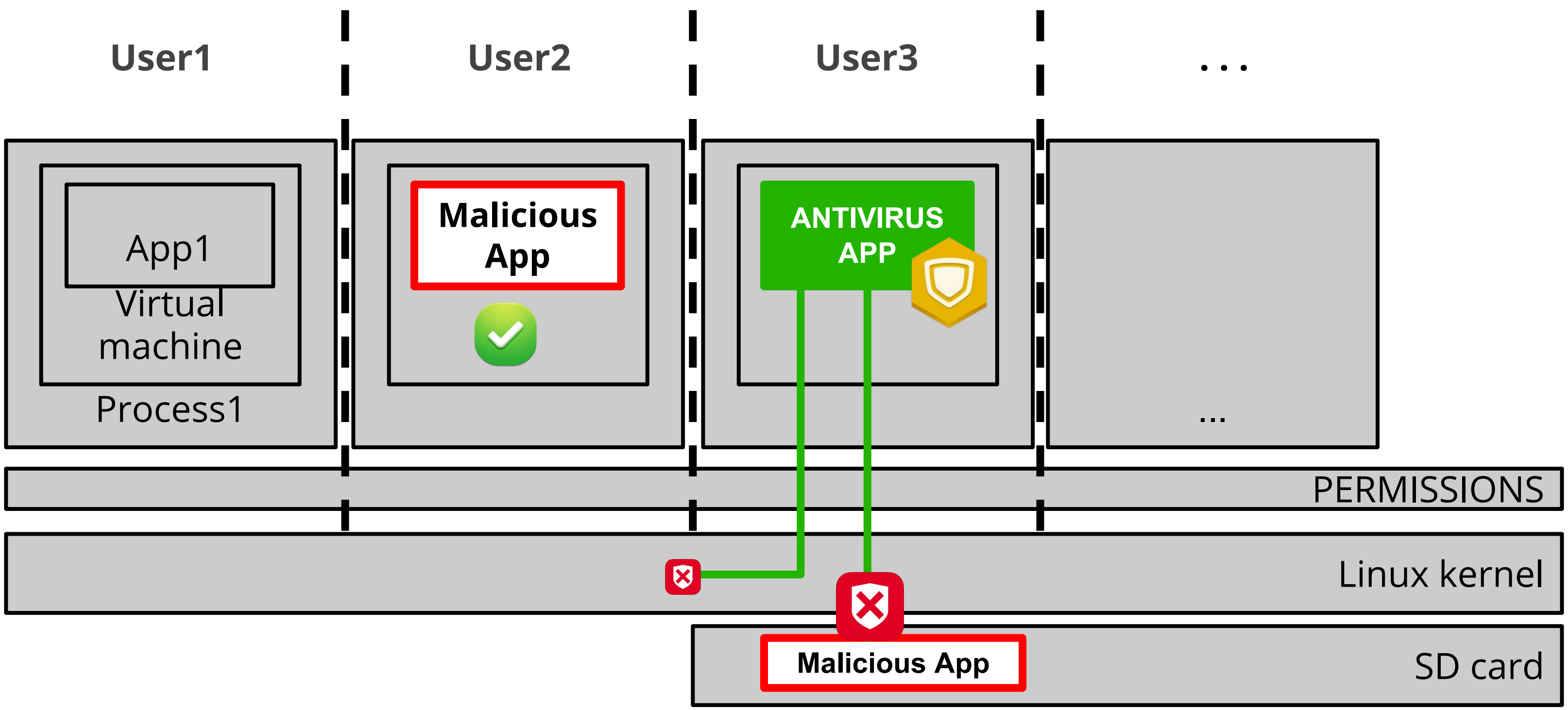
Reina et al., *"A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors"*, EuroSec 2013

Yan et al., *"DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis"*, USENIX Security 2013

Reflections on Antivirus for Mobiles

- The case of Android
- AVs are apps
- Apps cannot interfere with each other
- How can AVs check for malicious apps if they cannot access the entire system?
    - run check upon installation
    - list of package names of malicious apps
    - list of MD5s of malicious apps
    - limit scan to the SD card (world readable)
- Give the AV root privileges: does this increase the level of security?

Antivirus vs. Sandbox



Further Reading (AV evaluation)

Maggi et al., *"AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors"*, ACM SPSM 2013 - http://andrototal.org

Rastogi et al., *"DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks"*, ASIACCS 2013

Counteracting Countermeasures

Anti-analysis

- obfuscate or pack code
    - 17% malicious samples using Proguard
- encrypt strings
    - 27% malicious samples using encryption

Anti-debugging

- check if running in a VM or emulator
    - TelephonyManager.getDeviceId() ~> 0
    - TelephonyManager.getPhoneType() ~> 1

Further Reading (VM evasion)

T. Petsas et al., *"Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware"*, EuroSec 2014

Vidas and Christin, *"Evading Android Runtime Analysis via Sandbox Detection"*, CCS 2014

C. Zheng et al., *"On-Chip System Call Tracing: A Feasibility Study and Open Prototype"*, CNS 2016

Conclusions

Mobile security model (single user, multi app) is different from traditional security models (multi user).

Userland sandboxing solves the majority of problems, but mobile malware has been a reality in the past 4 years.

Malware authors adapted their tactics to leverage social engineering.

Researchers are very active in developing automatic analysis techniques.