Mobile Security

Presenter: Yinzhi Cao
Lehigh University

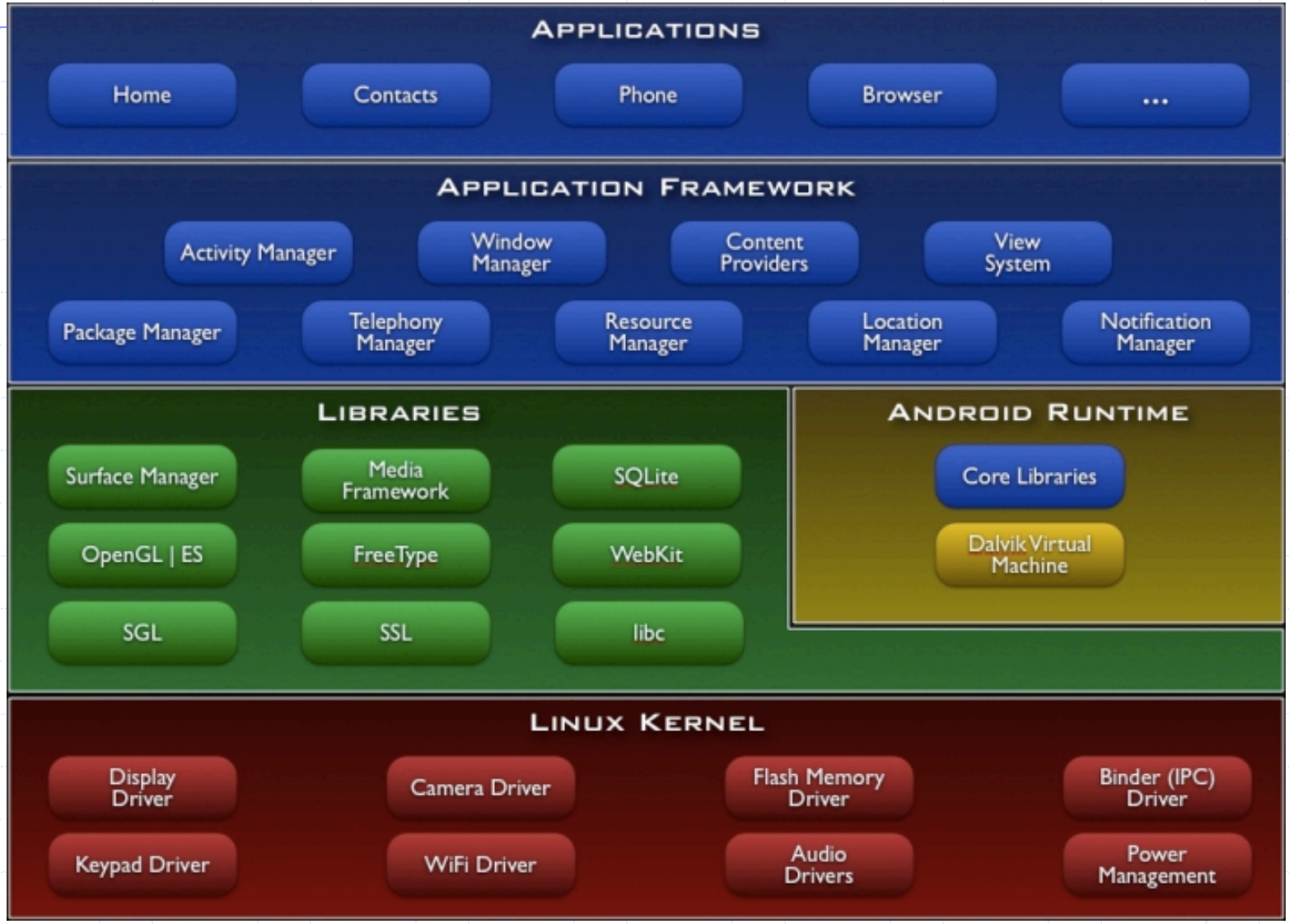
Some contents are borrowed from the following sources.

- http://siis.cse.psu.edu/slides/android-sec-tutorial.pdf
- http://blogs.ubc.ca/computersecurity/files/2012/01/mobile_security.pdf
- http://www.slideshare.net/pragatiogal/understanding-android-security-model?from_action=save
- https://source.android.com/devices/tech/security/index.html
- https://www.trust.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/LectureSlides/ESS-SS2012/9_iOS_-_hand-out.pdf
- https://developer.apple.com/library/ios/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html
- https://www.apple.com/br/privacy/docs/iOS_Security_Guide_Oct_2014.pdf

Android Security

- Android Security
  - Android Application Security
  - Android Kernel Security
- iOS Security
- Mobile Attacks

Android Model



Android Application

- While each application runs as its own UNIX uid, sharing can occur through application-level interactions.
  - Interactions based on components
  - Different component types
    - Activity
    - Service
    - Content Provider
    - Broadcast Receiver

Activity

- The user interface consists of a series of Activity components.
  - Each Activity is a “screen”.
  - User actions tell an Activity to start another Activity, possibly with the expectation of a result.
  - The target Activity is not necessarily in the same application.
  - Directly or via Intent “action strings”.
  - Processing stops when another Activity is “on top”.

Service

- Background processing occurs in Service components.
  - Downloading a file, playing music, tracking location, polling, etc.
  - Local vs. Remote Services (process-level distinction)
- Also provides a "service" interface between applications
  - Arbitrary interfaces for data transfer
    - Android Interface Definition Language (AIDL)
  - Register callback methods
  - Core functionality often implemented as Service components, e.g., Location API, Alarm service
- Multiple interfaces
  - Control: start, stop
  - Method invocation: bind

Content Provider

- Content Provider components provide a standardized interface for sharing data, i.e., content (between applications).
- Models content in a relational DB
  - Users of Content Providers can perform queries equivalent to SELECT, UPDATE, INSERT, DELETE
  - Works well when content is tabular
  - Also works as means of addressing “files”
- URI addressing scheme
  - content://<authority>/<table>/[<id>]
  - content://contacts/people/10

