Mobile Security

CS155 Computer and Network Security

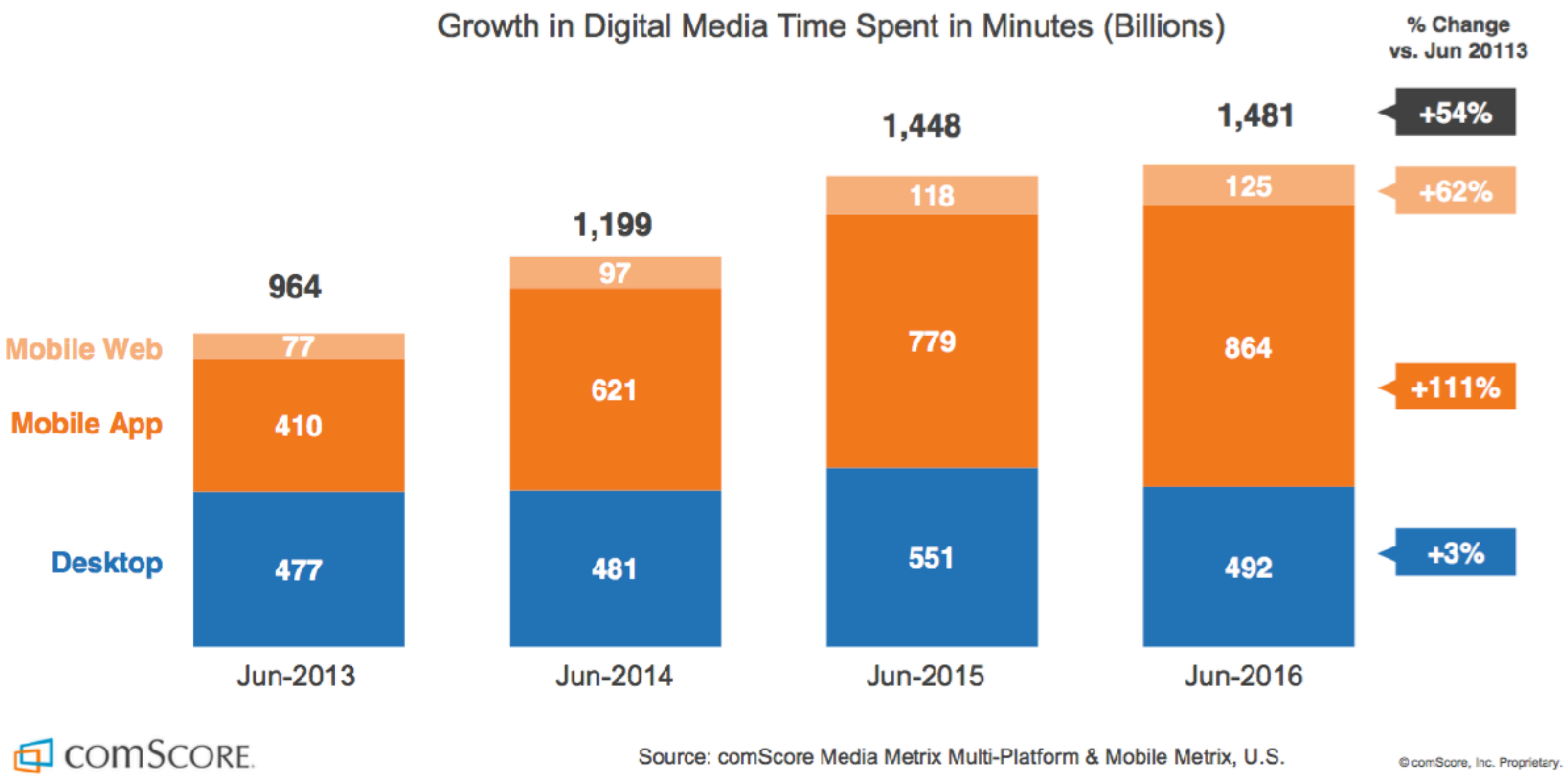
Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh and Zakir Durumeric at Stanford University. When slides are obtained from other sources, a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.

Stanford University

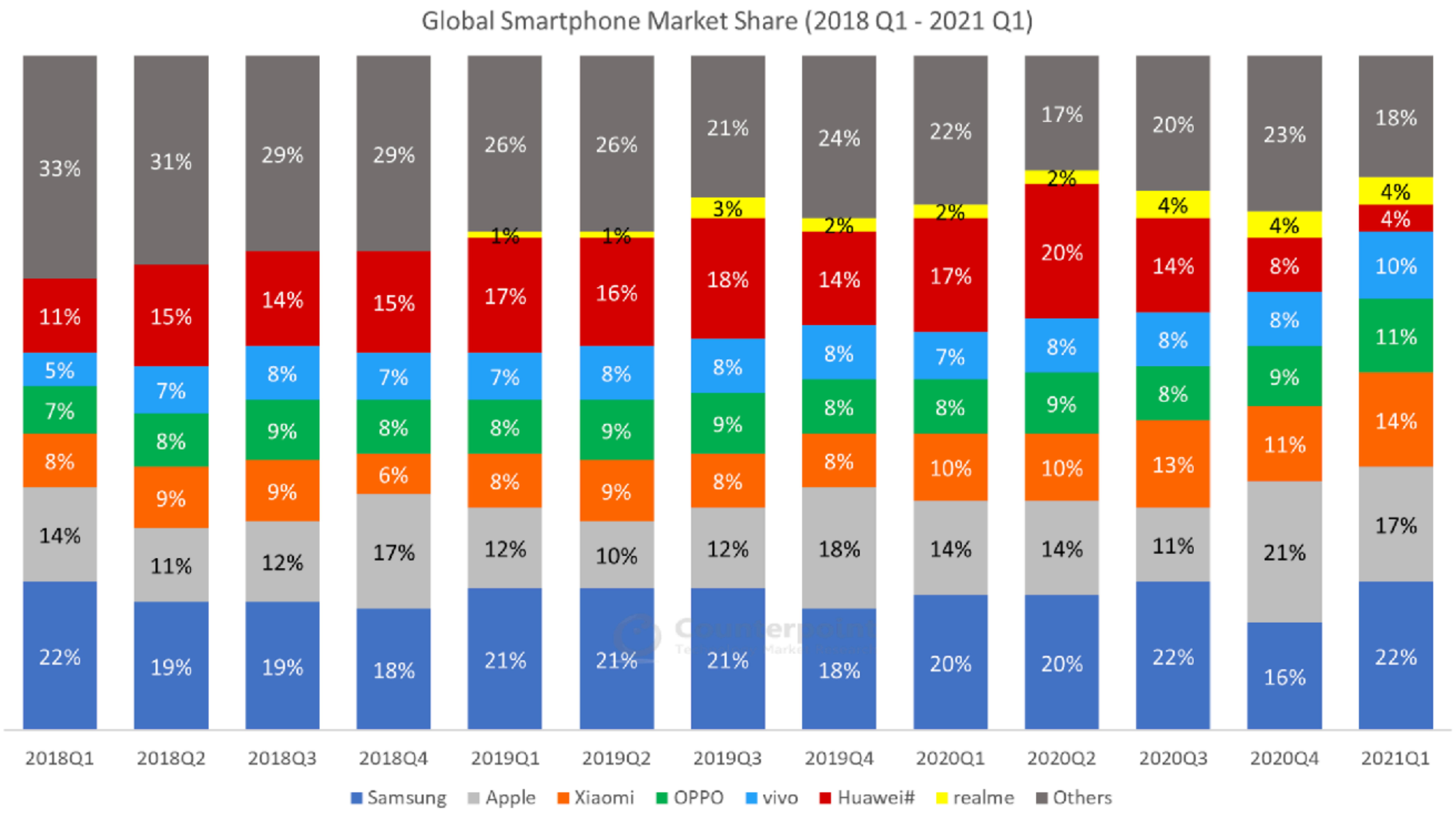
Mobile Security

Mobile is Big!

3.8B mobile users worldwide. Users spend more time on mobile than on desktops today (exact numbers iffy)

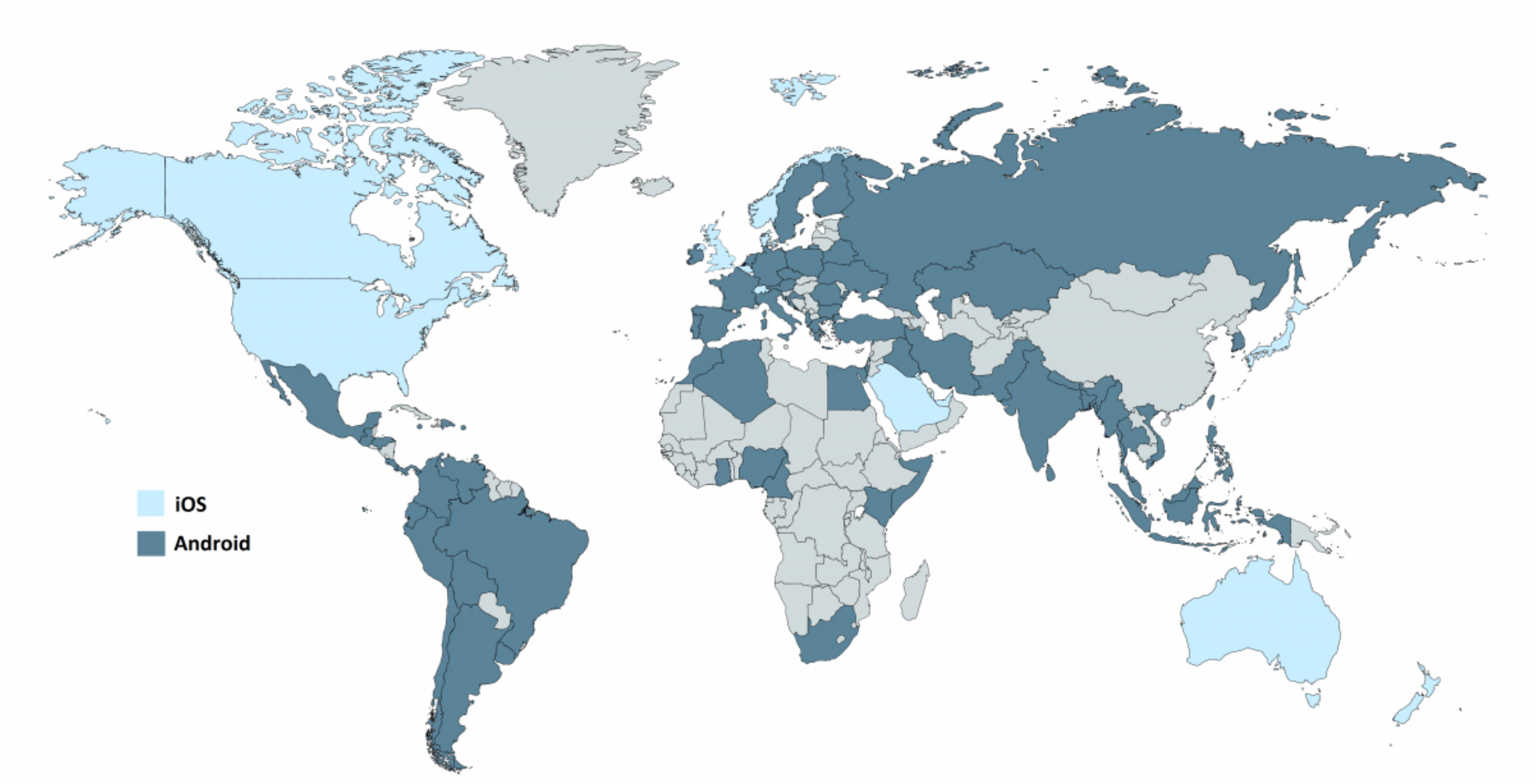


Mobile Market Share



10-20% Apple.
Most Android.

Global Bias in Market Share



What's Valuable on Phones?

Traditional (Similar to Desktop PCs)

- Steal data (e.g., contact list, email, messages, banking information, photos)
- Phishing
- Malvertising
- Join Bots

Mobile Specific

– Identify location

– Record phone calls

– Log SMS (What about 2FA SMS?)

– Send premium SMS messages

Unique Threat Model (Physical)

Powered-off devices under complete physical control of an adversary (including well-resourced nation states)

Screen locked devices under physical control of adversary (e.g. thieves)

Unlocked devices under control of different user (e.g. intimate partner abuse)

Devices in physical proximity to an adversary (with the assumed capability to control radio channels, including cellular, WiFi, Bluetooth, GPS, NFC)

Threat Model (Untrusted Code)

Android intentionally allows (with explicit consent by end users) installation of application code from arbitrary sources

Abusing APIs supported by the OS with malicious intent, e.g. spyware

Exploiting bugs in the OS, e.g. kernel, drivers, or system services

Mimicking system or other app user interfaces to confuse users

Reading content from system or other application user interfaces (e.g., screen-scrape)

