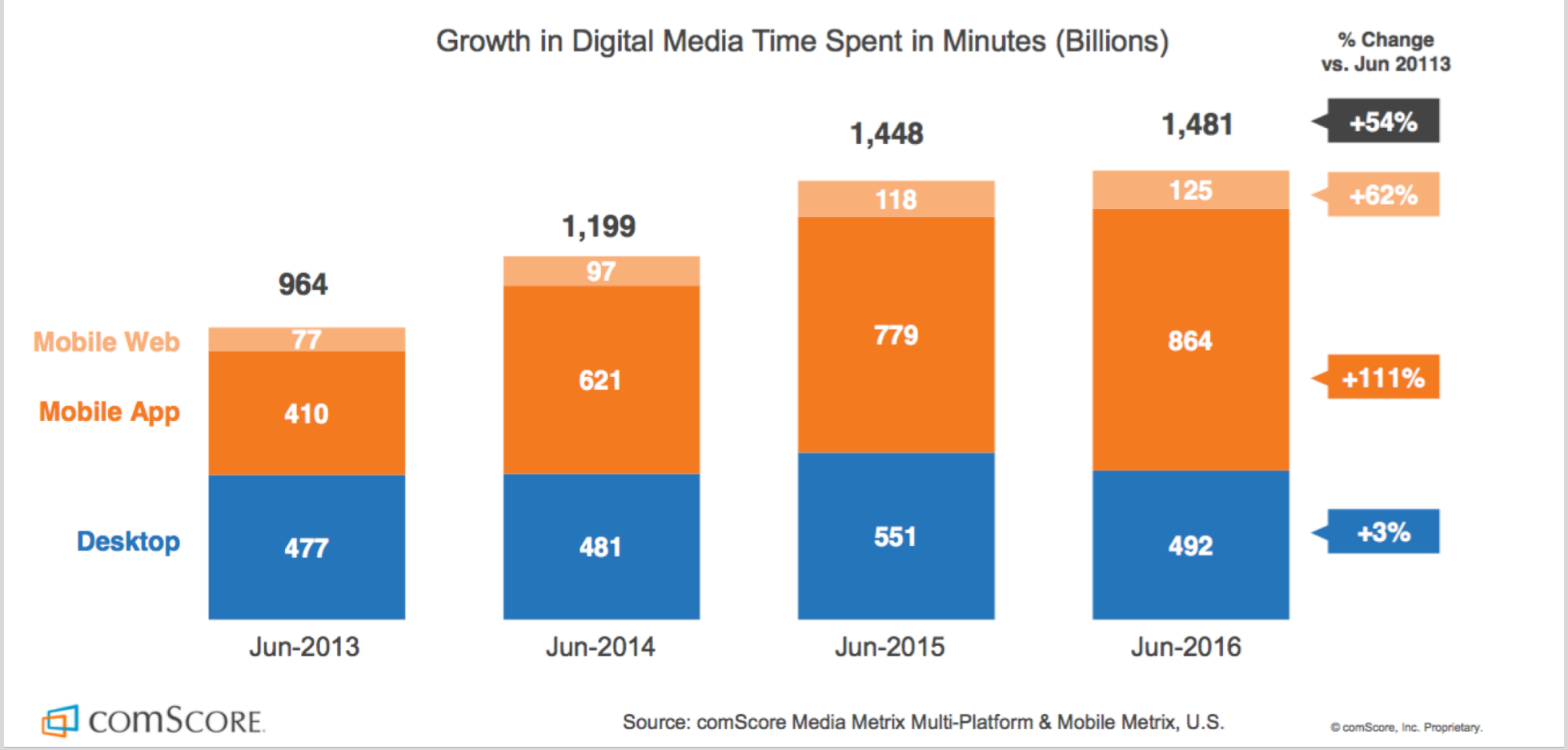
SECURITY ON MOBILE DEVICES

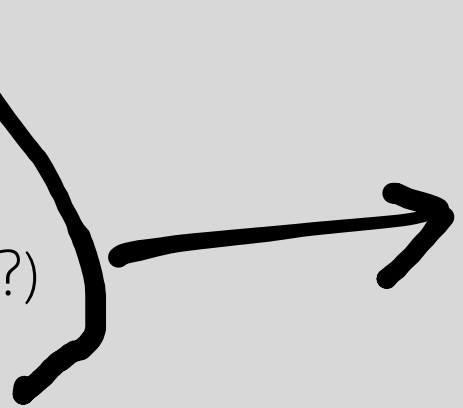
VITALY SHMATIKOV

Where Users Spend Time



What's Valuable on Phones?