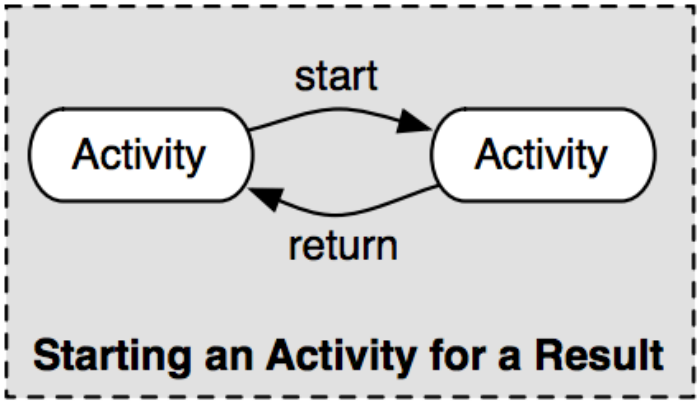
Broadcast Receiver

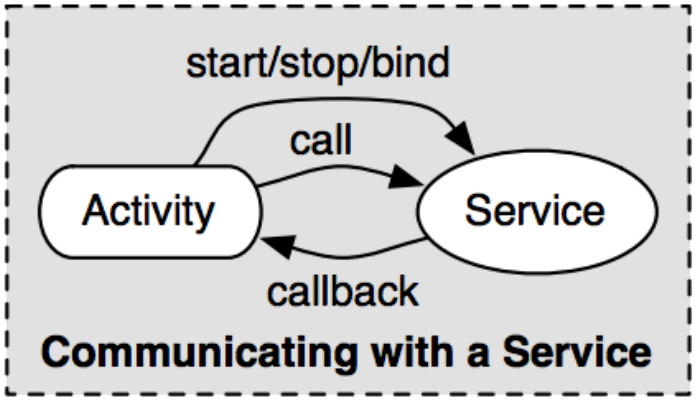
- Broadcast Receiver components act as specialized event Intent handlers (also think of as a message mailbox).
- Broadcast Receiver components "subscribe" to specific action strings (possibly multiple)
  - action strings are defined by the system or developer
  - component is automatically called by the system
- Recall that Android provides automatic Activity resolution using "action strings".
  - The action string was assigned to an Intent object
  - Sender can specify component recipient (no action string)

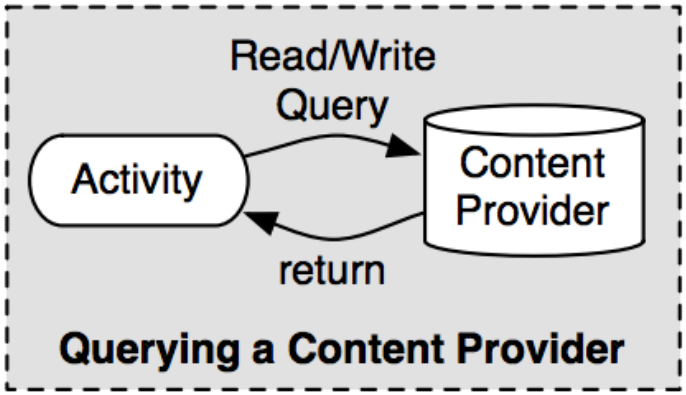
Intent

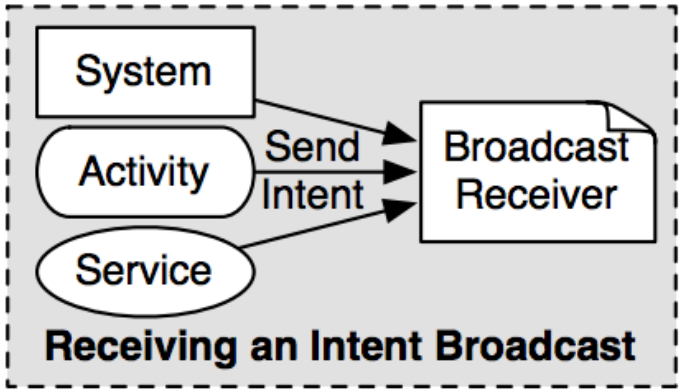
- Intents are objects used as inter-component signaling
  - Starting the user interface for an application
    - StartActivity
  - Sending a message between components
    - broadcastIntent
  - Starting a background service
    - StartService, BindService
- Format
  - Action: ACTION_VIEW, ACTION_DIAL
  - Data: URI (content://contacts/people/1, tel:123)

