Case Study: iOS Security

6.1600 Course Staff

Fall 2026

In this chapter, we look at how a real, widely deployed platform puts together many of the ideas from the platform-security part of the course: secure boot, isolated hardware for handling secrets, encryption of data at rest, and careful management of keys. Our case study is the security architecture of Apple's iPhone, as described in Apple's platform-security guide.[1]

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

[1] Apple Inc. *Apple Platform Security*. `https://support.apple.com/guide/security/welcome/web`. 2019.

Threat model. A smartphone faces many threats: malicious apps, a malicious network, tampered-with hardware, and so on. In this chapter, we focus on one specific threat:

> Someone steals your iPhone and wants to extract data from it.

The design makes two assumptions about the stolen device:

- the device is protected with a passcode, and
- the device is *locked* at the time of the theft.

If the thief grabs the phone out of your hand while it is unlocked, there is not much the phone can do.

The next chapter looks at a different threat: malicious apps, and how a mobile operating system isolates apps from each other while still letting them share data in controlled ways. We will focus on Android there.

Why does Apple care? The design we will describe represents a large engineering effort, and it requires dedicated hardware that makes every phone more expensive. Why might Apple invest in it?

- Some customers must keep secrets: doctors with medical records, government employees, reporters protecting their sources, and so on.
- Ordinary customers are nervous about what a thief could do with, say, the password to their bank account.
- Strong security is a competitive advantage over other phones.
- It is the right thing to do.

What could a thief try? Here are some potential attacks on a stolen phone:

- **Guess the passcode.** Passcodes are short—often only four or six digits—so exhaustive search is a real threat.
- **Impersonate the user's fingerprint or face**, to fool the biometric sensors.
- **Read the hardware directly.** The attacker can take the phone apart and remove the flash storage chips, or read the contents of RAM while the phone is powered up.

- **Exploit a bug in the operating-system kernel.** Even when the phone is locked, its USB, WiFi, and cellular-radio interfaces may be active and processing attacker-supplied input.
- **Install a modified operating system** that omits the security checks—for example, one that allows unlimited passcode guesses. The attacker could try to redirect the phone's network traffic to an update server that the attacker controls, or take the phone apart and write a different OS image directly onto the flash chips.
- **Downgrade the operating system** to an old version that has a known bug.

The iOS design is reasonably successful against these attacks. It is much harder to extract data from a stolen iPhone than it was ten years ago—hard enough that law-enforcement agencies, including the FBI, have complained about it publicly. History suggests, though, that this is not the end of the story: researchers and companies that sell phone-unlocking tools continue to find weaknesses.

In 2016, the FBI asked a court to compel Apple to write and sign a modified version of iOS that would let the FBI guess the passcode of a seized iPhone 5c without limits or delays. This is exactly the "install a modified OS" attack above: thanks to secure boot, the FBI could not mount it without Apple's signing key. The FBI eventually dropped the request after a third party found another way into the phone.

1 *iOS Hardware Architecture*

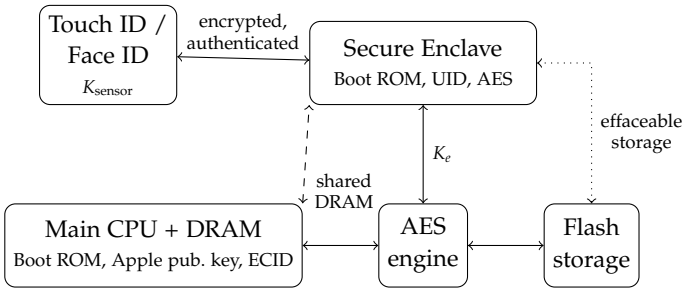


Figure 1: The main hardware components of an iPhone that are relevant to protecting data on a stolen device. The AES engine sits on the path between main memory and flash storage, so all data written to flash is encrypted.

Figure 1 shows the hardware components that matter for this chapter:

- **The main CPU and DRAM**, which run iOS and its apps.
- **Flash storage**, which holds the file system.
- **An AES encryption engine** that sits between main memory and flash storage. Every block that moves between them is encrypted or decrypted on the way, using direct memory access (DMA), without the data passing through the main CPU.
- **The Secure Enclave**, a separate processor that handles keys and passcodes.
- **A unique "UID" key** that is fused into the Secure Enclave's hardware when the chip is manufactured. The Secure Enclave can ask its hardware AES unit to encrypt or decrypt with the UID, but no software—not even the Secure Enclave's own software—can read the UID itself.

Not even Apple keeps a record of each device's UID.

- **Authentication sensors**: a fingerprint reader (Touch ID) or a face-recognition camera (Face ID). Each sensor shares a secret key, K_{sensor}, with the Secure Enclave.