Injecting input events into system or other app user interfaces

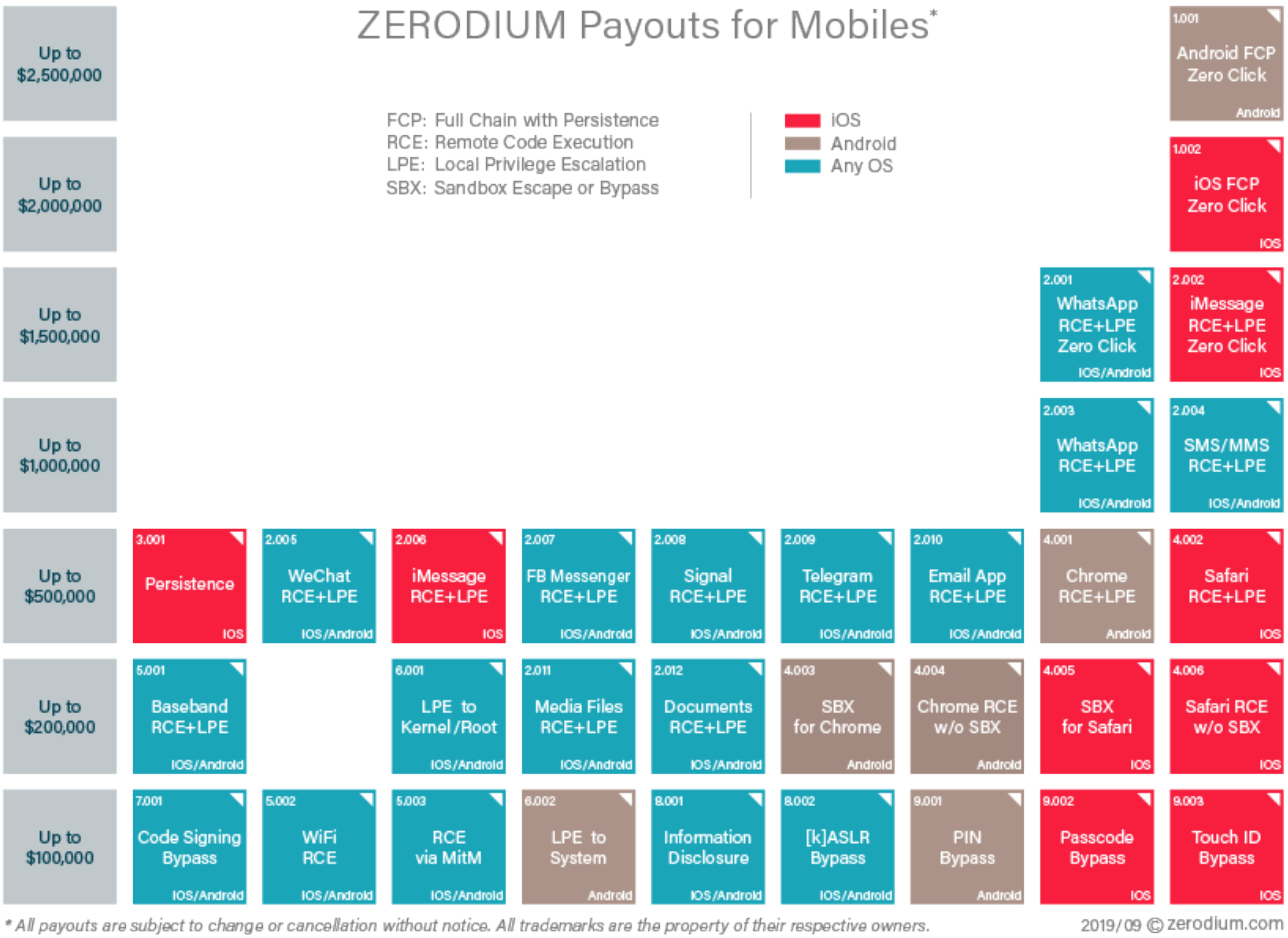
Unique Threat Model (Network)

The standard assumption of network communication under complete control of an adversary certainly also holds for Android. Assume fist hop (e.g., router) is also malicious.

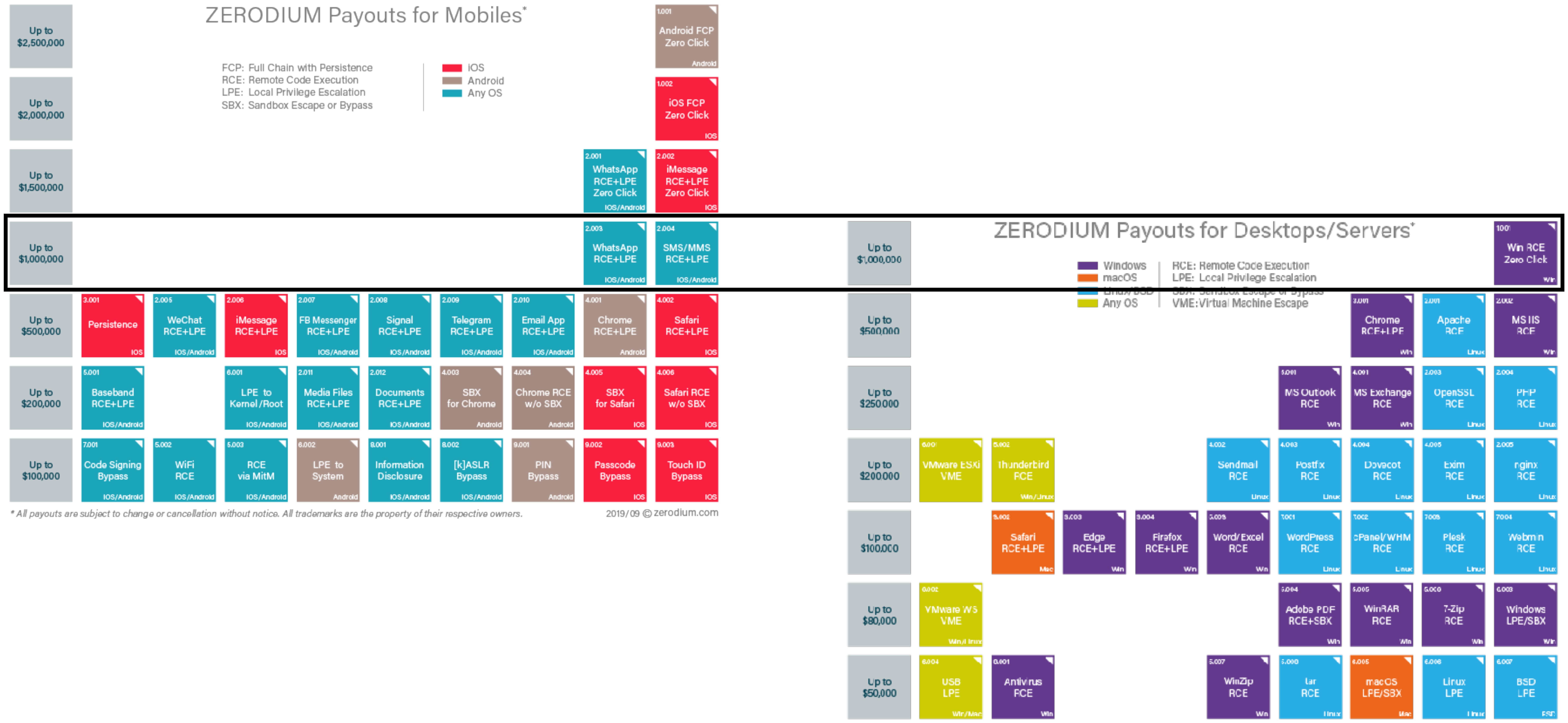
Passive eavesdropping and traffic analysis, including tracking devices within or across networks (e.g. based on MAC address or other device network identifiers)

Active manipulation of network traffic (e.g. MITM on TLS)

Mobile Exploits Very Valuable



Mobile Exploits Very Valuable



Physical Security

Unlocking Device

Typically: Need PIN, pattern, or alphanumeric password to unlock device

Some applications (e.g., banking apps) also require entering a PIN to access the app

Swipe Code Problems

Smudge attacks [Aviv et al., 2010]

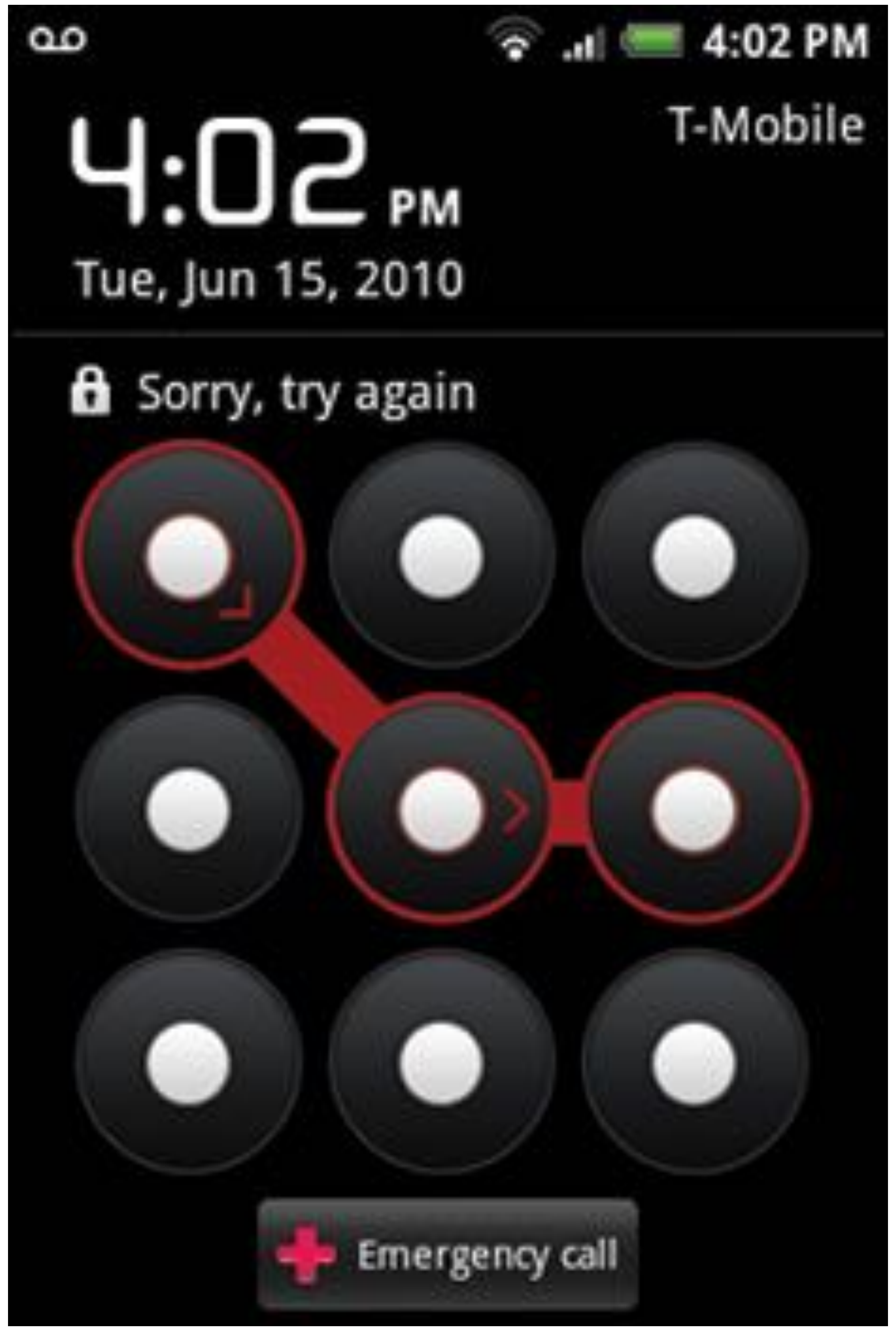
Entering pattern leaves smudge that can be detected with proper lighting

Smudge survives incidental contact with clothing

Another problem: entropy

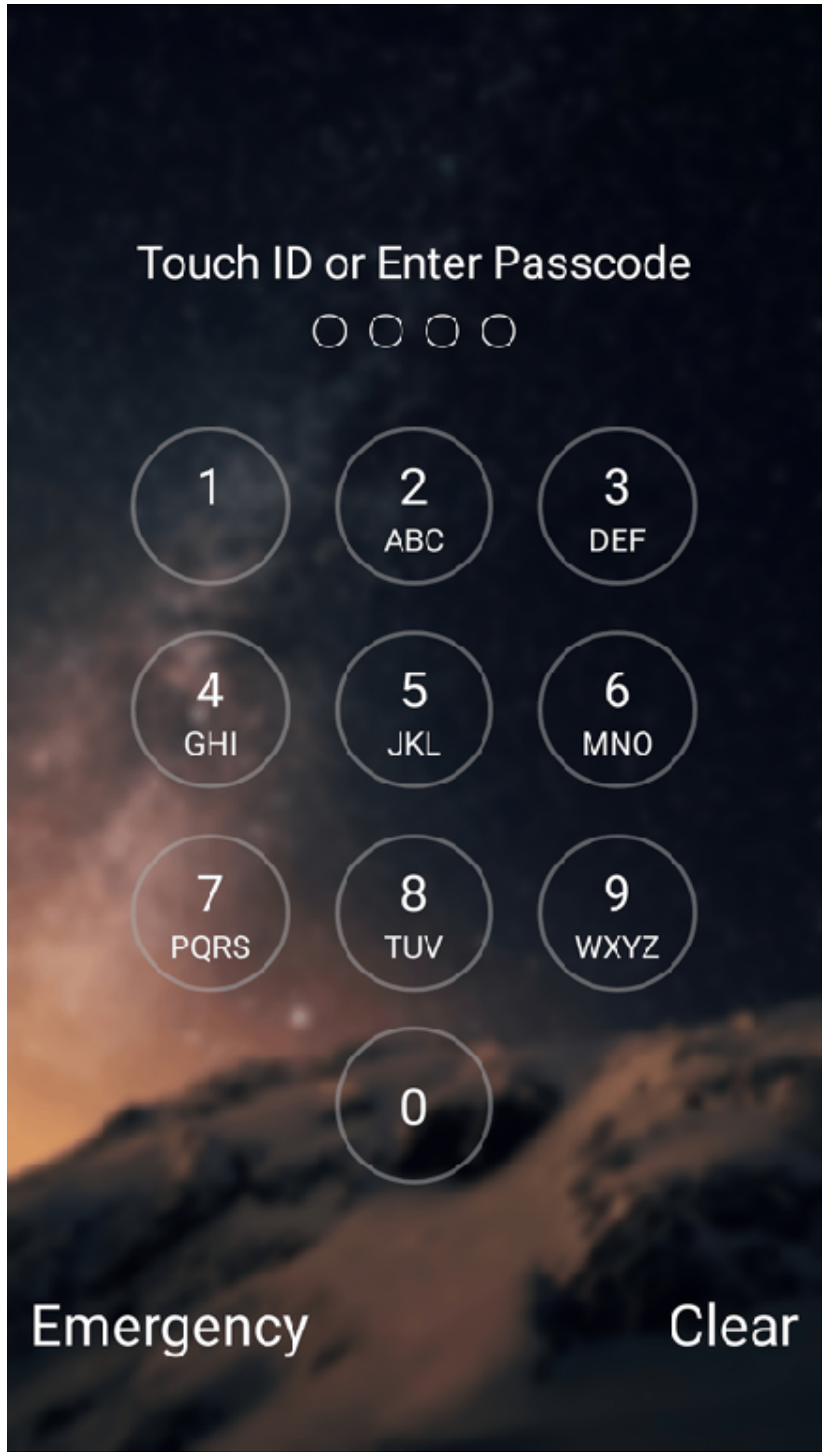
People choose simple patterns – few strokes

At most 1,600 patterns with <5 strokes



Passcodes

How do you allow a 4-6 digit PIN and be secure?