Components Interations

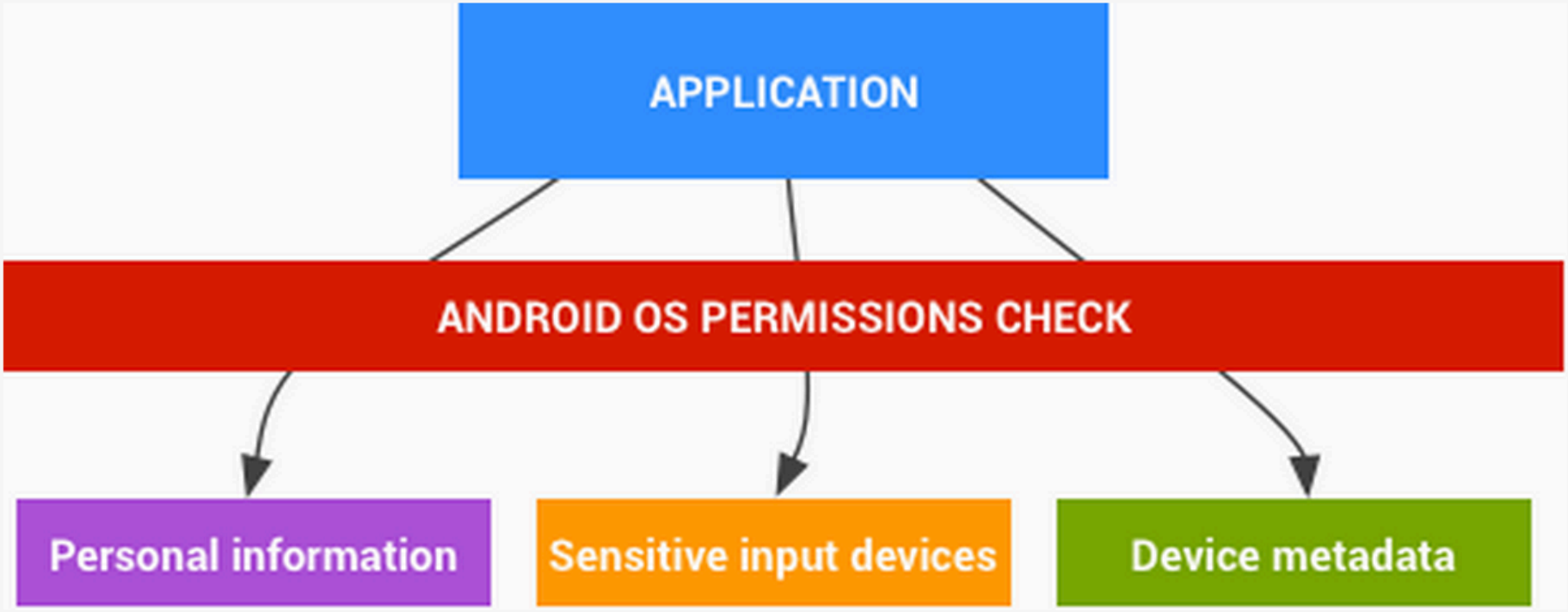








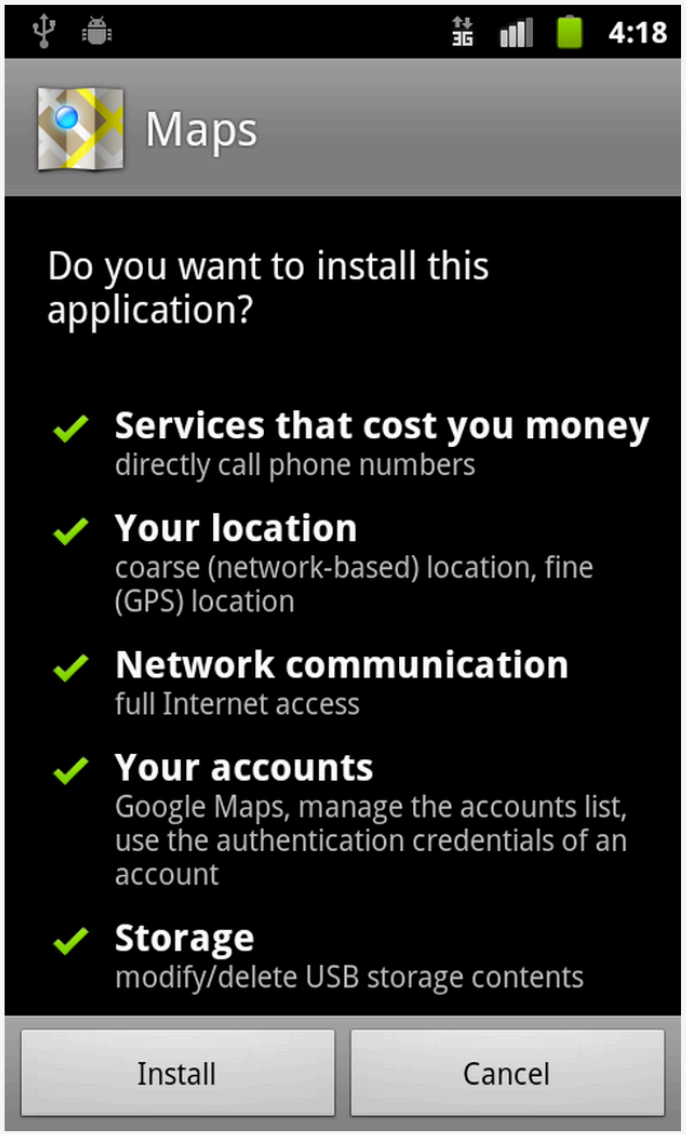
Android Permissions



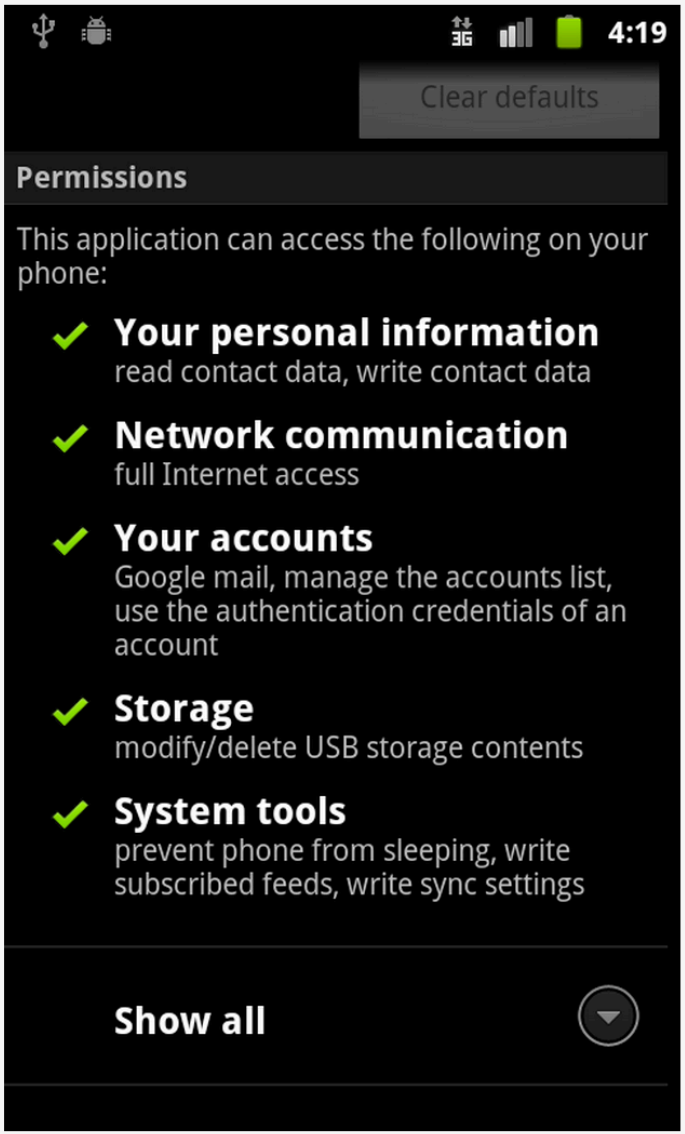
Permission Levels

- Normal
  - `android.permission.VIBRATE`
  - `com.android.alarm.permission.SET_ALARM`
- Dangerous
  - `android.permission.SEND_SMS`
  - `android.permission.CALL_PHONE`
- Signature
  - `android.permission.FORCE_STOP_PACKAGES`
  - `android.permission.INJECT_EVENTS`
- SignatureOrSystem
  - `android.permission.ACCESS_USB`
  - `android.permission.SET_TIME`