- Identify location
- Record phone calls
- Log SMS (remember 2FA SMS?)
- Send premium SMS messages



specific to mobile

- Steal contact list, email, messaging, banking/financial information, private photos…
- Phishing
- Malvertising
- Join Bots

Physical Threats

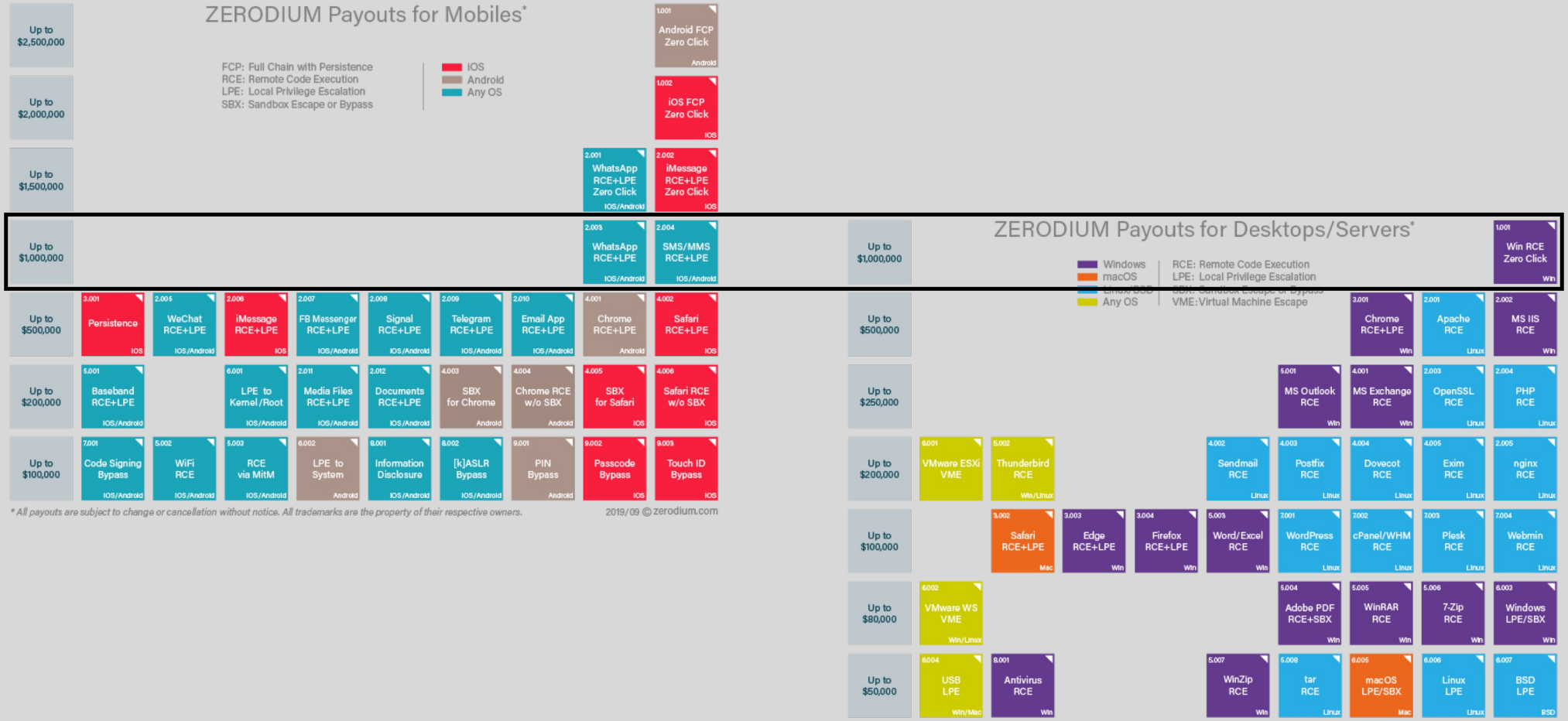
- Powered-off devices under complete physical control of an adversary
  - Including well-resourced nation-states, police, etc.
- Screen-locked devices under physical control of an adversary (e.g. thieves)
- Unlocked devices under control of different user (e.g. intimate partner abuse)
- Devices in physical proximity to an adversary who control radio channels, including cellular, WiFi, Bluetooth, GPS, NFC

Untrusted Code

Mobile OSes intentionally allow (with explicit user consent) installation of application code from arbitrary sources that can…

- Abuse APIs supported by the OS with malicious intent, e.g. spyware
- Exploit bugs in the OS
- Mimic system or other app user interfaces to confuse users
- Read content from system or other application user interfaces (e.g., screen-scrape)
- Inject input events into system or other app user interfaces

Mobile Exploits Very Valuable



Apple patches an NSO zero-day flaw affecting all devices

Citizen Lab says the ForcedEntry exploit affects all iPhones, iPads, Macs and Watches

Emergency Apple update on September 13, 2021

The flaw in iMessage was exploited by the Pegasus spyware from Israeli firm NSO group to gain full access to victims' devices.

NSO is infamous for developing, weaponizing, and deploying zero-click exploits against messaging apps

The exploit broke through new iPhone defenses that Apple had baked into iOS 14, dubbed BlastDoor, which were supposed to prevent silent attacks by filtering potentially malicious code.