Traditional Password Hashing

Plain Text Passwords (Terrible)

- Store the password and check match against user input
- Don’t trust anything that can provide you your password

Traditional Password Hashing

Plain Text Passwords (Terrible)

- Store the password and check match against user input
- Don’t trust anything that can provide you your password

Store Password Hash (Bad)

- Store SHA-1(pw) and check match against SHA-1(input)
- Weak against attacker who has hashed common passwords

Traditional Password Hashing

Plain Text Passwords (Terrible)

- Store the password and check match against user input
- Don’t trust anything that can provide you your password

Store Password Hash (Bad)

- Store SHA-1(pw) and check match against SHA-1(input)
- Weak against attacker who has hashed common passwords

Store Salted Hash (Better)

- Store **`(r, SHA-1(pw||r))`** and check against **`SHA-1(input||r)`**
- Prevents attackers from pre-computing password hashes

Modern Password Hashing

Store Salted Hash (Best)

- Store (r, **H**(pw || r)) and check match against **H**(input || r)
- Prevents attackers from pre-computing password hashes

Making sure to choose an **H** that's expensive to compute:

SHA-512: 3,235 MH/s
SHA-3 (Keccak): 2,500 MH/s
BCrypt: 43,551 H/s

Use one of bcrypt, scrypt, or pbkdf2 when building an application

iPhone Password Hashing

Come up with a password hashing approach where 4-6 digits takes a very long time to crack, even if the device is physically compromised…

Additional Constraints:

- Lots of computation uses up battery (limited resource)!
- Physical access allows copying secret off and cracking remotely

Secure Enclave

iPhones have a second secure processor known as "secure enclave"

- Memory is inaccessible to normal OS
- Secure boot process that ensures its software is signed
- Each secure enclave has an AES key burned in at manufacture.

Processor has instructions that allow encrypting and decrypting content using the stored key, but the key itself is never accessible

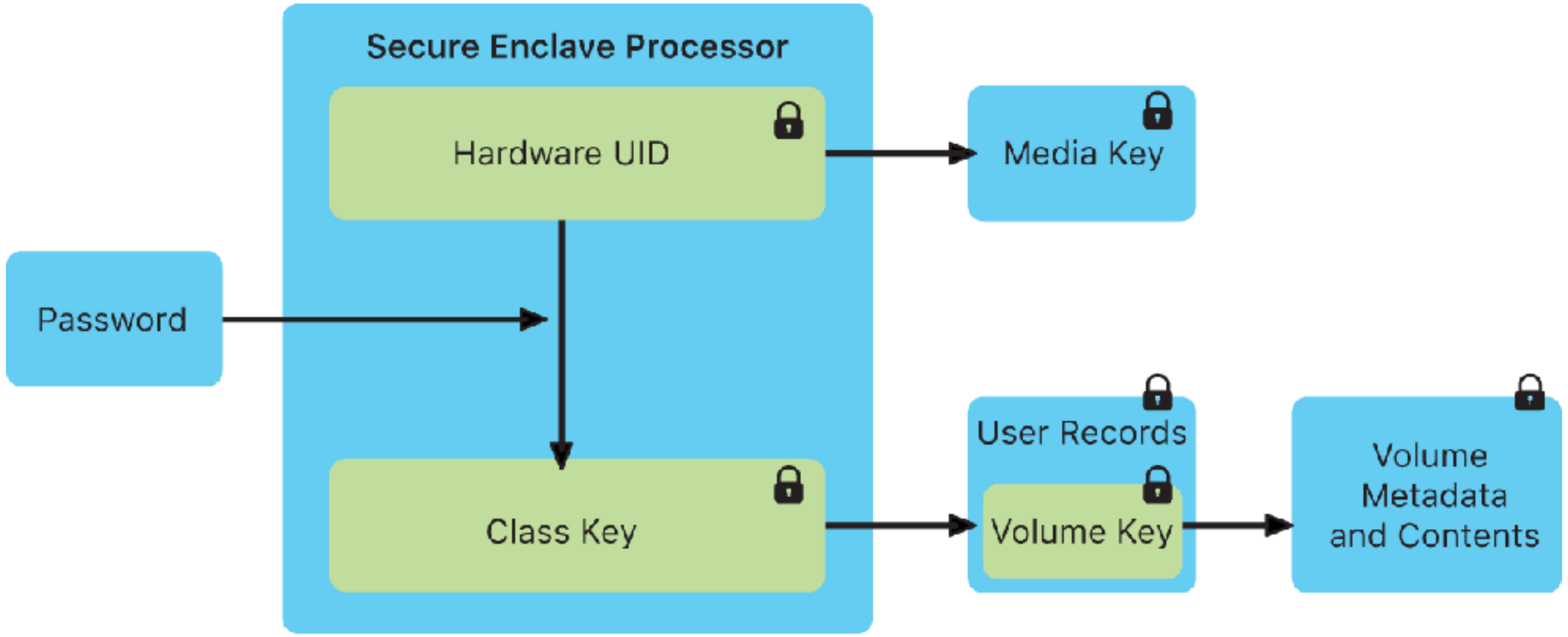
iPhone Unlocking

User passcode is intertwined with AES key fused into secure enclave (known as UID) when it is entered by the user

Imagine: key = $\text{Encrypt}_{\text{UID}}$(passcode).

This means that the 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.

iPhone Unlocking Key



What prevents asking secure enclave repeatedly to try different passwords?

The passcode is entangled with the device's UID many times —requires approximately 80ms per password guess.

Imagine: $\text{Encrypt}_{\text{UID}}(\text{Encrypt}_{\text{UID}}(\text{Encrypt}_{\text{UID}}(\text{passcode})\ldots))$

iPhone Unlock Time Estimate

At 80ms per password check…

- 5.5 years to try all 6 digits pins
- 5 failed attempts ⇒ 1min delay, 9 failures ⇒ 1 hour delay
    - >10 failed attempts ⇒ erase phone

All of this enforced by firmware on the secure enclave itself — cannot be changed by any malware that controls iOS

FBI–Apple Encryption Dispute

After the San Bernardino shooting in 2016, FBI tried to compel Apple to “unlock” iPhone. What were they specifically requesting?

Not possible to make password guessing any faster—innately dependent on performance of burned-in AES key

FBI–Apple Encryption Dispute

Remember…

- 5 failed attempts ⇒ 1min delay, 9 failures ⇒ 1 hour delay
    - >10 failed attempts ⇒ erase phone

This is managed by code on the secure enclave, which *can* be updated by Apple, not managed in hardware.

Technical Details

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

1. "it will bypass or disable the auto-erase function whether or not it has been enabled" (this user-configurable feature of iOS 8 automatically deletes keys needed to read encrypted data after ten consecutive incorrect attempts)
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"

What happened?

Apple planned to fight the order, "*The United States government has demanded that Apple take an unprecedented step which threatens the security of our customers. We oppose this order, which has implications far beyond the legal case at hand. This moment calls for public discussion, and we want our customers and people around the country to understand what is at stake.*"

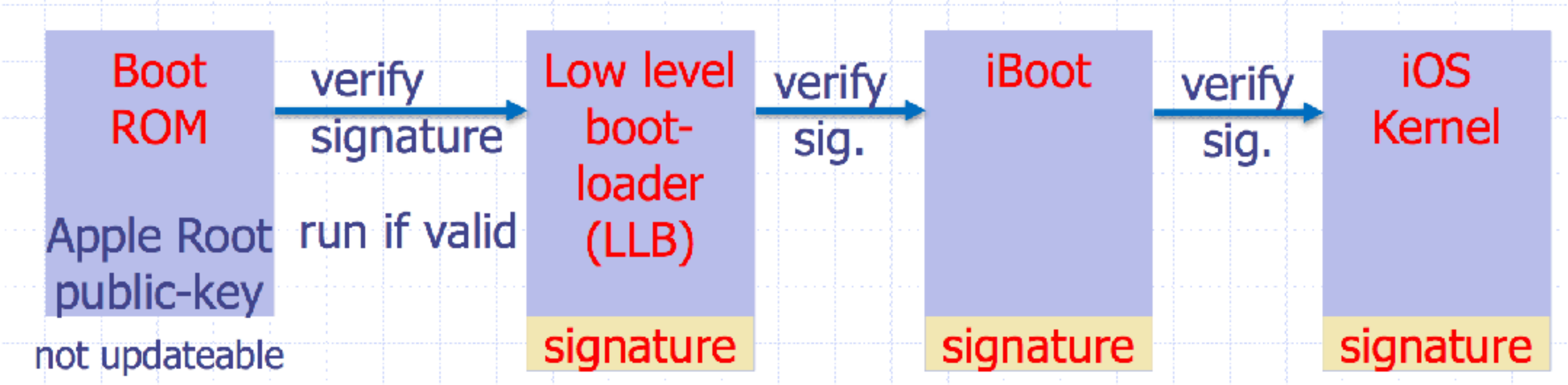
One day before hearing, FBI dropped the request, saying a third party had demonstrated a possible way to unlock the iPhone in question.

Secure Boot Chain

Why couldn't the FBI just upload their own firmware onto the secure enclave?

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.



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).

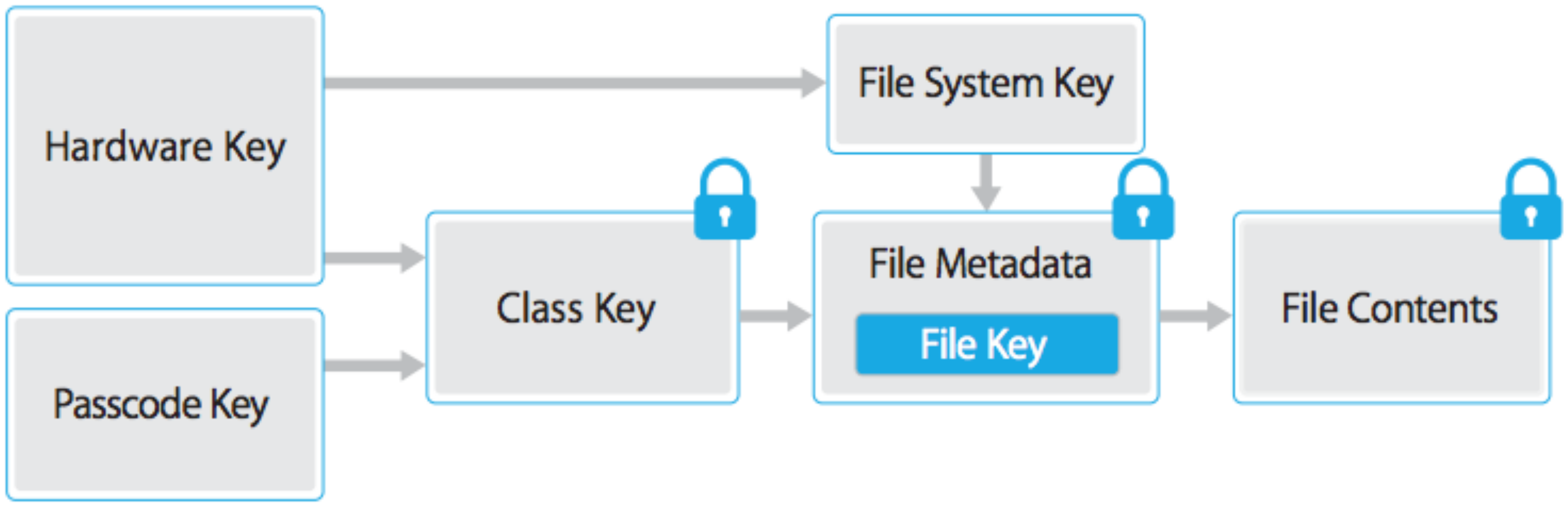
Apple signs device-personalized message allowing update, which boot loader verifies.

FaceID/TouchID

Files are encrypted through a hierarchy of encryption keys

Application files written to Flash are encrypted:

- 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



FaceID/TouchID

Files are encrypted through a hierarchy of encryption keys

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.

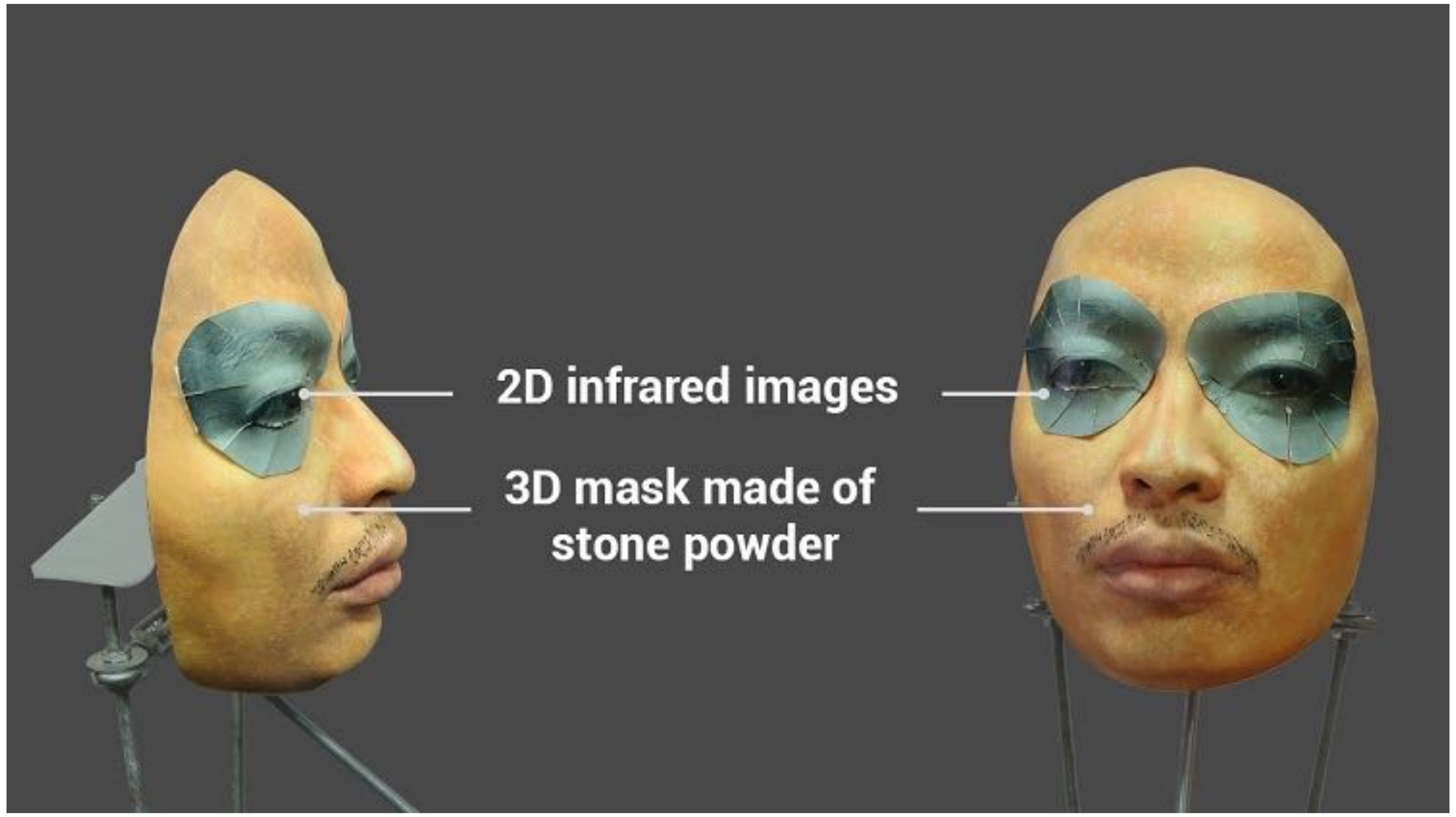
How Secure is TouchID?