Permissions at Application Install -- Google Maps



Permissions of an Installed Application -- Gmail



Android Manifest File

- Manifest files are the technique for describing the contents of an application package (i.e., resource file)
- Each Android application has a special AndroidManifest.xml file (included in the .apk package)
  - describes the contained components
  - components cannot execute unless they are listed
  - specifies rules for “auto-resolution”
  - specifies access rules
  - describes runtime dependencies
  - optional runtime libraries
  - required system permissions

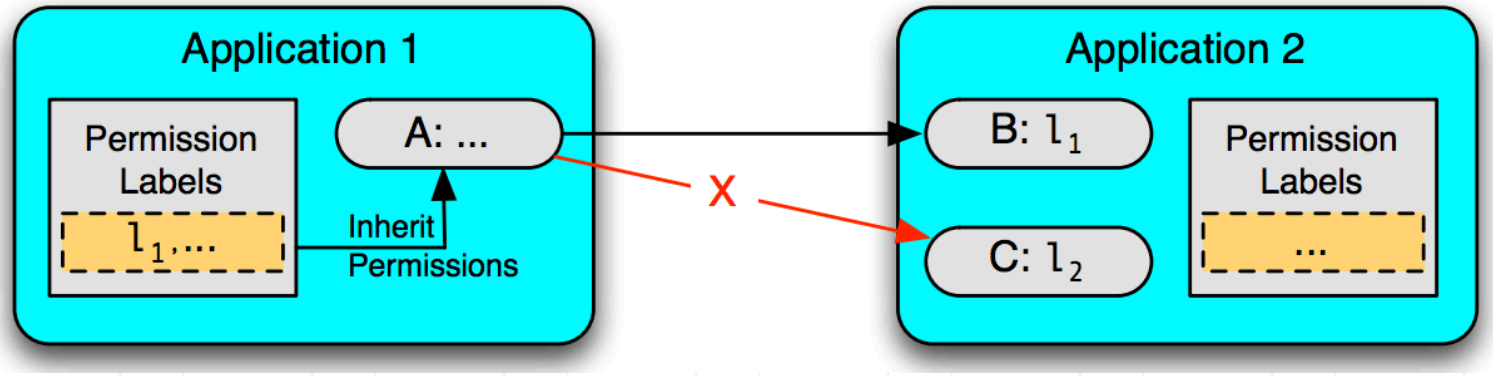
Permissions in Android Manifest

- Define Permissions
- Use Permissions
- Associate Permissions

```
<manifest . . . >
   <permission android:name="com.example.project.DEBIT_ACCT" . . . />
   <uses-permission android:name="com.example.project.DEBIT_ACCT" />
   . . .
   <application . . .>
      <activity android:name="com.example.project.FreneticActivity"
                android:permission="com.example.project.DEBIT_ACCT"
                . . . >
         . . .
      </activity>
   </application>
</manifest>
```

Permission Model

- Android focuses on Inter Component Communication (ICC)
  - The Android manifest file allows developers to define an access control policy for access to components
  - Each component can be assigned an access permission label
  - Each application requests a list of permission labels (fixed at install)
- Android's security model boils down to the following:



Note 1

- Components can be public or private.
  - Default is dependent on “intent-filter” rules
  - The manifest schema defines an “exported” attribute
- Why: Protect internal components
  - Especially useful if a “sub-Activity” returns a result
  - Implication: Components may unknowingly be (or become) accessible to other applications.
- Best Practice: Always set the “exported” attribute.

<activity android:name=“myApp” android:exported=“false”></activity>

Note 2:

- If the manifest file does not specify an access permission on a public component, any component in any application can access it.
- Why: Some components should provide "global" access, e.g., the main Activity for an Application
  - Permissions are assigned at install-time
- Implication: Unprivileged applications have access
- Best Practice: Components without access permissions should be exceptional cases, and inputs must be scrutinized (consider splitting components).

Note 3

- The code broadcasting an Intent can set an access permission restricting which Broadcast Receivers can access the Intent.
- Why: Define what applications can read broadcasts
- Implication: If no permission label is set on a broadcast, any unprivileged application can read it.
- Best Practice: Always specify an access permission on Intent broadcasts (unless explicit destination).

sentBroadcast(intent, permission)

Note 4

- PendingIntent objects allow another application to "finish" an operation for you via RPC.
  - Execution occurs in the originating application's "process" space
- Why: Allows external applications to send to private components
  - Used in a number of system APIs (Alarm, Location, Notification), e.g., timer
- Implication: The remote application can fill in unspecified values.
- Best Practice: Only use Pending Intents as "delayed callbacks" to private Broadcast Receivers/Activities and always fully specify the Intent destination.

Note 5

- Content Providers have two additional security features
  - Separate “read” and “write” access permission labels
- Why: Provide control over application data
  - e.g., FriendProvider uses read and write permissions
- Implication: Content sharing need not be all or nothing URI permissions allow delegation (must be allowed by Provider)
- Best Practice: Always define separate read and write permissions.
  - Allow URI permissions when necessary

Note 6

- A component (e.g., Service) may arbitrarily invoke the checkPermission() method to enforce ICC.
- Why: Allows Services to differentiate access to specific methods.
- Implication: The application developer can add reference monitor hooks
- Best Practice: Use checkPermission() to mediate “administrative” operations.