Based on an integer overflow vulnerability in CoreGraphics image rendering library.

https://techcrunch.com/2021/09/13/apple-zero-day-nso-pegasus/

Unlocking Device

PINs,
patterns,
alphanumeric passwords
…

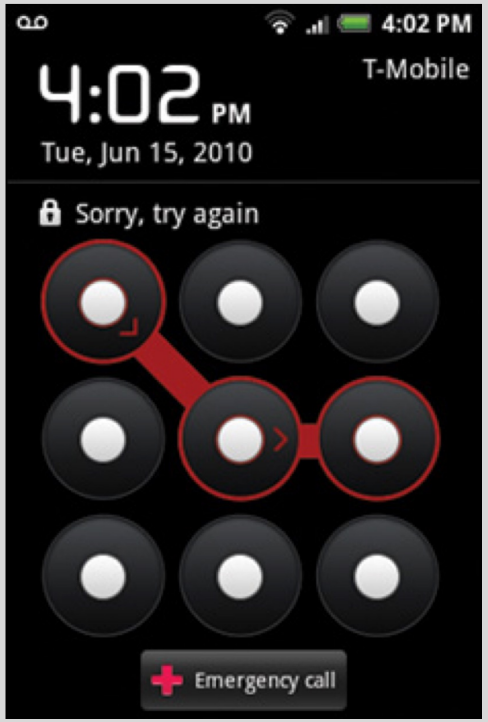
Swipe Code Problems

Smudge attacks [Aviv et al., 2010]

- Entering pattern leaves smudge that can be detected with proper lighting
- Smudge survives incidental contact with clothing

Another problem: entropy

- People choose simple patterns – few strokes
- At most 1,600 patterns with <5 strokes



iPhone Password Hashing

Goal: password hashing where a 4-8 digit PIN takes a very long time to crack, even if the device is physically compromised

Phone-specific aspects:

- Lots of computation uses up battery
- Physical access allows copying secret off the device and cracking remotely

Secure Enclave

Additional secure processor inside every iPhone

- Memory inaccessible to normal OS
- Utilizes a secure boot process that ensures its software is signed
- AES key burned in at manufacture

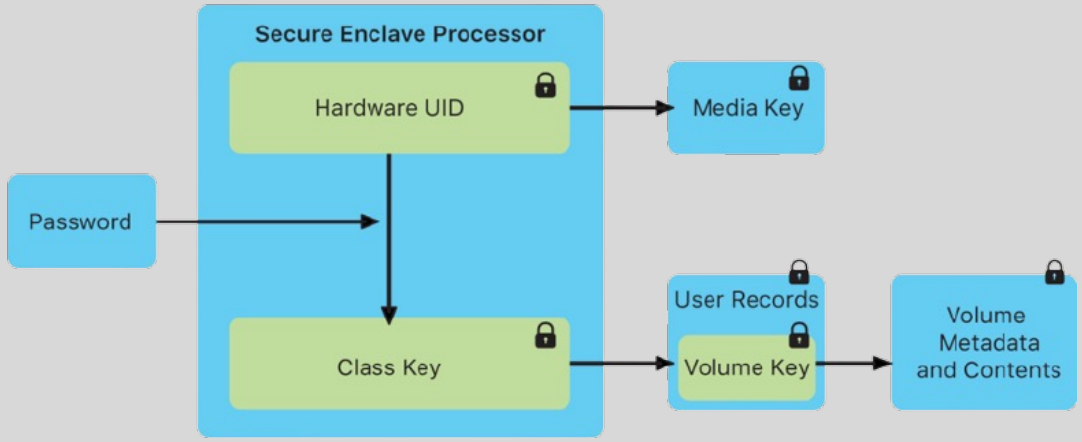
Secure enclave has instructions that allow encrypting and decrypting content using the key, but the key itself is never accessible (incl. via JTAG)

iPhone Unlocking

The passcode is entangled with the device's UID many times: $Encrypt_{UID}(Encrypt_{UID}(Encrypt_{UID}(passcode)...))$

User passcode is entangled with the AES key fused into secure enclave (known as UID)