Easy to build a fake finger if you have someone's fingerprint

- Several demos on YouTube. ~20 min
- Similar work on FaceID

The problem: fingerprints are not secret. Cannot replace.

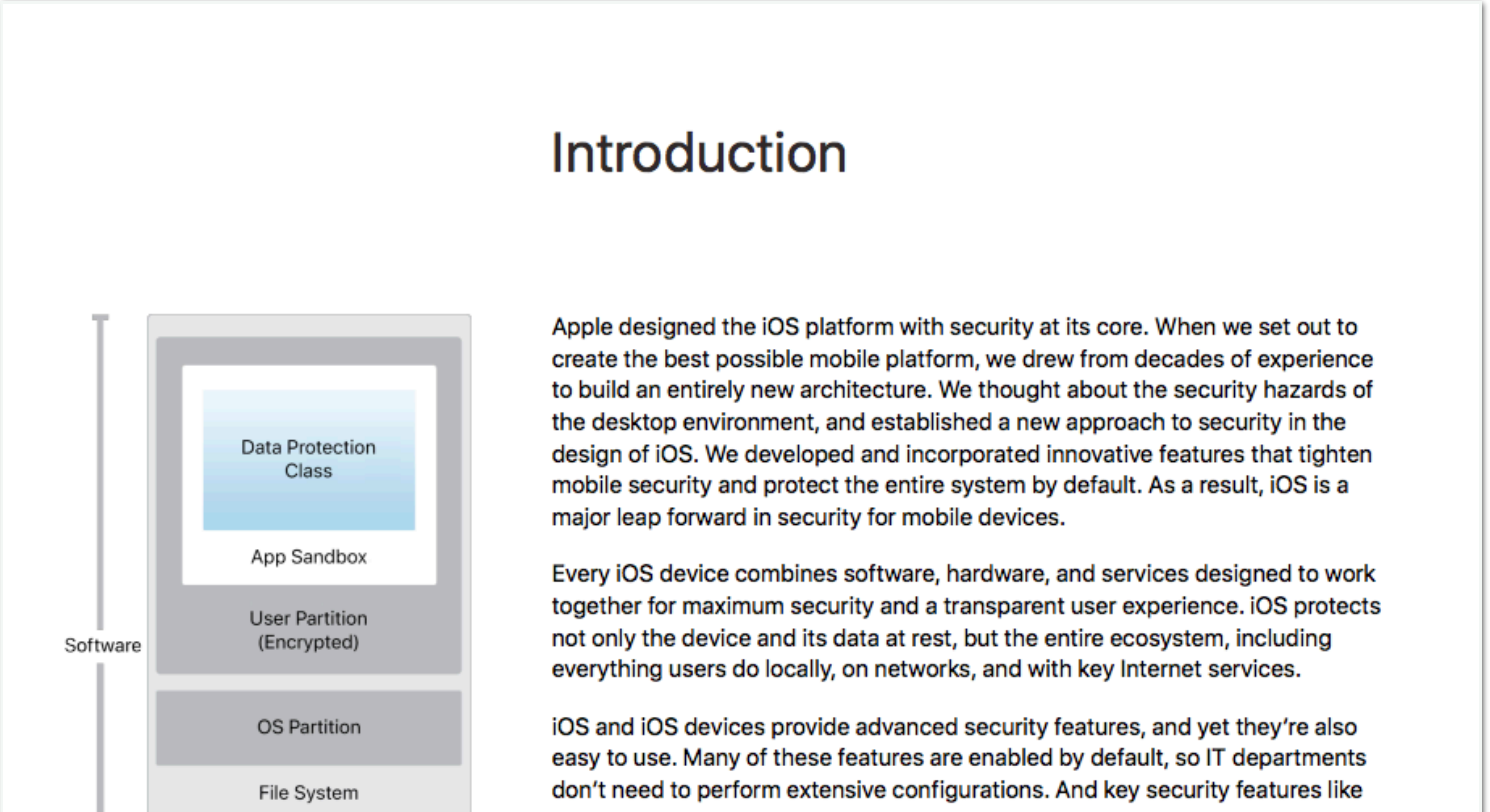
Convenient, but more secure solutions exist, e.g., unlock phone via bluetooth using a wearable device



More Information

iOS Security

https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf



Introduction

Apple designed the iOS platform with security at its core. When we set out to create the best possible mobile platform, we drew from decades of experience to build an entirely new architecture. We thought about the security hazards of the desktop environment, and established a new approach to security in the design of iOS. We developed and incorporated innovative features that tighten mobile security and protect the entire system by default. As a result, iOS is a major leap forward in security for mobile devices.

Every iOS device combines software, hardware, and services designed to work together for maximum security and a transparent user experience. iOS protects not only the device and its data at rest, but the entire ecosystem, including everything users do locally, on networks, and with key Internet services.

iOS and iOS devices provide advanced security features, and yet they're also easy to use. Many of these features are enabled by default, so IT departments don't need to perform extensive configurations. And key security features like

Bring Your Own Device (BYOD)

Many companies are now allowing users to bring/use their own personal devices — company data resides on devices

In the past, enterprise workstations were centrally managed.

How do you handle when users want to bring their own devices?

Mobile Device Management

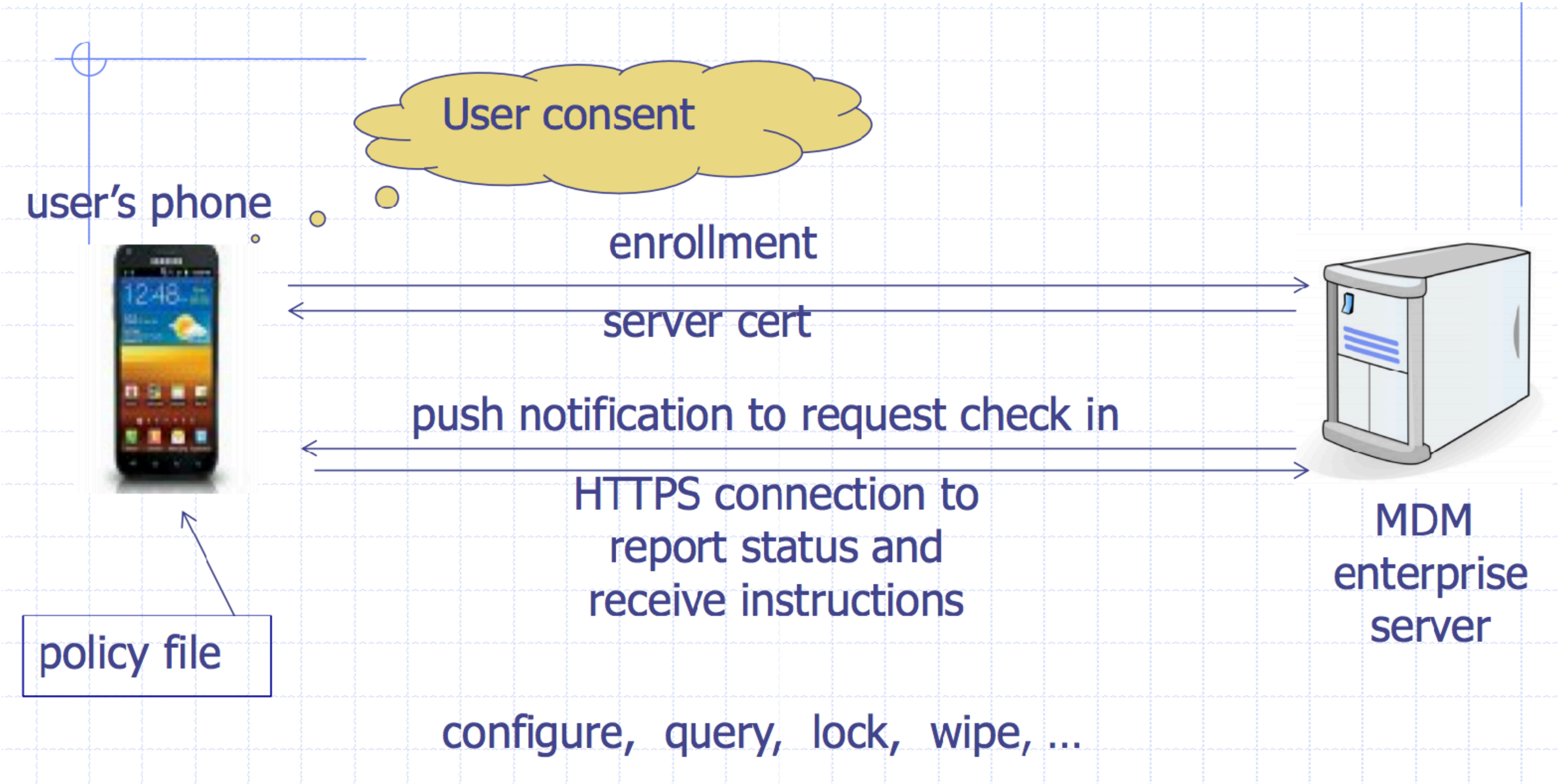
Manage mobile devices across organization

Consists of central server and client-side software. Now part of mobile OSes too.

Allows:

- Diagnostics, repair, and update
- Backup and restore
- Policy enforcement (e.g. only allowed apps)
- Remote lock and wipe
- GPS Tracking

Sample MDM Enrollment



Mobile Malware

What’s Different?

Applications are isolated

- Each runs in a separate execution context
- No default access to file system, devices, etc.
- Different than traditional OSes where multiple applications run with the same user permissions!

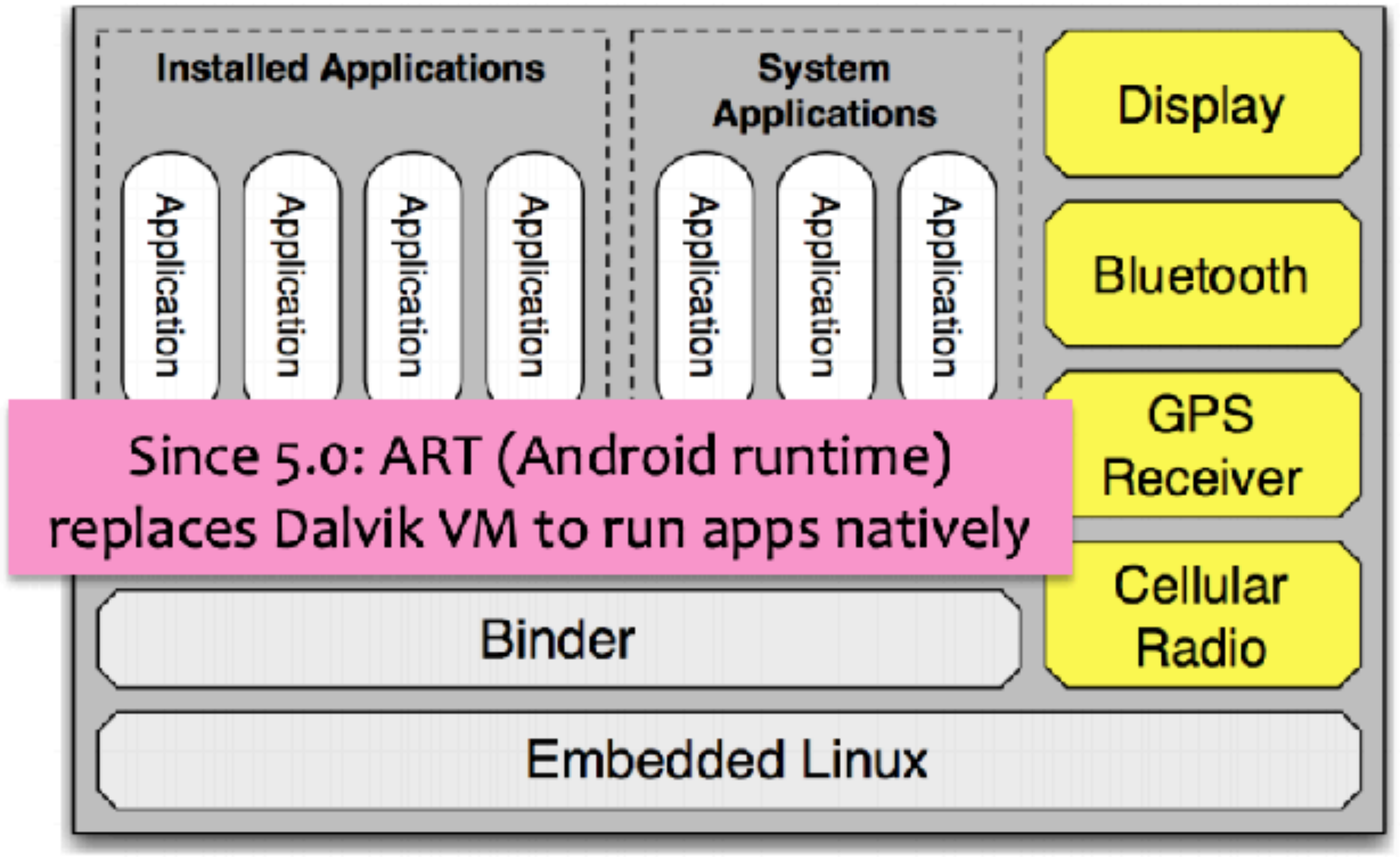
Applications are installed via App Store (and malware spreads)

- Market: Vendor controlled (Apple) / open (Android)
- User approval of permissions

Android Isolation

Based on Linux with sandboxes (SE Linux)

- Apps run as separate UIDs, in separate processes.
- Memory corruption errors only lead to arbitrary code execution in application, not complete system compromise!
- Can still escape sandbox – must compromise Linux kernel



What is Rooting?