Note 7

- Permission requests are not always granted
- Why: Malicious applications may request harmful permissions
- Implication: Users may not understand implications when explicitly granting permissions.
- Best Practice: Use signature permissions for application “suites” and dangerous permissions otherwise
  - Include informative descriptions

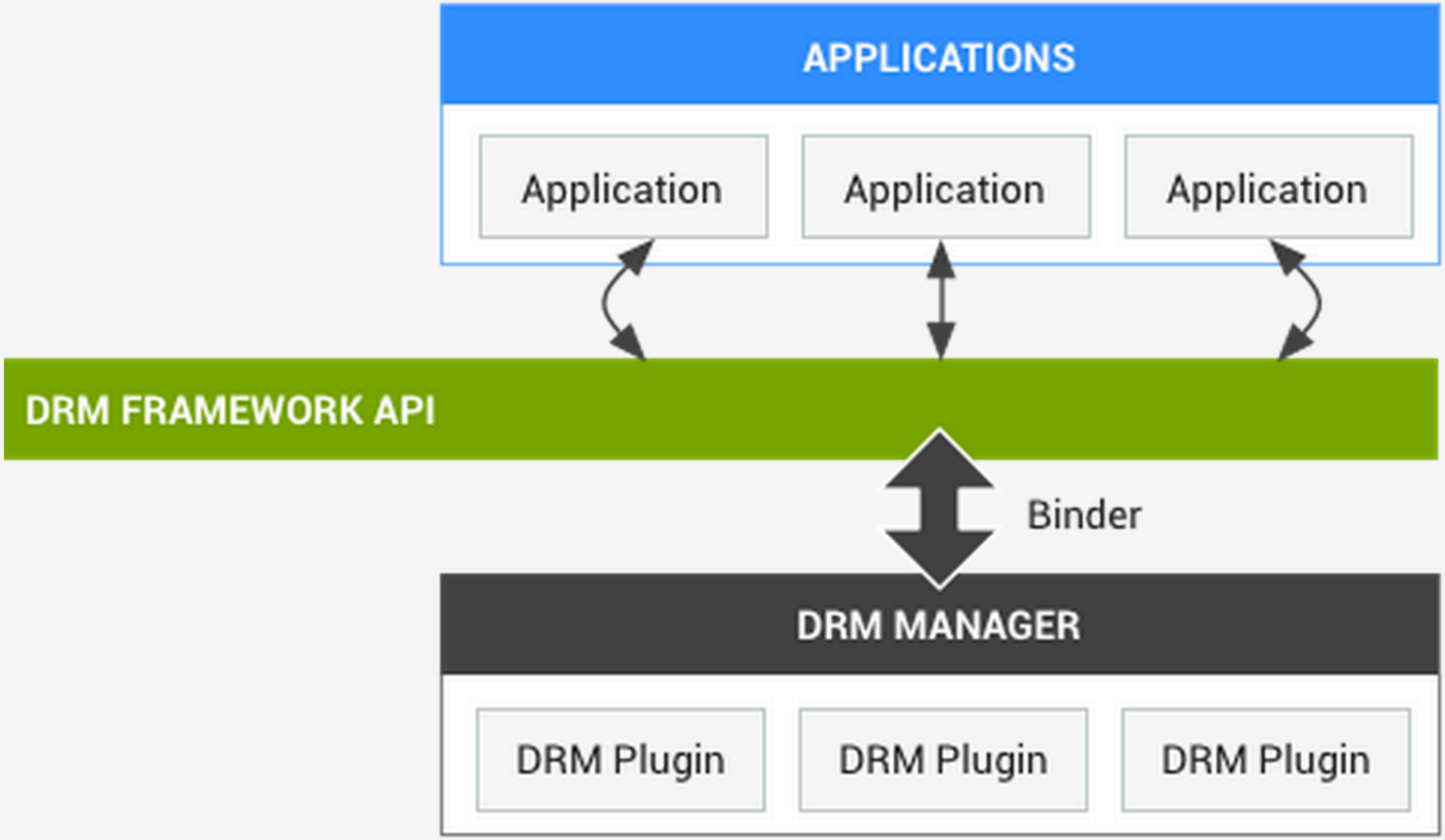
```
<permission
android:name="org.tutorial.permission.my"
android:label="@string/permlab"
android:description="@string/permdesc"
Android:protectionLevel="dangerous">
<string name="permlab">...</string>
<string name="permdesc">...</string>
```

Kernel Security

- Linux Security
  - A user-based permissions model
    - User: An application in Android with unique UID
  - Process isolation
  - Extensible mechanism for secure IPC
  - The ability to remove unnecessary and potentially insecure parts of the kernel
- Therefore
  - Prevents user A from reading user B's files
  - Ensures that user A does not exhaust user B's memory
  - Ensures that user A does not exhaust user B's CPU resources
  - Ensures that user A does not exhaust user B's devices (e.g. telephony, GPS, bluetooth)