The aspects of this design that we will look at are:

1. **Secure boot** (Section 2): making sure that the phone runs only Apple's software, and only recent versions of it.
2. **The Secure Enclave** (Section 3): hiding all keys and cryptographic operations from the main CPU, gating data decryption on the passcode, defending against exhaustive passcode search, and protecting communication between hardware units.
3. **Data protection** (Section 4): encrypting files in a way that still allows apps to run in the background while the phone is locked.

2 *Secure Boot*

The goal of secure boot is to make sure that an adversary cannot run a modified operating system, modified apps, or modified Secure Enclave software on the phone.

iOS uses the secure-boot scheme that we saw in the chapter on software trust. When the phone powers on, the CPU starts executing code from its *Boot ROM*. The Boot ROM is read-only: its contents are fixed when the chip is fabricated, and they include Apple's public signature-verification key. The boot sequence is then:

$$\text{Boot ROM} \rightarrow \text{iBoot} \rightarrow \text{OS kernel} \rightarrow \text{apps}$$

and each stage checks the next one before running it:

- The Boot ROM checks that the iBoot boot loader is signed by Apple.
- iBoot checks that the OS kernel is signed by Apple.
- The OS kernel checks that each app is signed by a developer that Apple has approved.

The Secure Enclave has its own Boot ROM and goes through a similar secure-boot sequence for its own software. Frazelle gives a broader discussion of hardware roots of trust and secure boot.[2]

[2] Jessie Frazelle. "Securing the Boot Process". In: *ACM Queue* 17.6 (2019).

2.1 *Downgrade attacks*

Signature checks alone do not stop an adversary from installing an *old* OS kernel. An old kernel is still signed by Apple! If that old kernel has a known bug, the adversary can install it on the stolen phone and then exploit the bug.

Strawman: record versions. One idea is for each boot stage to check version numbers: iBoot could refuse to run an OS kernel older than some minimum version. But where would the phone store the minimum version number? And what would prevent the adversary—who has physical access to the phone and can rewrite its flash chips—from modifying the stored version number?

Downgrade prevention with the ECID. Each iPhone chip has a unique identifier called the *ECID*. Like the Boot ROM, the ECID is read-only and set when the chip is fabricated. Unlike the UID, the ECID is not a secret.

When a phone installs an update, it sends its ECID to Apple's update servers. Apple's servers sign the update *for that specific ECID*: the signature covers both the software and the phone's ECID. During boot, each stage checks not only that the next stage is signed by Apple, but also that the signature is for this phone's ECID.

Why does this help?

- The adversary is unlikely to have a copy of the old software that was signed for the victim's ECID.
- Apple's update servers will not sign an out-of-date kernel for any ECID.

In effect, the "minimum version" is enforced by Apple's servers at the time of the update, rather than by state stored on the phone that the adversary might modify.

Apple calls this process *system software authorization*, and the signed updates are said to be *personalized* to the device. The phone also generates a fresh random nonce in its request to Apple's servers, so that a signature that was captured once cannot be replayed later. Members of the jailbreaking community used to save the personalized signatures for their own phones ("SHSH blobs"), so that they could later downgrade to a version of iOS with a known jailbreak. The anti-replay nonce makes this much harder. On many devices, though, the nonce is computed from a stored "generator" value, so an attacker who can run privileged code on the phone (for example, through a jailbreak or the `checkm8` Boot ROM bug) can restore the generator that was saved with a blob, which makes the old nonce and the old blob valid again.

3 *The Secure Enclave*

The Secure Enclave has two main jobs:

- prevent the main CPU from ever seeing cryptographic keys, and
- defend against passcode guessing.

The main CPU asks the Secure Enclave to do things on its behalf—for example, to help it decrypt a file—but never gets the keys themselves.

The Secure Enclave is implemented as a separate CPU with its own secure boot sequence. It shares DRAM with the main CPU and communicates with the main CPU through that shared memory. Since the main CPU (or an attacker probing the memory bus) can read DRAM, the Secure Enclave encrypts the contents of its own memory. It also authenticates its memory, to ensure integrity and freshness: the main CPU cannot modify the Secure Enclave's memory, or roll it back to an older value, without being detected. The state for this memory authentication lives in a small on-chip SRAM, which is not accessible to the main CPU.

Comparison to SGX. Intel's SGX (Software Guard Extensions) has a similar goal: hide secrets from a kernel that is not very trustworthy. It also encrypts and authenticates the memory it uses. The Secure Enclave differs from SGX in two ways:

- It is a *separate CPU*, rather than a special mode of the main CPU. This makes it much less exposed to side channels through shared micro-architectural state.
- It runs only fixed software from Apple, rather than arbitrary application code.

Intel's SGX was broken by the Foreshadow attack, which is an extension to Meltdown. Foreshadow was able to get Intel CPUs to sign anything!

3.1 User authentication

The overall plan is:

- all user data is encrypted;
- decryption is allowed when the phone is unlocked; and
- decryption is forbidden when the phone is locked.