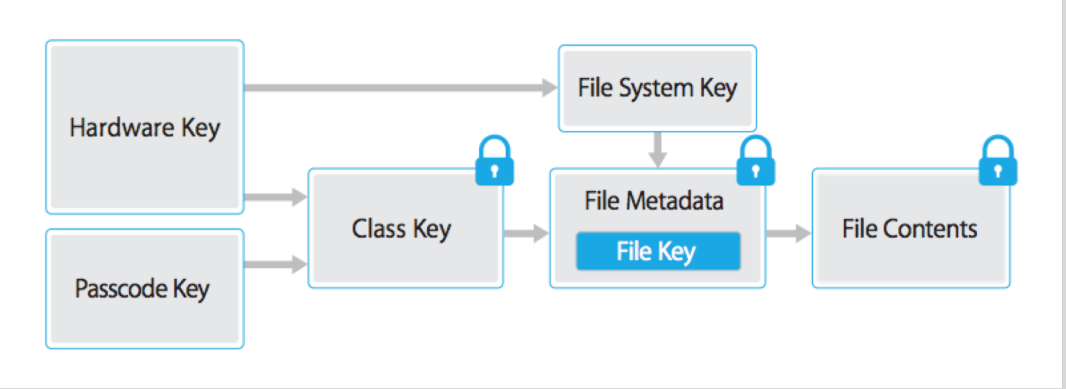
The key to decrypt the device can only be derived on the single secure enclave on a specific phone -- not possible to take offline and brute force



Approx. 80ms per password check
5 failed attempts ⇒ 1 min delay
9 failures ⇒ 1 hour delay
10 failed attempts ⇒ erase phone

enforced by firmware on the secure enclave itself — cannot be changed by iOS

FaceID / TouchID



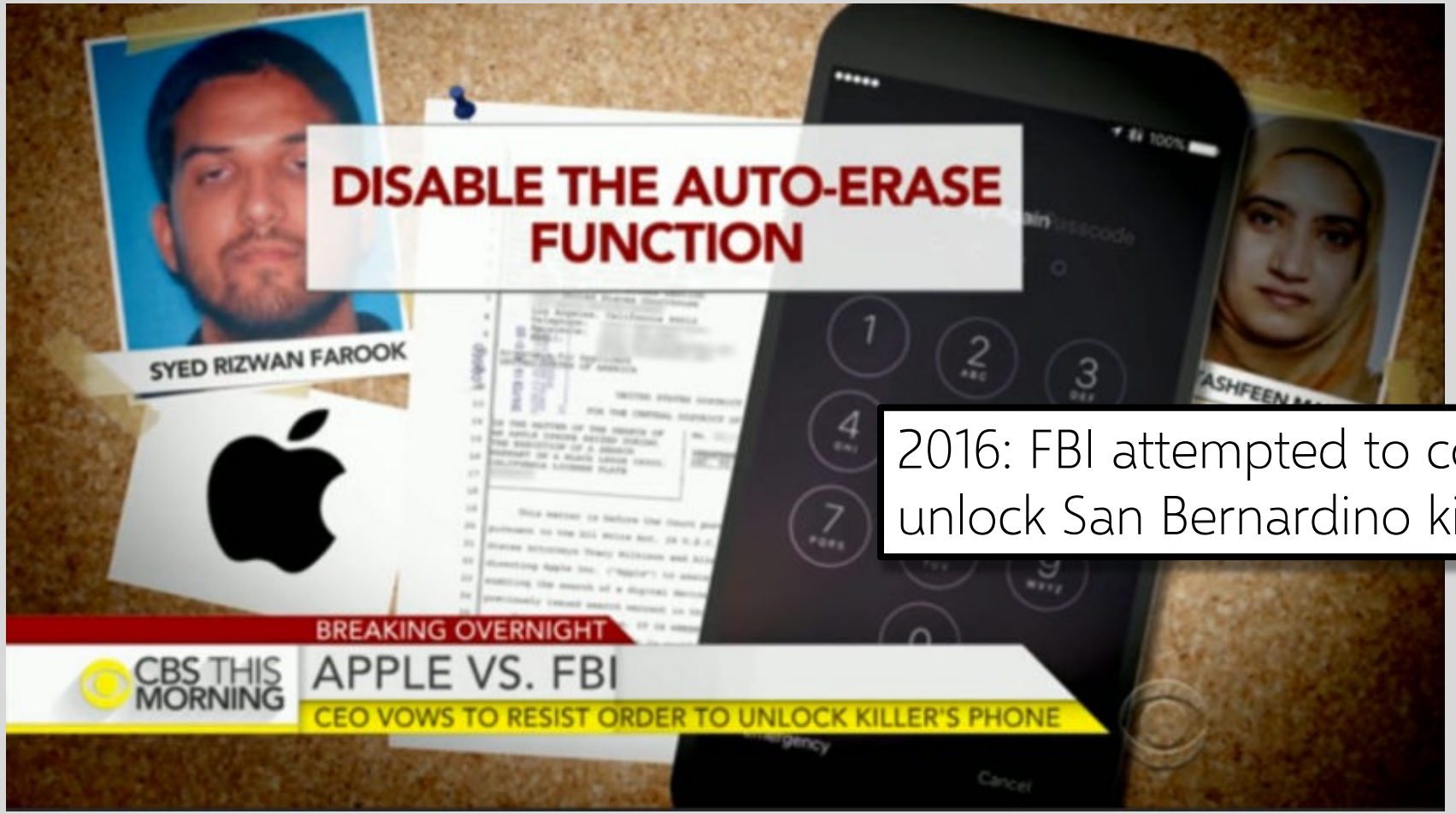
Application files written to flash memory are encrypted through a hierarchy of keys:

- Per-file key: encrypts all file contents (AES-XTS)
- Class key: encrypts per-file key (ciphertext stored in metadata)
- File-system key: encrypts file metadata

By default (no FaceID, TouchID), class encryption keys are erased from memory of secure enclave whenever the device is locked or powered off

When TouchID/FaceID is enabled, class keys are kept and hardware sensor sends fingerprint image to secure enclave. All ML/analysis is performed within the secure enclave.