Allows user to run applications with root privileges, e.g., modify/delete system files and app, CPU, network management

Done by exploiting vulnerability in firmware to install a custom OS or firmware image

Double-edged sword… lots of malware only affects rooted devices

Examples of Malware

DroidDream (Android)

- Over 58 apps uploaded to Google app market
- Conducts data theft; send credentials to attackers

Attacked vulnerability in Android itself

Zitmo (Symbian, BlackBerry, Windows, Android)

- Poses as mobile banking application
- Captures info from SMS – steal banking 2FA codes
- Works with Zeus botnet

Malicious application that tricked users

Ikee (iOS)

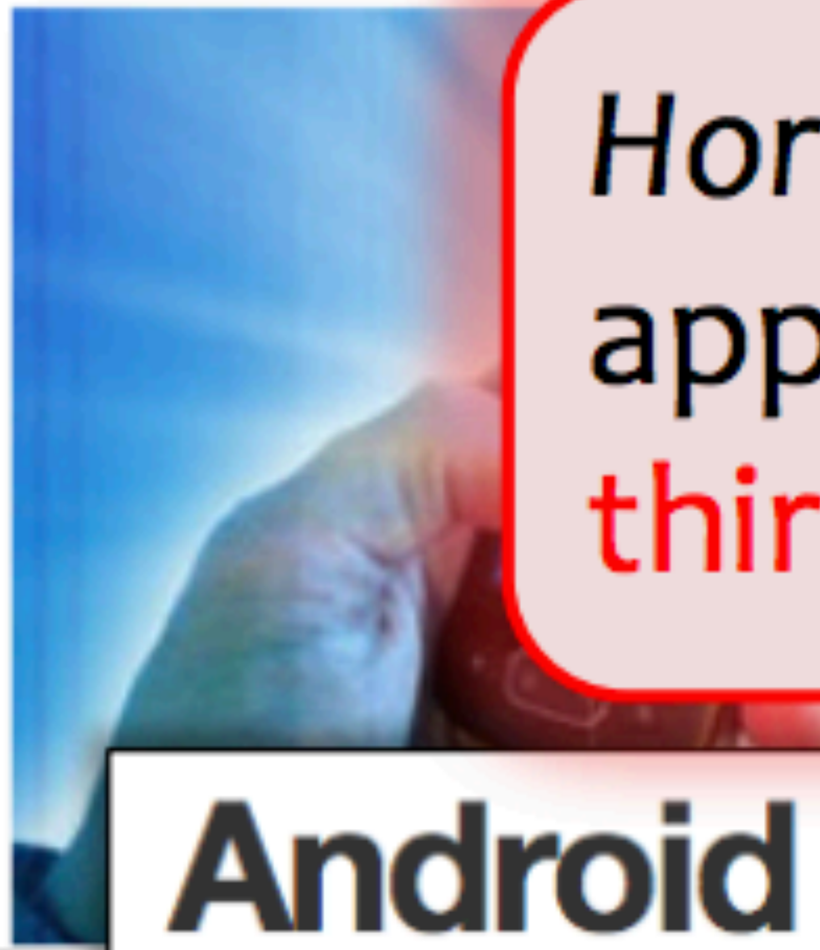
- Worm capabilities (targeted default ssh password)
- Worked only on jailbroken phones with ssh installed

Attacked vulnerability in rooted iPhones

Legitimate Apps Too…

Top Mobile Apps Overwhelmingly Leak Private Data: Study

By Robert Lemos | Posted 2013-07-31 Email Print



paid apps
pplication-

isk more often
more likely to
applications,
according to a survey of the top 400 mobile applications

Hornyack et al.: 43 of 110 Android applications sent location or phone ID to third-party advertising/analytics servers.

Android flashlight app tracks users via GPS, FTC says hold on

By Michael Kassner in IT Security, December 11, 2013, 9:49 PM PST

Challenges with Isolated Apps

So mobile platforms isolate applications for security, but….

1) **Permissions:** How can applications access sensitive resources?

2) **Communication:** How can applications communicate with each other?

Permission Granting Problem

Smartphones (and other modern OSes) try to prevent such attacks by limiting applications' default access to:

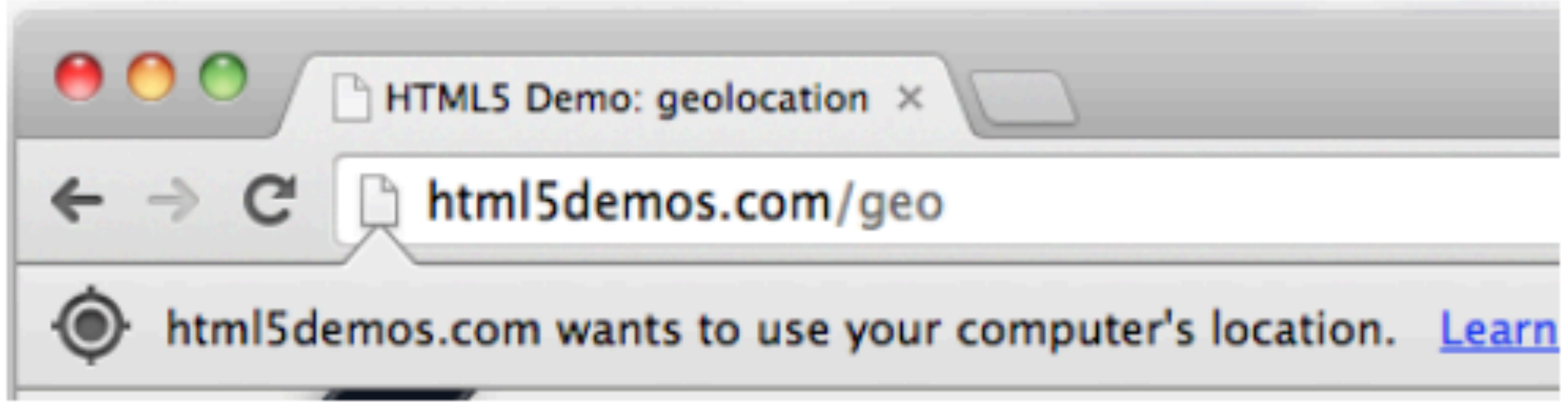
- System Resources (clipboard, file system)
- Devices (e.g., camera, GPS, phone, …)

How should operating system grant permissions to applications?

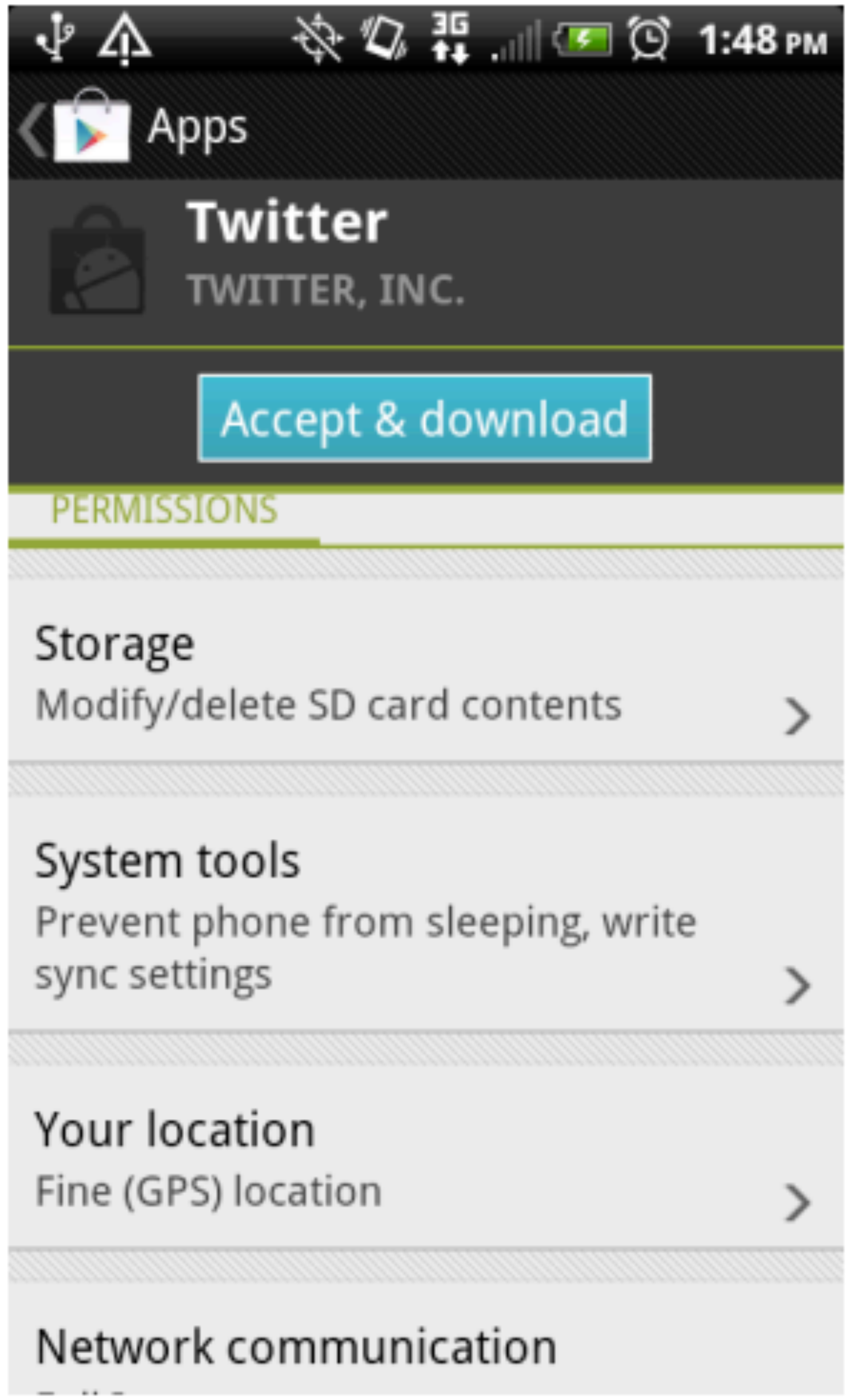
Standard approach: Ask the user.

State of the Art

Prompts (time-of-use)

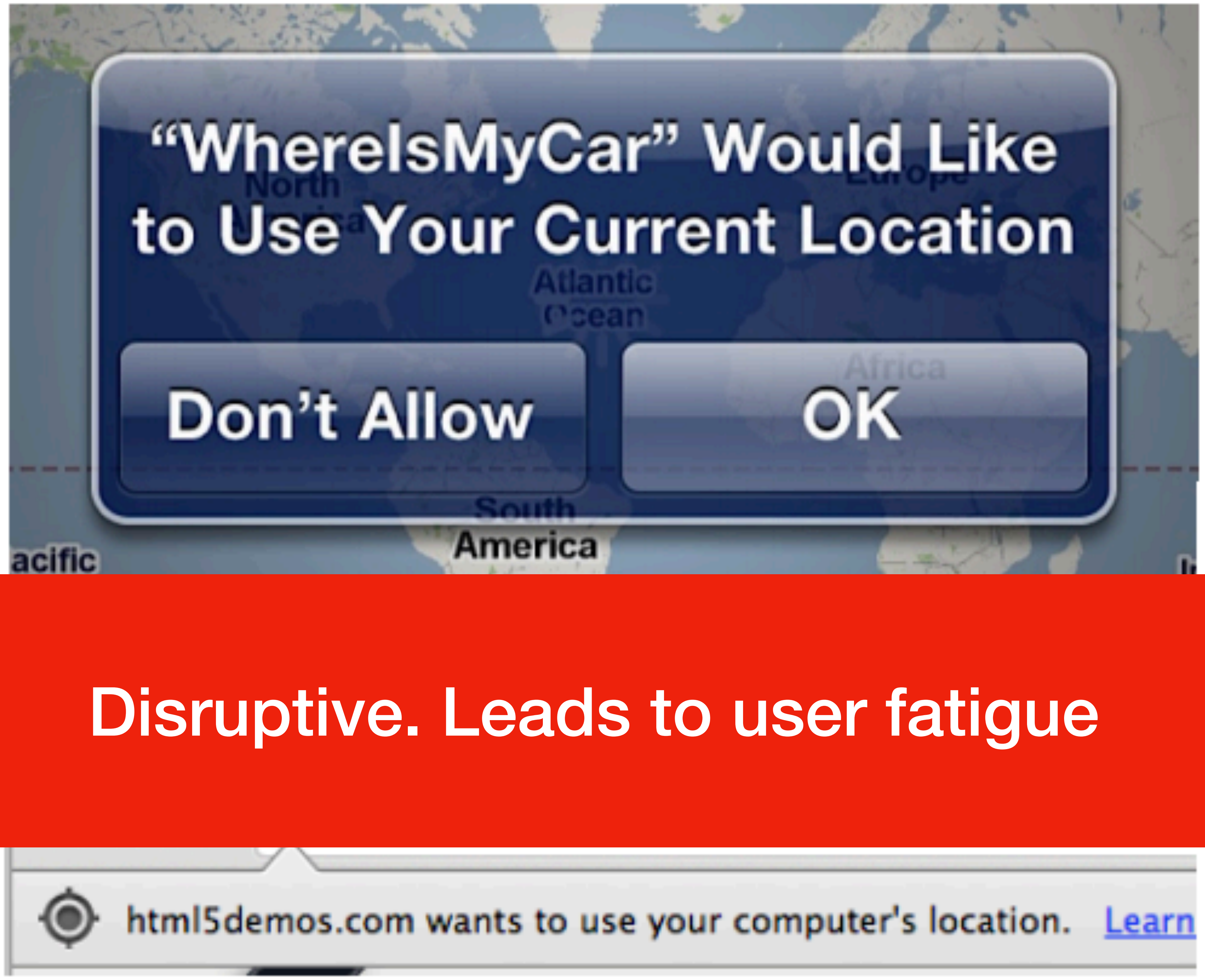


Manifests (install-time)