Digital Right Managements

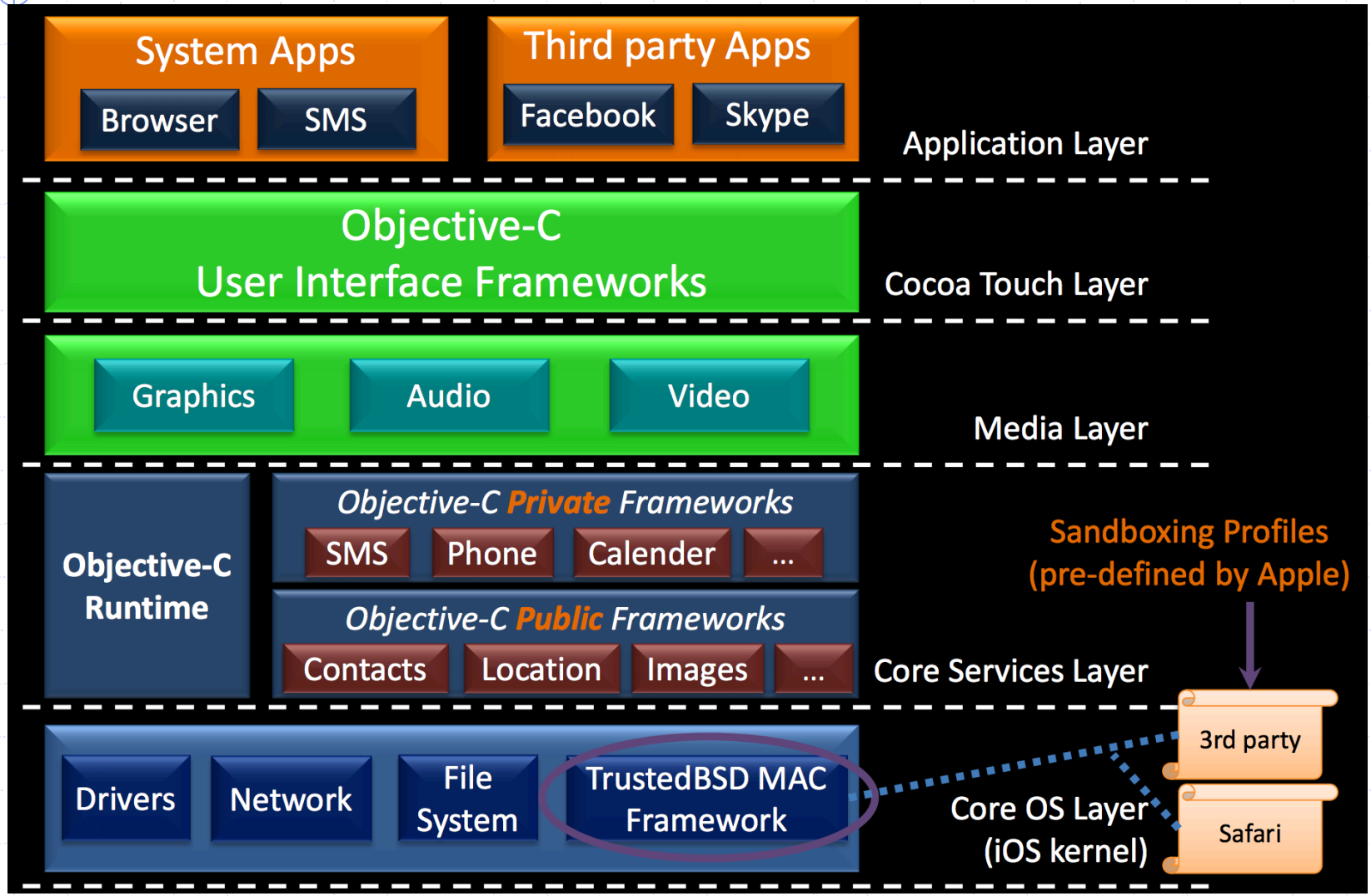
- The Android platform provides an extensible DRM framework that lets applications manage rights-protected content according to the license constraints that are associated with the content.
  - A DRM framework API
  - A native code DRM manager

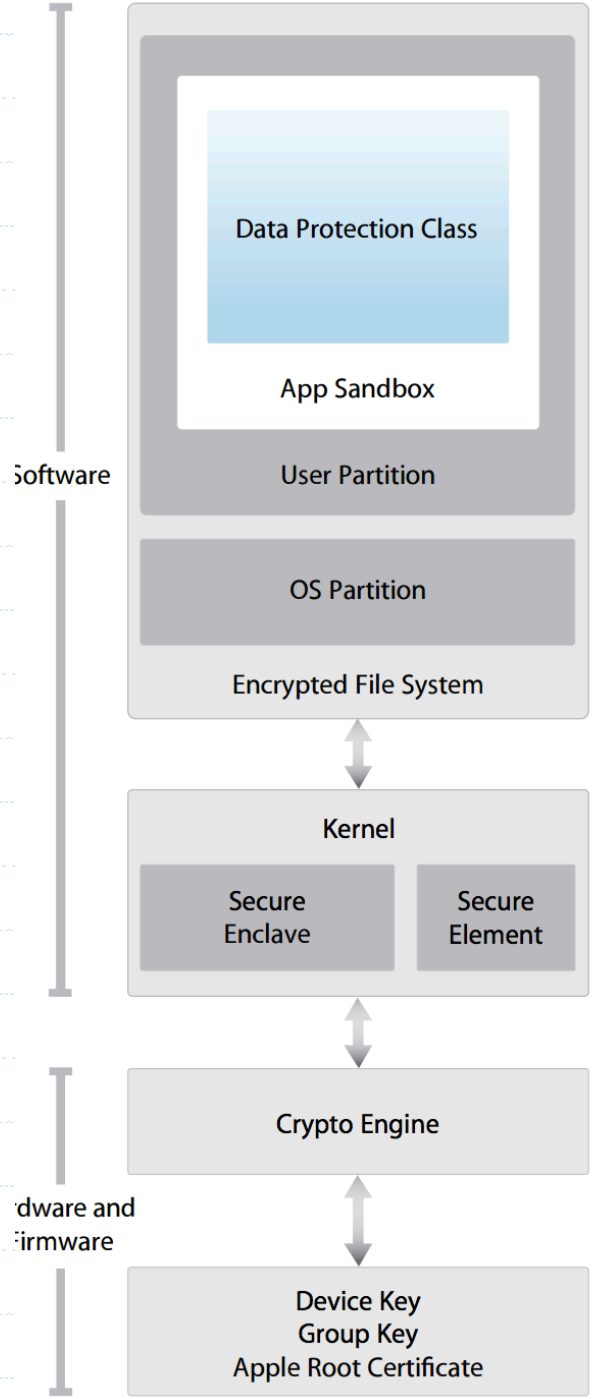

APPLICATIONS
Application
Application
Application
DRM FRAMEWORK API
Binder
DRM MANAGER
DRM Plugin
DRM Plugin
DRM Plugin

iOS Security

- Advanced Mobile OS
- Distributed exclusively for Apple hardware
- User Interface- multi-touch gestures (swap, tap, pinch and reverse pinch)
- Internal accelerometers respond to shaking the device or rotating it.
- Not fully Unix compatible

iOS Architecture




Data Protection Class
App Sandbox
User Partition
OS Partition
Encrypted File System
Kernel
Secure Enclave
Secure Element
Crypto Engine
Device Key
Group Key
Apple Root Certificate