Apple-FBI Dispute



2016: FBI attempted to compel Apple to unlock San Bernardino killer's phone

Technical Details of the Apple-FBI Dispute

The court order wanted a custom version of the secure enclave firmware that would...

this user-configurable feature of iOS 8 automatically deletes keys needed to read encrypted data after ten consecutive incorrect attempts

1."it will bypass or disable the auto-erase function whether or not it has been enabled"

2."it will enable the FBI to submit passcodes to the SUBJECT DEVICE for testing electronically via the physical device port, Bluetooth, Wi-Fi, or other protocol"

3."it will ensure that when the FBI submits passcodes to the SUBJECT DEVICE, software running on the device will not purposefully introduce any additional delay between passcode attempts beyond what is incurred by Apple hardware"

How FBI Got Access?

Paid $900,000 to Azimuth Security for an exploit

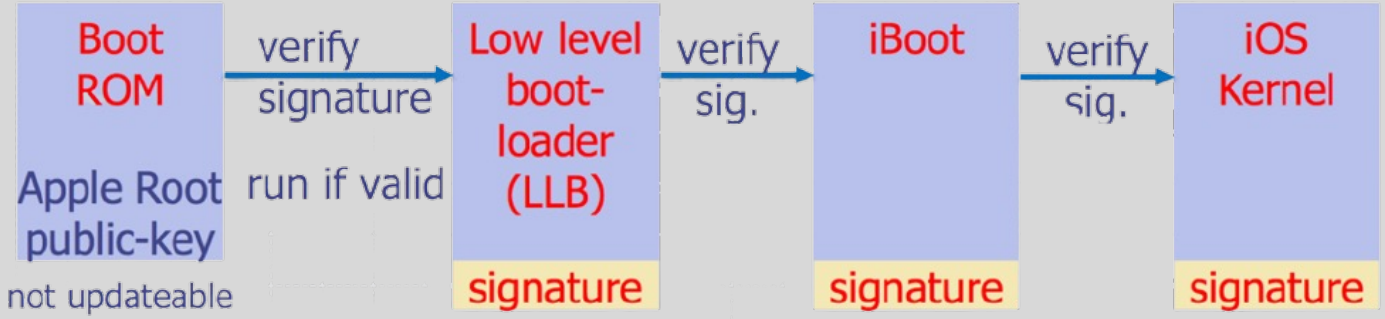
- Based on a vulnerability in open-source code from Mozilla that iOS used to let accessories to be plugged into the lightning port + several other exploits

One of the exploit authors founded Corellium

- ARM virtualization
- Lets researchers test “virtual” iOS and Android phones on a server, look for bugs, etc.

Apple tried to acquire Corellium, then sued them for copyright infringement and unlawful bypass of Apple’s security measures, settled in 2021

Why Couldn't FBI Upload Their Own Firmware?



Secure boot chain

- When an iOS device is turned on, it executes code from read-only memory known as Boot ROM. This immutable code, known as the hardware root of trust, is laid down during chip fabrication, and is implicitly trusted.
- The Boot ROM code contains the Apple Root CA public key, which is used to verify that the bootloader is signed by Apple. This is the first step in the chain of trust where each step ensures that the next is signed by Apple.

Secure Software Updates

To prevent devices from being downgraded to older versions that lack the security updates, iOS uses System Software Authorization

Device connects to Apple with cryptographic descriptors of each component update (e.g., boot loader, kernel, and OS image), current versions, a random nonce, and device-specific Exclusive Chip ID (ECID)

Apple signs device-personalized message allowing update, which boot loader verifies, both for main processor and secure enclave

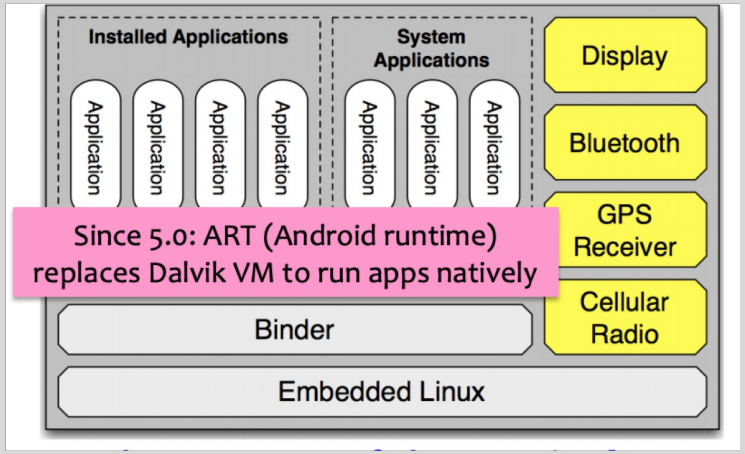
Android Isolation

Based on SE Linux (Linux with sandboxes)