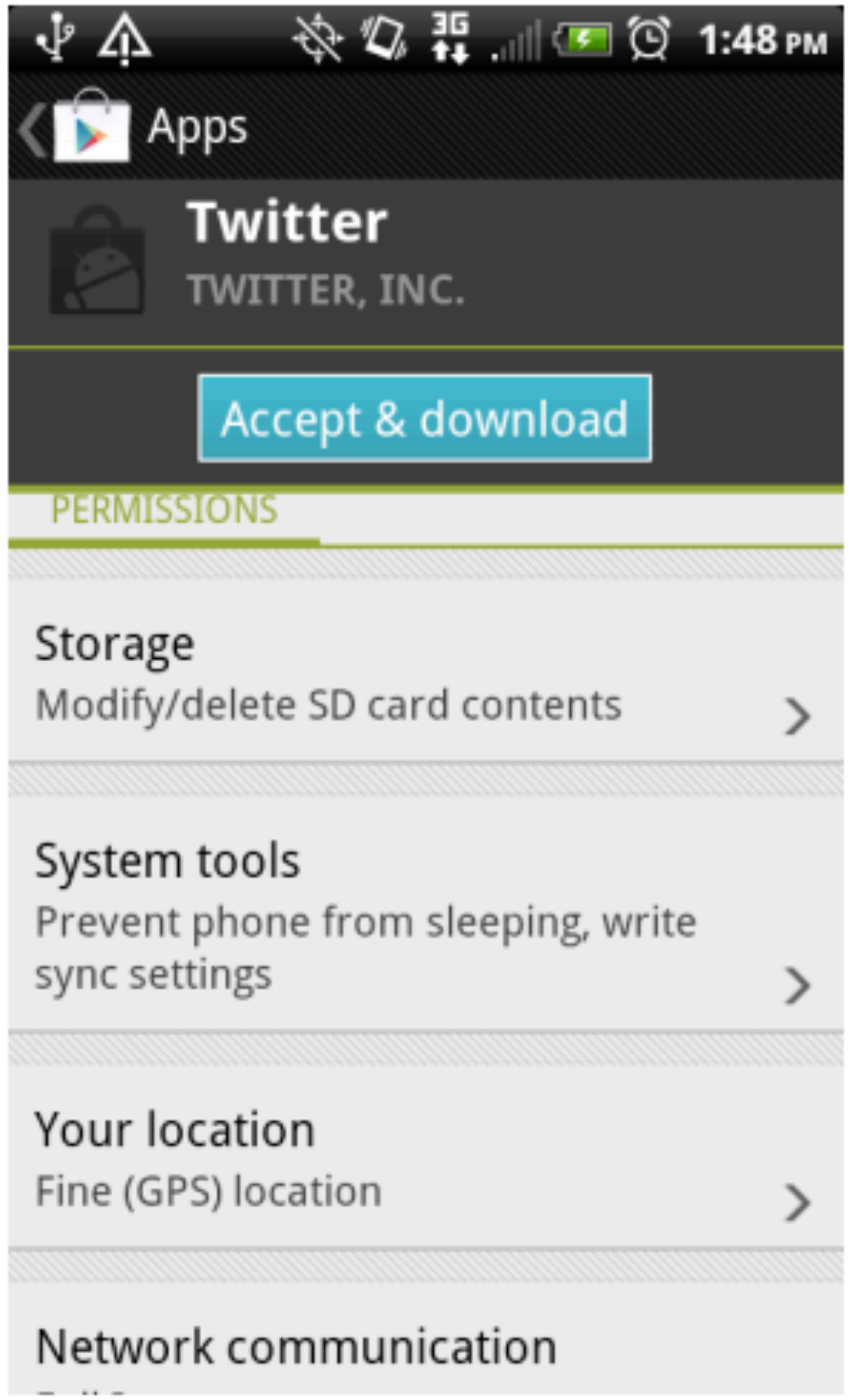
State of the Art

Prompts (time-of-use)



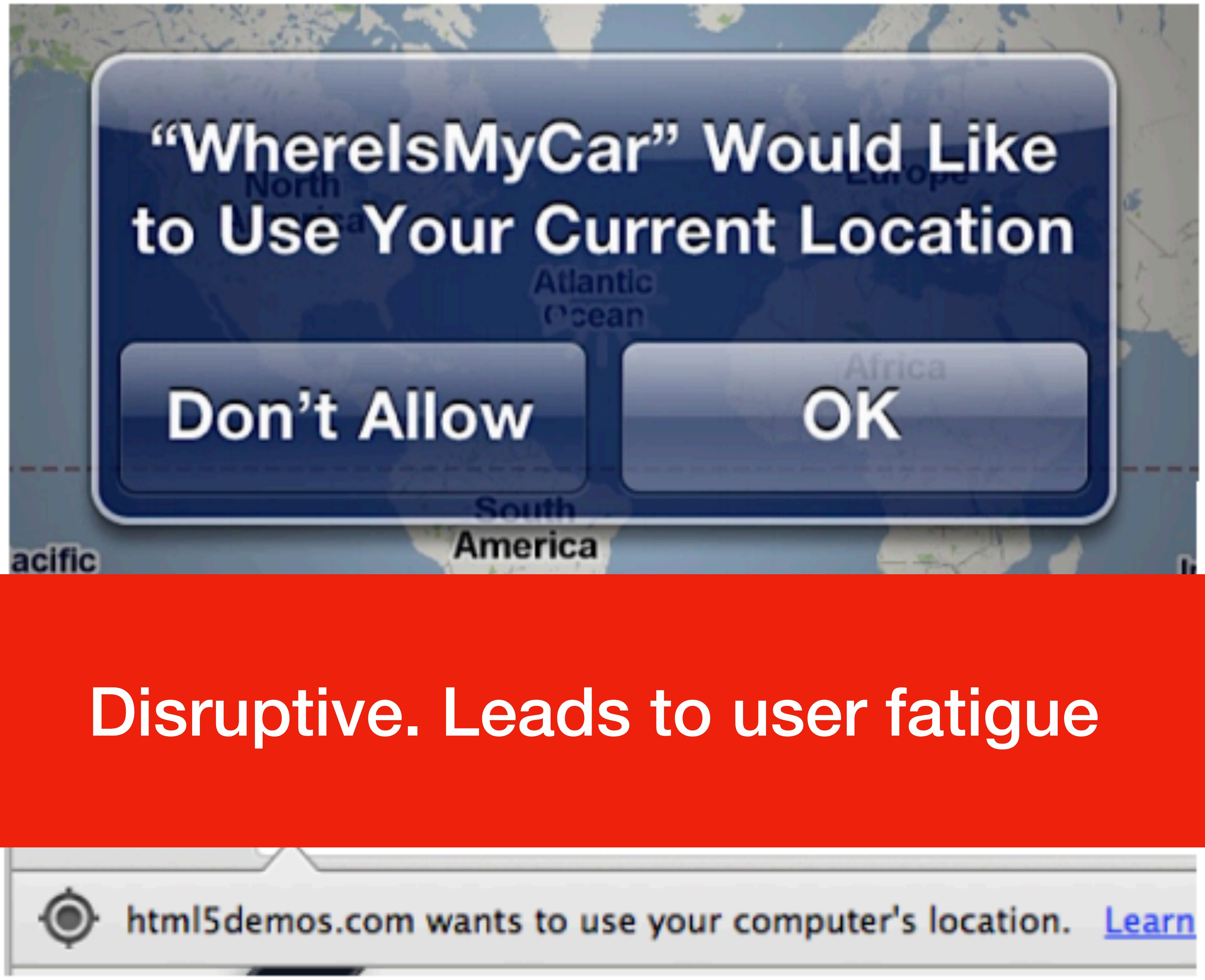
Disruptive. Leads to user fatigue

Manifests (install-time)



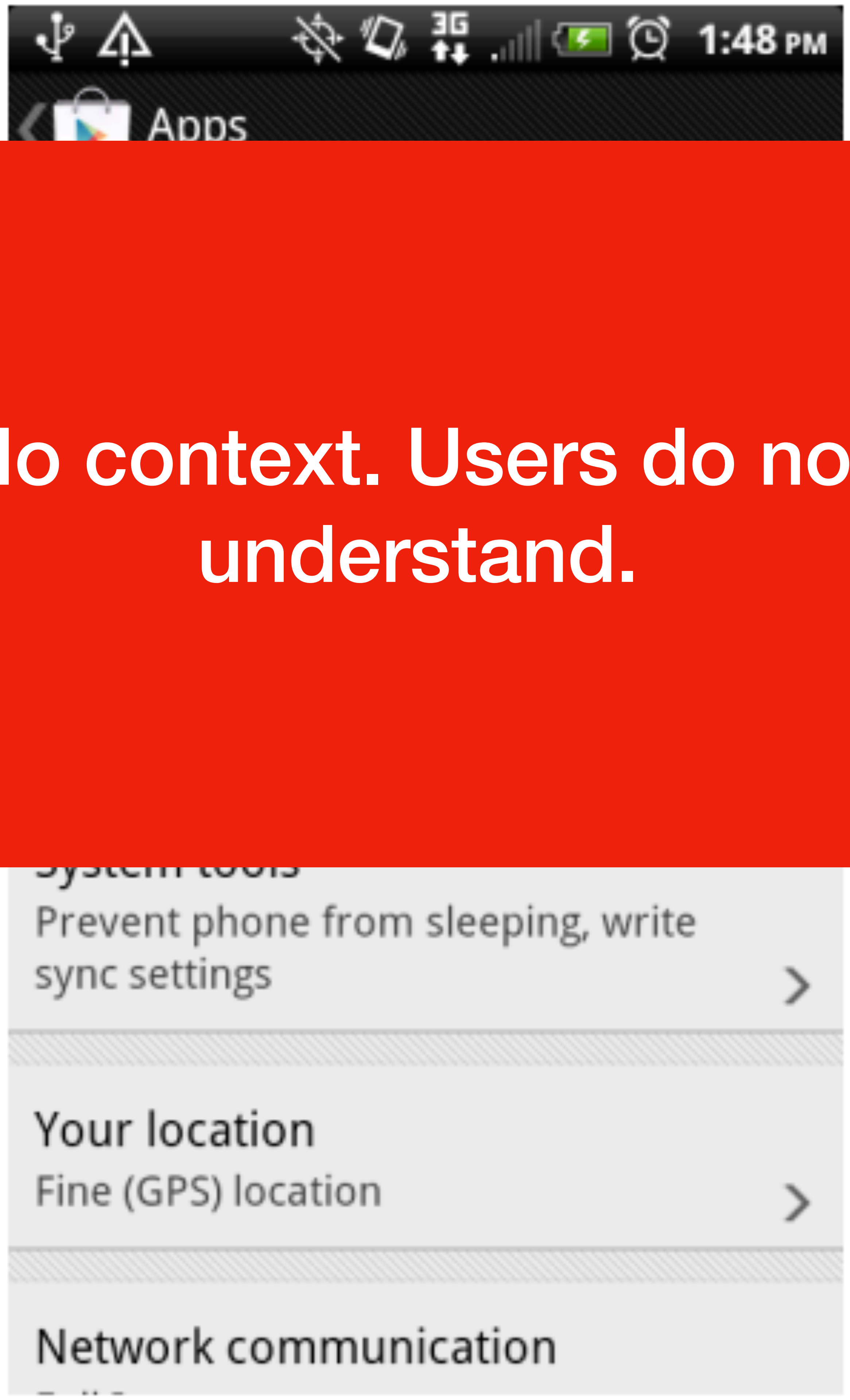
State of the Art

Prompts (time-of-use)



Disruptive. Leads to user fatigue

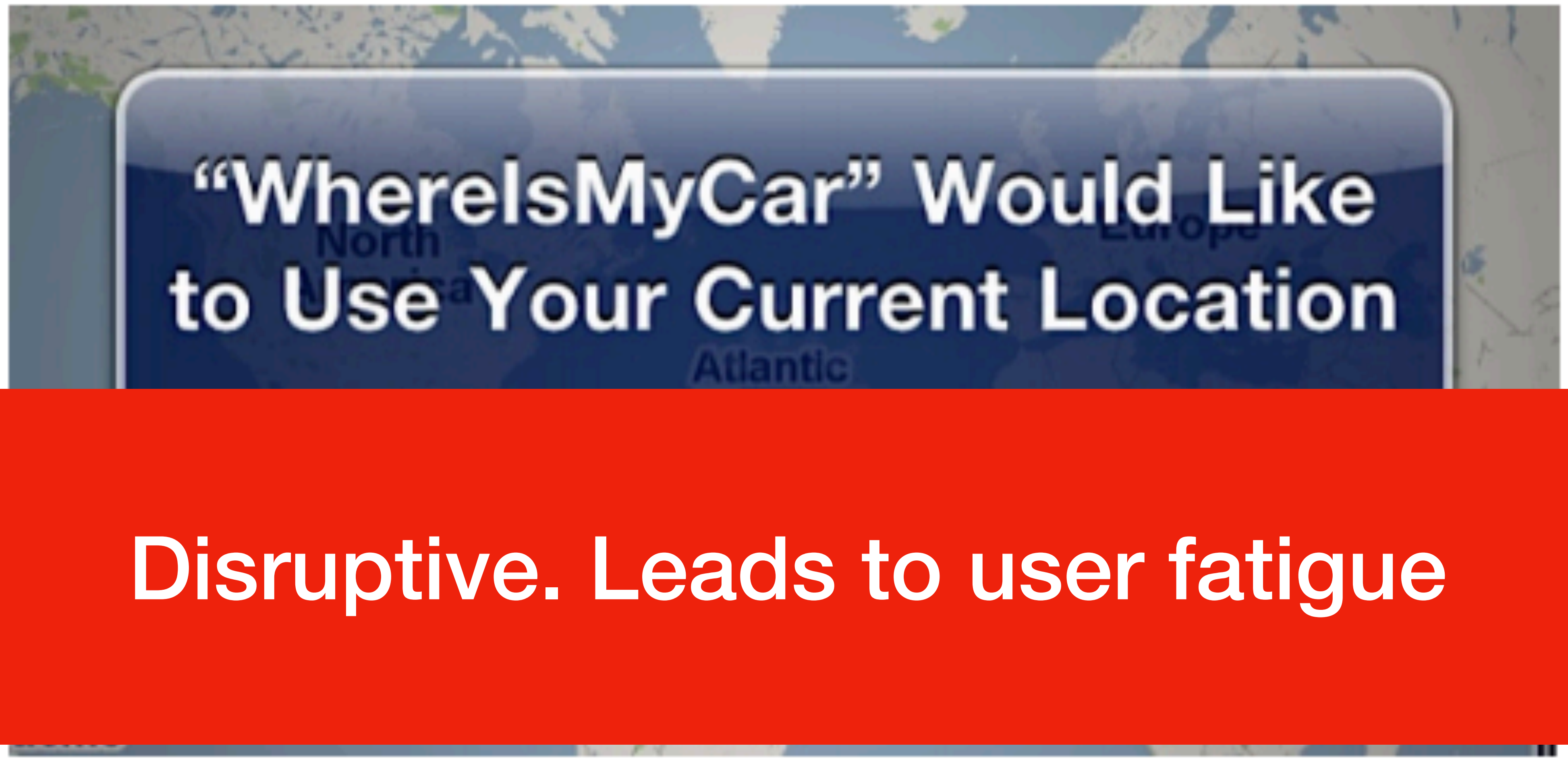
Manifests (install-time)