iOS Security

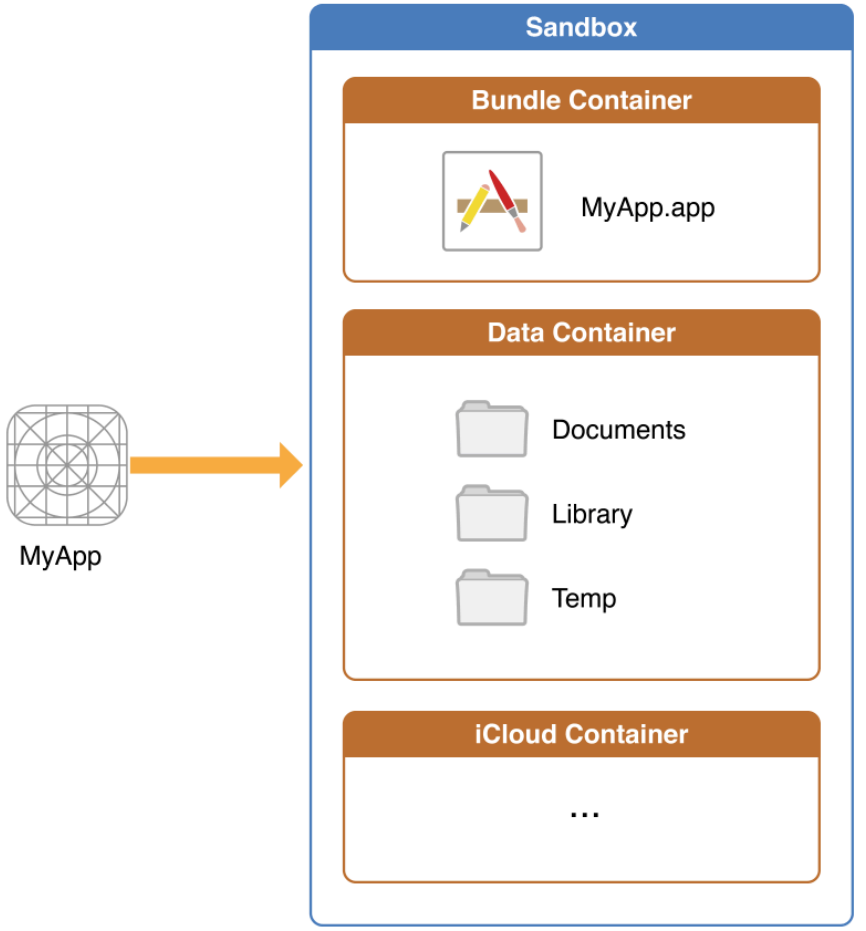
- Application Security
  - Installation Security
    - Code signing: only from App Store
  - Runtime Security
- System/Kernel Security

Installation Security

- iOS will refuse to execute unsigned code
  - Specifically, all executable code (System Apps and Third-Party Apps) has to be signed using an Apple-issued certificate
  - In addition, Apple enforces Mandatory Code Signing
  - Third-Party Apps are not allowed to load unsigned code resources at runtime or using self-modifying code
- Code Signing Enforcement (CSE)
  - At runtime, iOS enforces code signature checks on executable memory pages to ensure that an app has not been modified while it is executing
  - Exception: Safari and Webapps
  - Since CSE would restrict any code generation, iOS added an execption to web applications so that they can use just-in-time (JIT) code generation

iOS Application Security

- Running in the same user: mobile
- App Sandboxing by TrustedBSD MAC kernel
  - allows the definition of sandboxing profiles, while profiles can be attached at process-level
  - sandboxing profiles contain access control rules based on system call and file-system level



Rule Example

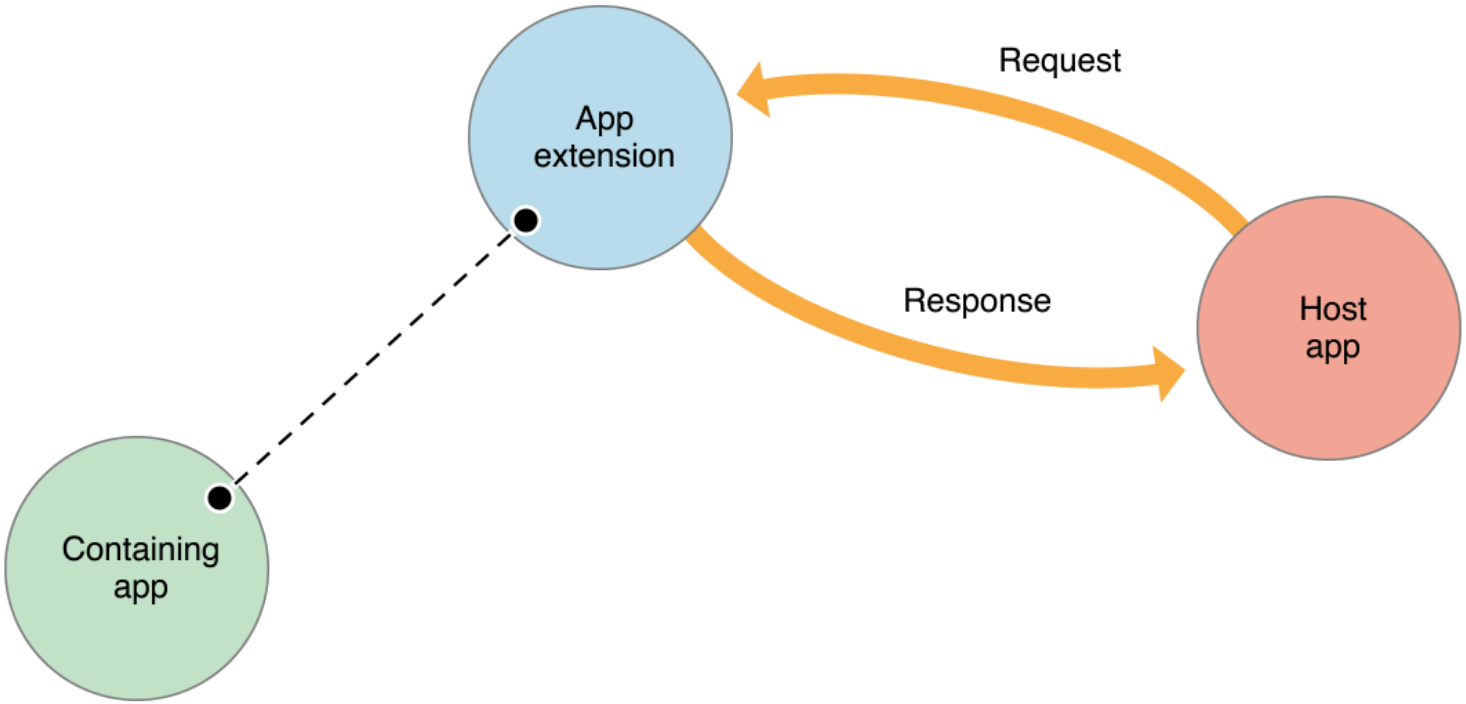
Action	Resource (Filepath)	Decision
file-read	^/private/var/mobile/Media/Photos/Thumbs$	ALLOW
file-read	^/private/var/logs(/|$)	DENY