Applications run as separate user IDs, in separate processes

- Attacks compromise the application, but not the entire system

To escape sandbox, must compromise Linux kernel



Structure of Android Applications

User interface

Background processing

Messages from other apps

Data storage

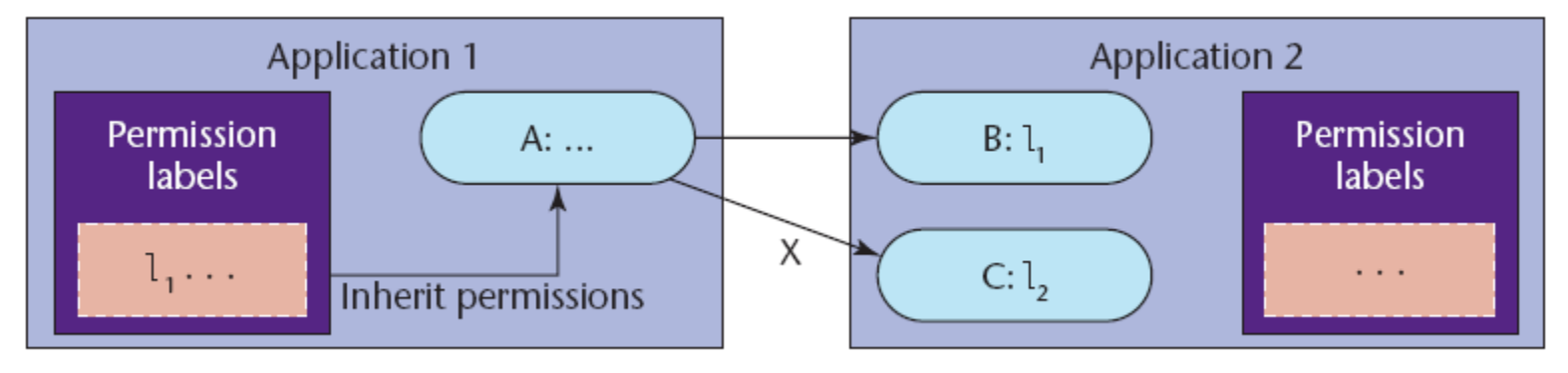
Intents are primary messaging mechanism for interaction between components

Image source: https://medium.com/@Abderraouf/understand-android-basics-part-1-application-activity-and-lifecycle-b559bb1e40e

Android Security Model

Based on permission labels assigned to applications and components

Android middleware mediates inter-component communication



Access permitted if labels assigned to the invoked component are in the collection of invoking component

Mandatory Access Control

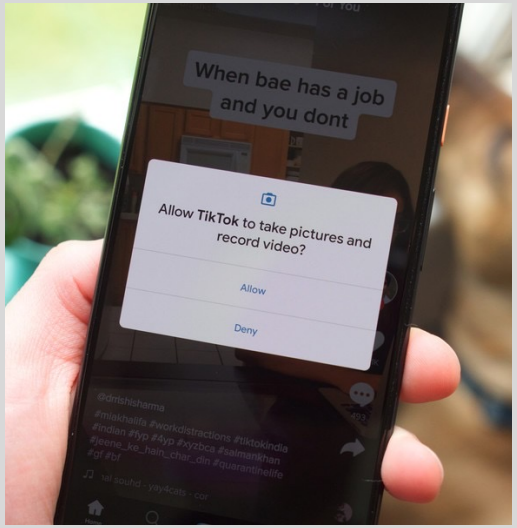
- Permission labels are set (via manifest) when app is installed and cannot be changed
- Permission labels only restrict access to components, they do not control information flow

Means what?