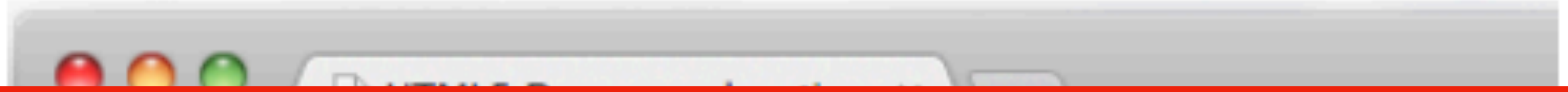
No context. Users do not understand.

State of the Art

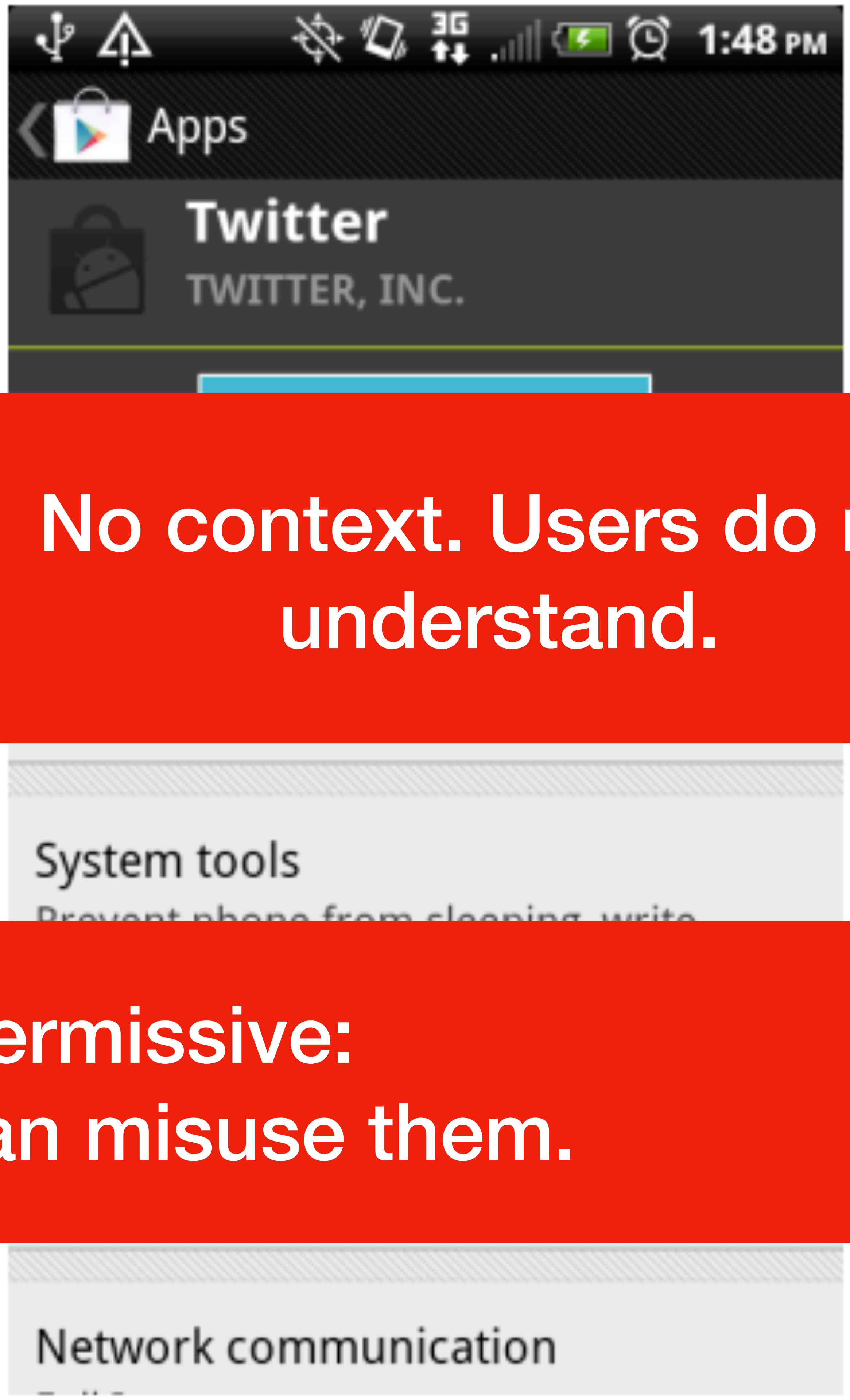
Prompts (time-of-use)



Disruptive. Leads to user fatigue



Manifests (install-time)

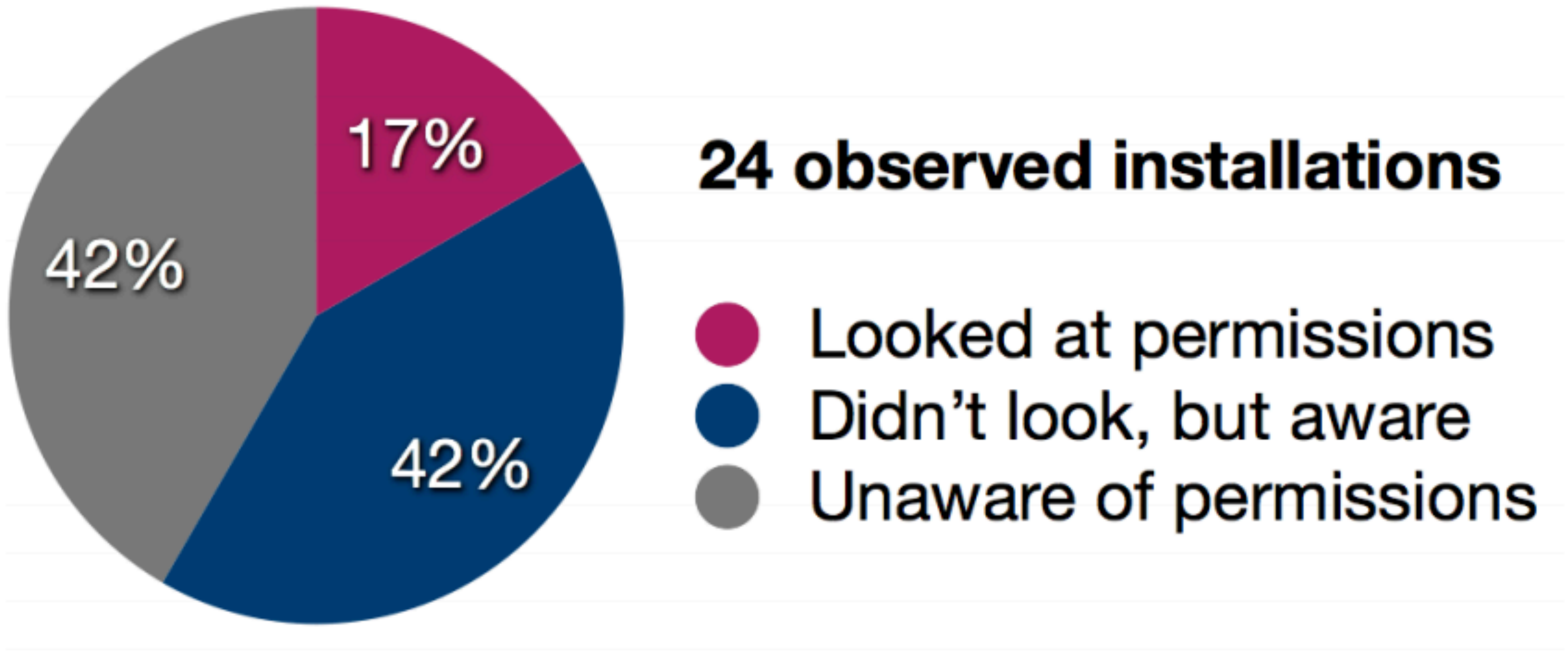


No context. Users do not understand.

In practice, both are overly permissive:
Once granted permissions, apps can misuse them.

Are Manifests Usable? (Felt et al)

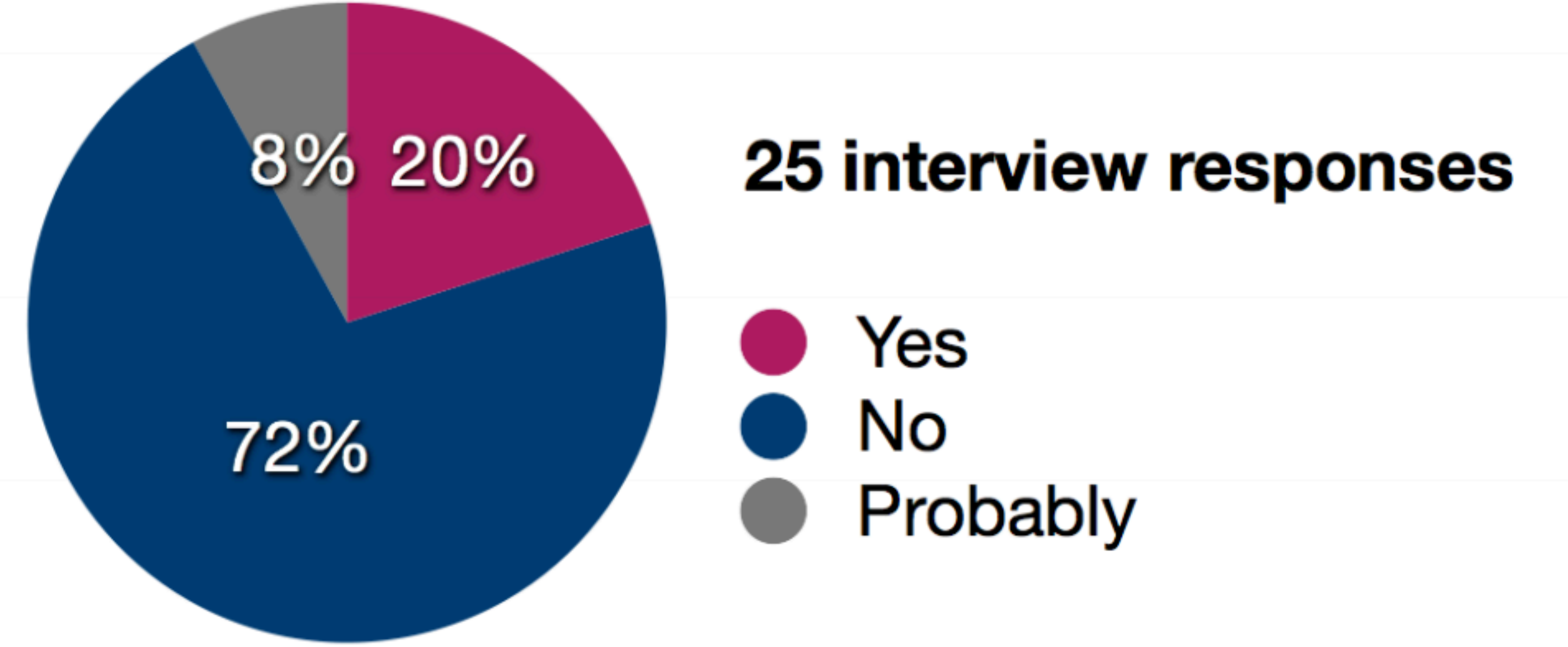
Do users pay attention to permissions?