Personal Information Access

- Each 3rd party app can freely access
  - the entire address book
  - Location information
  - Device information (e.g., IMSI, phone number)
  - E-Mail account configurations
  - WiFi configurations
  - Recent browser searches
  - Keyboard cache
  - Personal photos

Exporting Code

- App Extension



Comparison with Android

- Installation
  - Android: Check permissions
  - iOS: Check code signature
- Runtime
  - Android: Enforced by Linux user isolation
  - iOS: Enforced by APIs
- App Communication
  - Android: pendingIntent
  - iOS: App Extension

System/Kernel Security

- Secure Booting Chain
- Hardware Security Feature
- File System Encryption

Secure Booting Chain

- iOS enforces Secure Boot
  - Each component that is part of the boot-process is signed by Apple (to ensure integrity)
  - If one component of the boot process cannot be correctly loaded or verified, boot-up is stopped
  - In case boot-up is stopped, iOS will either try to connect to iTunes or return into DFU (Device Firmware Upgrade) mode
- Boot Chain Sequenze
  1. Boot ROM
     - Immutable code (stored in read-only memory during chip fabrication)
     - Contains Apple Root CA public key, which is used during the boot process to verify each involved component
  2. Low-Level Bootloader
  3. Next-Stage Bootloader (iBoot)
  4. iOS Kernel

Hardware Security Feature

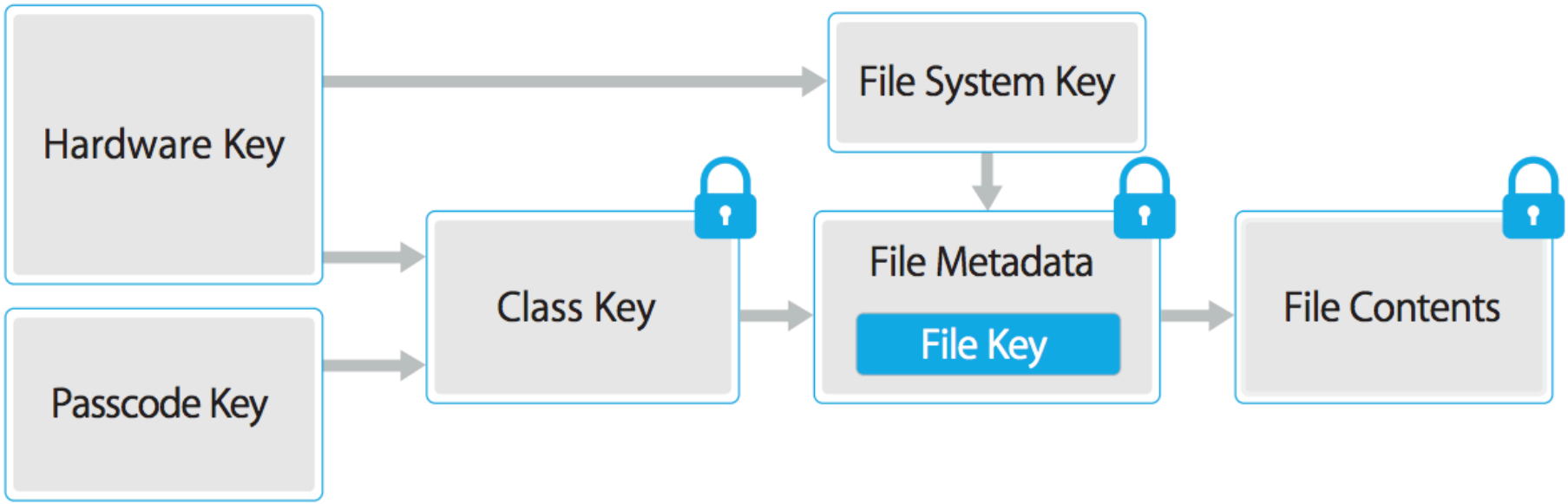
- Each iOS device has a dedicated AES-256 crypto engine
  - Crypto Engine is provided as a Hardware Module (due toperformance and power efficiency reasons)
  - Along with the AES engine, Apple also provides a SHA-1 hardware module
- Manufacture Keys
  - Apple provides the Device ID (UID) and the device group ID (GID) as AES 256 Bit keys
  - While the UID is unique to each device, the GID represents a processor class (e.g., Apple A5 processor)
  - The UID and GID keys are directly burned into the silicon and can only be accessed by the Crypto Engine
- Other Cryptographic Keys
  - All other keys are generated by the system‘s random number generator (RNG)

File System Encryption

- The iOS file system is encrypted by default
  - The encryption key for the file system is referred to as File System Key
  - This key is created when iOS is first installed and is protected by the Device UID
- Effectiveness
  - If the device gets stolen, a remote wipe command can be set up which simply wipes the File System Key rendering the entire file system unreadable
  - However, an adversary can use the device itself to decrypt the file system before the remote wipe command is delivered

File System Encryption Cont'd

- Every file is encrypted with a unique File Key, that is generated when the file created
  - The file key is wrapped with a Class Key (because each file is associated to a specific protection class) and stored in the file's metadata
  - The metadata is encrypted with the File System Key
  - The Class key is protected by the Device UID and (if configured for some files) the User Passcode


Hardware Key
Passcode Key
File System Key
Class Key
File Metadata
File Key
File Contents

Data Protection Class

- Complete Protection
  - The class key is protected with a key derived from the user passcode and the device UID.
- Protected Unless Open
  - Some files may need to be written while the device is locked. A good example of this is a mail attachment downloading in the background.
- Protected Until First User Authentication
  - The decrypted class key is not removed from memory when the device is locked.
- No Protection