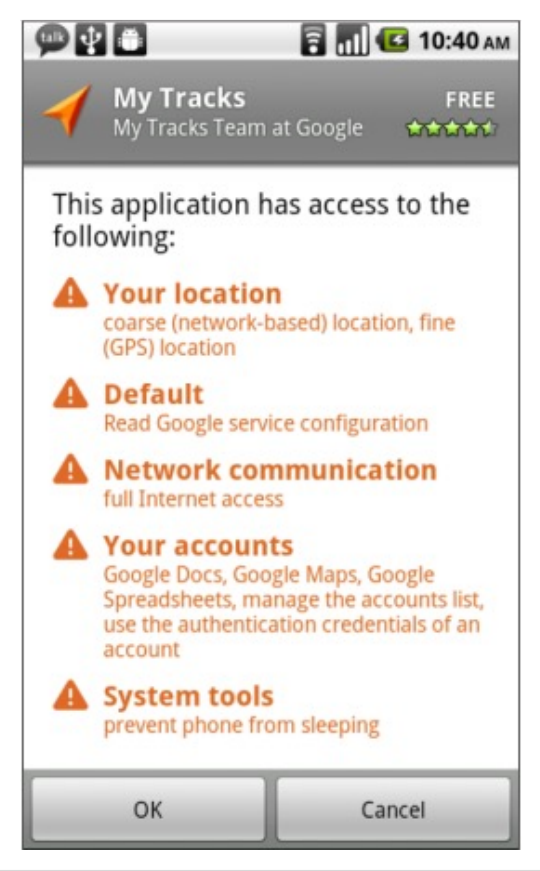
- Apps may contain “private” components that should never be accessed by another app (example?)
- If a public component doesn’t have explicit permissions listed, it can be accessed by any app

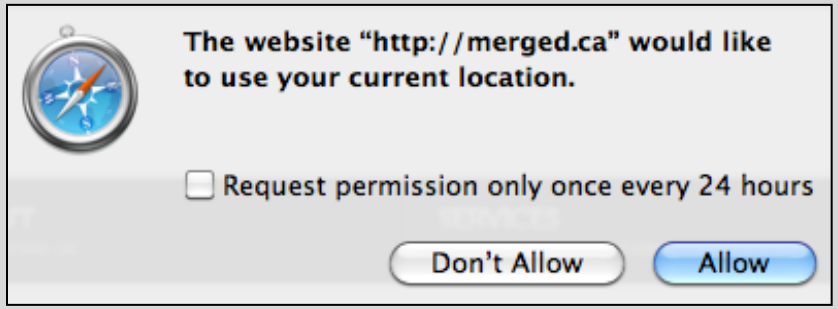
System API Access

- System functionality (eg, camera, networking) is accessed via Android API, not system components
- App must declare the corresponding permission label in its manifest + user must approve at the time of app installation
- Signature permissions are used to restrict access only to certain developers
  - Example: Only Google apps can directly use telephony API



Permissions: Not Just Android



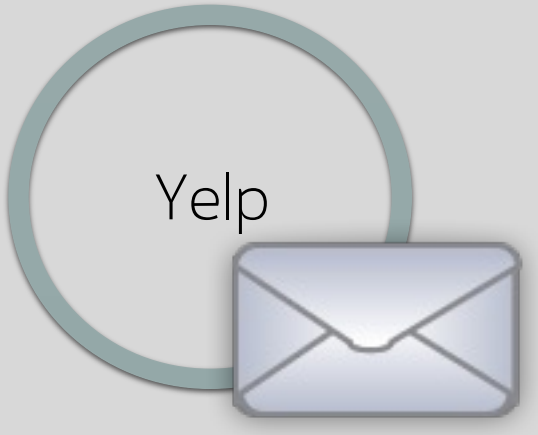


All mobile OSes, HTML5 apps, browser extensions…

Explicit Intents

Only the specified destination receives this message

Implicit Intents



Implicit Intent
Action: VIEW

Handles Action: VIEW

Map
App

Handles Action: VIEW

Browser
App

Security Issues with Implicit Intents

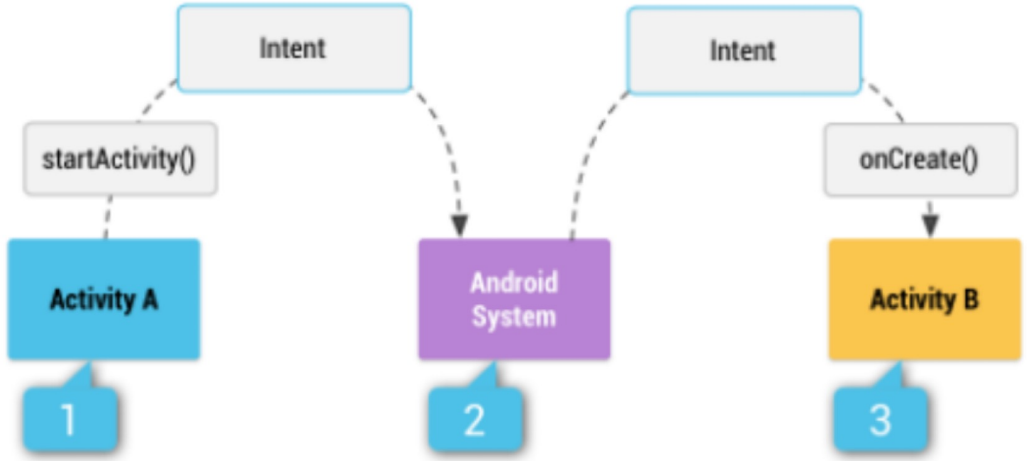


Figure 1. How an implicit intent is delivered through the system to start another activity: **[1]** *Activity A* creates an `Intent` with an action description and passes it to `startActivity()`. **[2]** The Android System searches all apps for an intent filter that matches the intent. When a match is found, **[3]** the system starts the matching activity (*Activity B*) by invoking its `onCreate()` method and passing it the `Intent`.