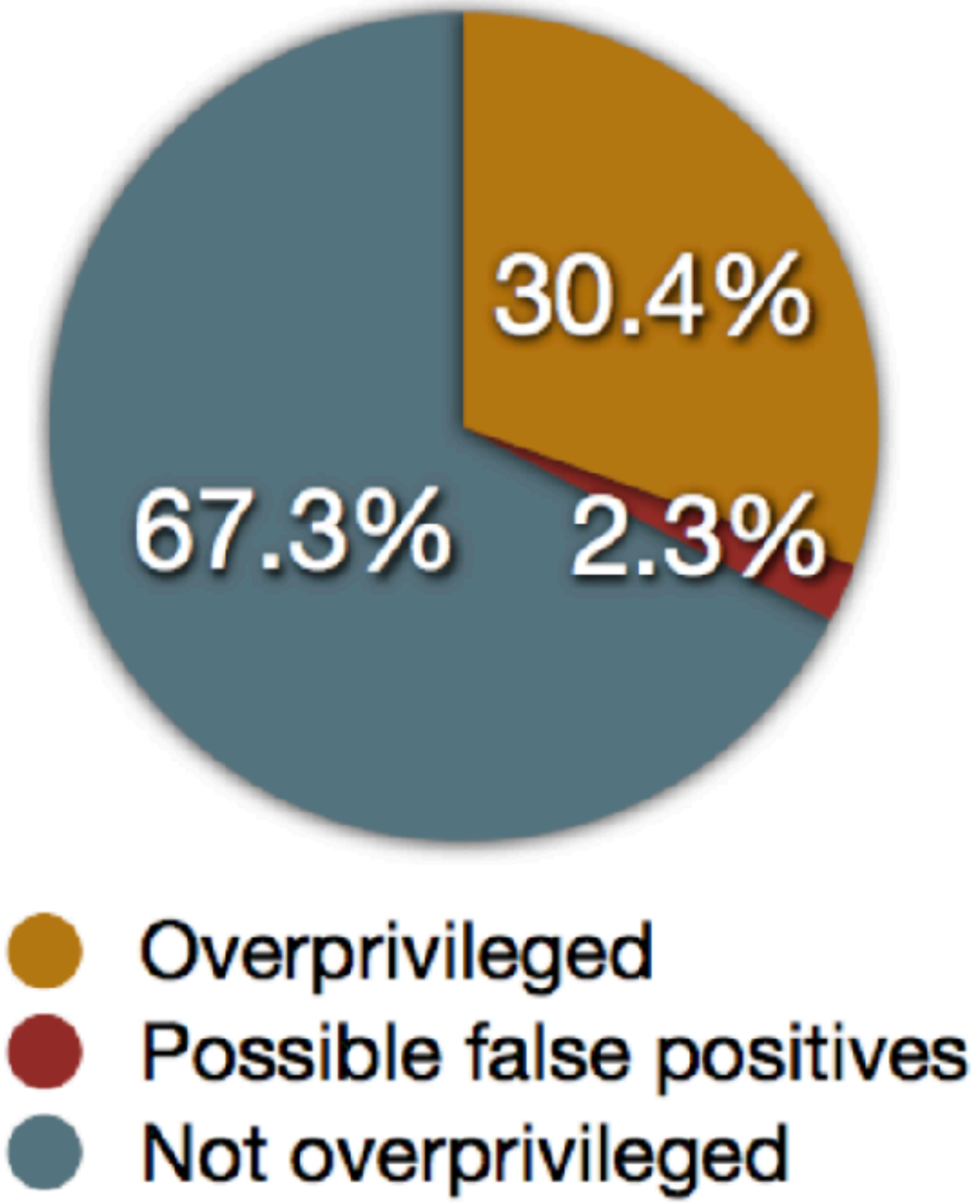
… but 88% of users looked at reviews.

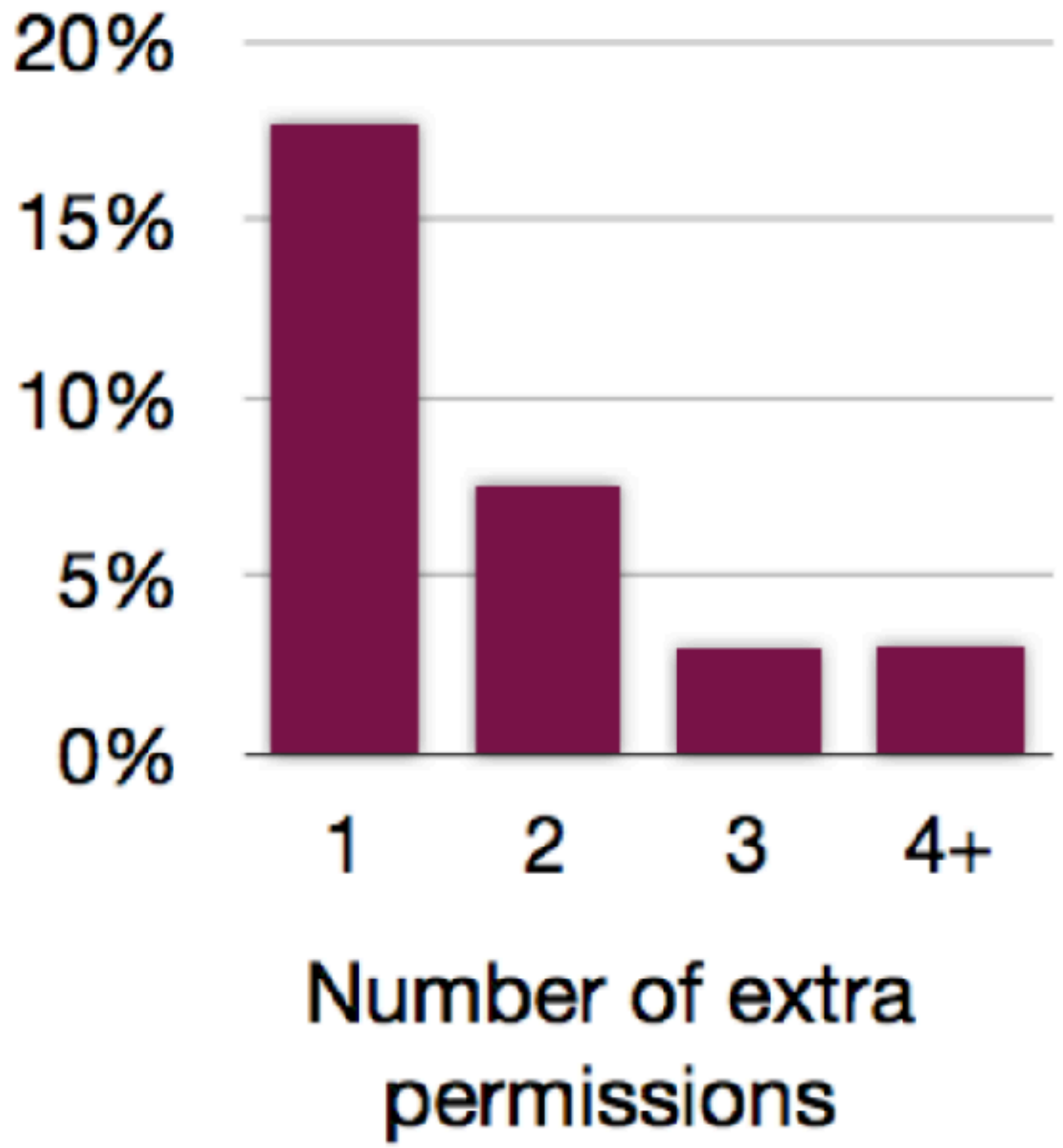
Do users act on permission information?

"Have you ever not installed an app because of permissions?"

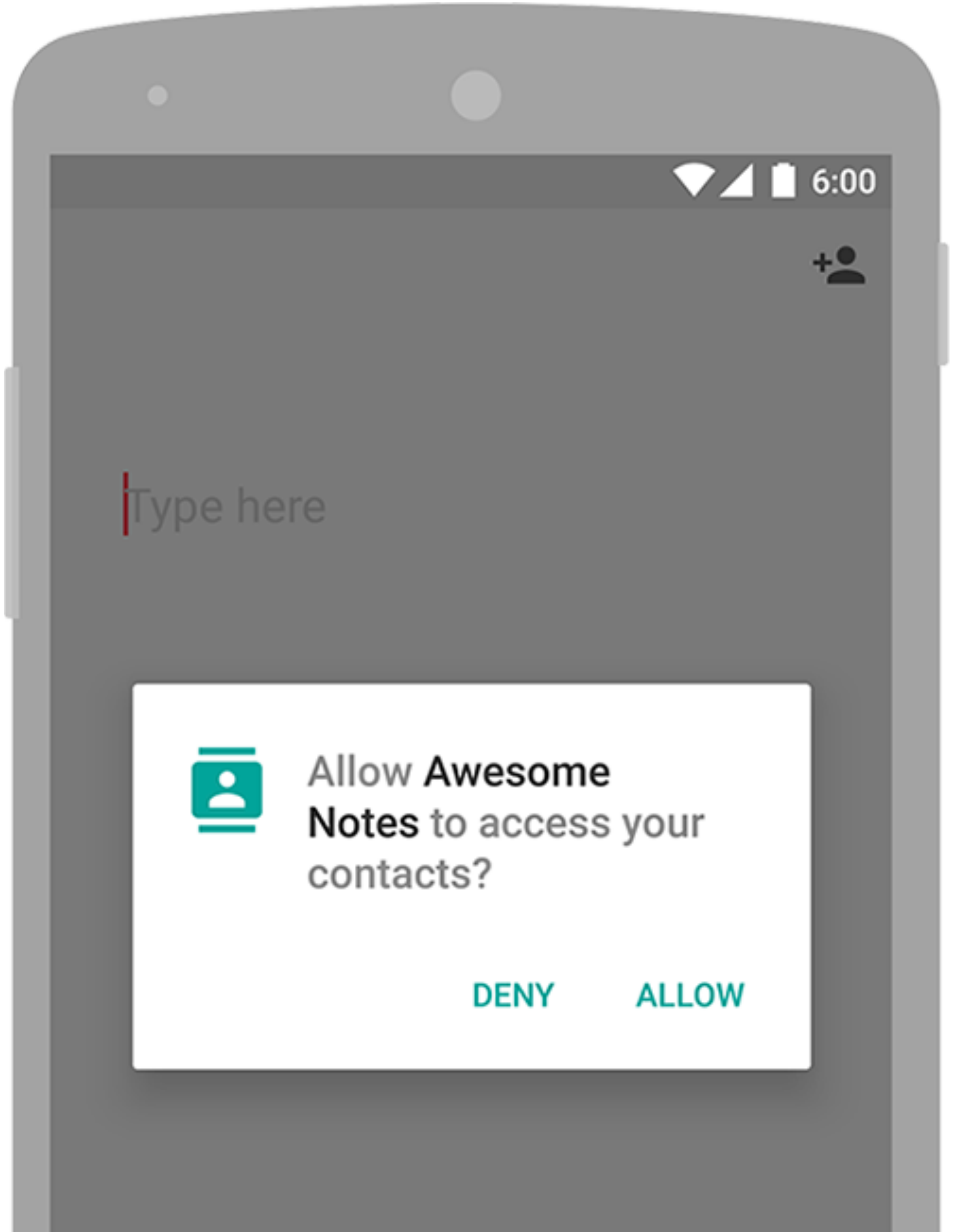


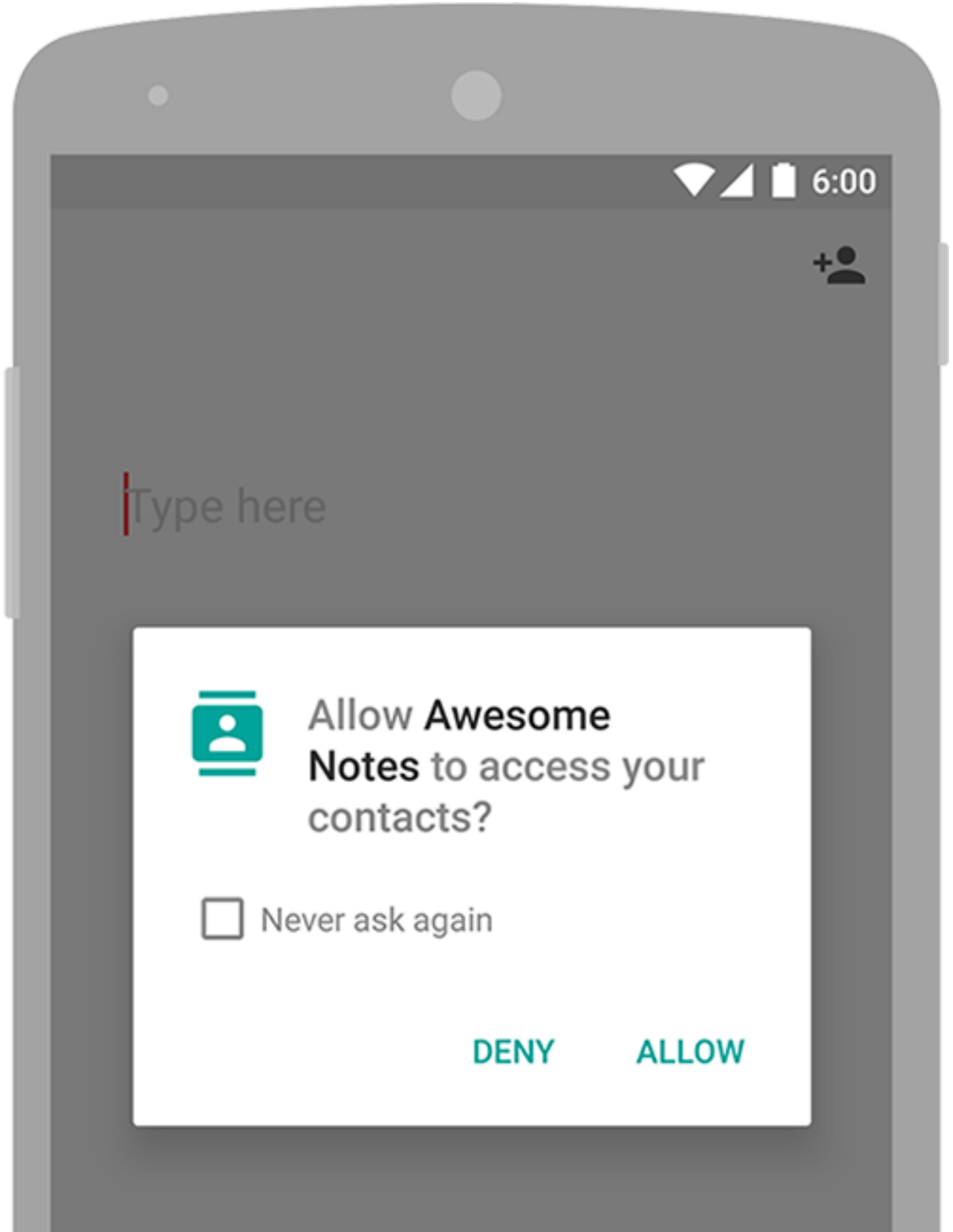
Developers Don’t know the Permissions They Need





Android Now Asks at Runtime (was not the case historically)





Manifests

In both cases, the Android app needs to request permission in its manifest—it's just up to the Android OS when it asks the user.

The OS might also just grant the right it doesn't seem dangerous

Manifest also defines what exported endpoints *other* apps can access. Whole class of malware that takes advantage of this of misconfiguration.