"Caution: To ensure that your app is secure, always use an explicit intent when starting a Servier. Using an implicit intent to start a service is a security hazard because you can't be certain what service will respond to the intent, and the user can't see which service starts."

Access Control for Intents

Permission labels on broadcast intents

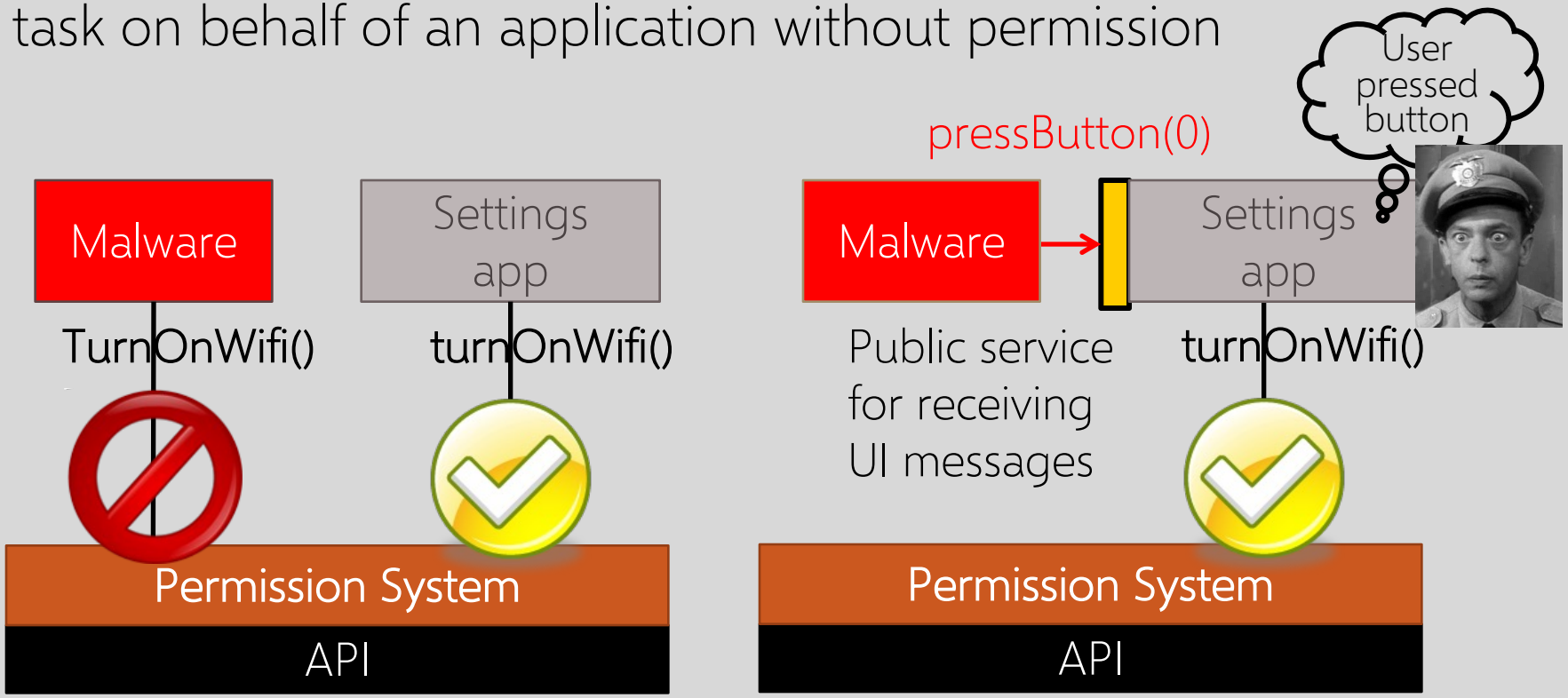
- Prevents unauthorized apps from receiving these intents – why is this important?

Pending intents: Instead of directly performing an action via intent, create an object that can be passed to another app, thus enabling it to execute the action

- Invocation involves RPC to the original app
- Introduces delegation into Android's MAC system

Permission Re-Delegation

An application with a permission performs a privileged task on behalf of an application without permission



Felt et al. "Permission Re-Delegation: Attacks and Defenses" (2011)

Re-Delegation is a Confused Deputy Problem

The “deputy” app may accidentally expose privileged functionality…

… or intentionally expose it, but the attacker invokes it in a surprising context

- Example: broadcast receivers in Android

… or intentionally expose it and attempt to reduce the invoker’s authority, but do it incorrectly

Remember URL checks on webhooks?
Parent frame origin checks in frame-busting code?