Why offload authentication onto the Secure Enclave, rather than implementing it in the iOS kernel on the main CPU?

- To prevent a compromised OS kernel from obtaining decryption keys or the user's fingerprint and face data.
- To prevent a compromised OS kernel from bypassing the limits on passcode retries.
- Because the Secure Enclave can hide keys using memory encryption and authentication, so that directly reading the phone's RAM yields no cryptographic secrets. Encrypting and authenticating all memory would be too costly for the main CPU, but it is acceptable for the small amount of memory that the Secure Enclave uses.

The primary authentication mechanism is the passcode: the keys that decrypt the user's data are computed from the passcode, as we will see in Section 4. Touch ID and Face ID are additional, more convenient mechanisms. Once the user enters the passcode, the Secure Enclave caches the resulting decryption keys. Later, if the sensor presents a correct fingerprint or face match, the Secure Enclave uses these cached keys.

The phone requires the passcode, rather than a fingerprint or face, in several situations: for example, after the phone restarts, if it has not been unlocked for 48 hours, or after five failed biometric match attempts.

3.2 Protecting sensor-to-enclave communication

The fingerprint and face sensors do not talk to the Secure Enclave directly: the main CPU forwards their data. This leads to two possible attacks:

- A compromised OS on the main CPU records a legitimate fingerprint reading and later *replays* it to the Secure Enclave.

- The adversary replaces the phone's fingerprint sensor with a *fake sensor* that always reports a match.

The thief could do this on a phone that has been unlocked once since it was last powered on—which is almost always the case—before its battery runs out.

The defense is cryptographic authentication between each sensor and the Secure Enclave. The sensor has a secret AES key, K_{sensor}, and the Secure Enclave knows the same key; the two are paired when the phone is manufactured. The sensor encrypts and authenticates all of the data it sends with this key. To prevent replay of old readings, the protocol must also include a fresh nonce or session identifier. A compromised OS cannot forge or replay readings, and a replacement sensor does not know the key.

Apple's documentation says that the sensor and the Secure Enclave use the shared key to set up a fresh session key, and protect the session with AES-CCM authenticated encryption.

Aside: facial recognition. The Face ID camera is surprisingly sophisticated. It includes an infrared (IR) dot projector and an IR camera. The projector shines many IR dots—tens of thousands—in a random pattern onto the user's face, and the IR camera reads the pattern and reports it to the Secure Enclave. The shape of the dot pattern helps distinguish a three-dimensional face from a two-dimensional photo of the face.

4 Data Protection

By *data*, we mean the files stored in flash. The attack we worry about is an adversary who takes the phone apart and reads the data directly from the flash chips. The plan is to encrypt all data stored on the flash chips.

4.1 Deriving keys from the passcode

The data-encryption keys are derived from the passcode, rather than stored permanently somewhere on the device. Why?

- So that, after the phone reboots, the keys exist nowhere on the device until the user enters the passcode.
- So that the phone can *forget* the keys when it locks.

How should the phone derive keys from the passcode? There are two dangers:

- There are only 10^4 four-digit passcodes (or 10^6 six-digit ones), so it is easy to try them all.
- An attacker might extract the flash chips and try all of the passcodes on the attacker's own (fast) machines.

Simply hashing the passcode, or stretching it with a slow hash as we did for password storage, does not help much: an exhaustive search over 10^6 candidates is cheap even with a slow hash.

The iOS scheme. iOS ties the derivation to the device's UID, which only the Secure Enclave's hardware can use. The data-encryption key is, roughly,

$$K_{\text{passcode}} = \mathsf{Enc}\big(\text{UID}, \mathsf{Enc}(\text{UID}, \cdots \mathsf{Enc}(\text{UID}, \text{passcode}) \cdots)\big),$$

that is, the result of encrypting the passcode with the UID many times. Each property of this construction defends against one of the attacks:

- **It depends on the passcode**, so the key exists nowhere on the device after a reboot.
- **It is computed inside the Secure Enclave**, which limits the number of guesses—for example, by imposing increasing delays after incorrect guesses—regardless of what the main CPU does.
- **It is slow**, which limits the speed of exhaustive search even by code that can talk to the Secure Enclave directly.
- **It depends on the UID**, which cannot be extracted from the hardware, so the adversary cannot run the search outside of the device, on many fast machines in parallel.

Apple describes this as "tangling" the passcode with the UID, using a PBKDF2-style construction whose pseudorandom function is AES keyed with the UID. The iteration count is calibrated so that one attempt takes about 80 milliseconds.

4.2 *Challenge: background apps*

When the phone is locked, some apps must still be able to run and read or write files. For example, the mail app should download email attachments while the phone is in your pocket. But the phone forgets the decryption keys when it locks!

iOS addresses this challenge with *key wrapping*. To *wrap* a key k_2 under another key k_1 means to encrypt it:

$$\mathsf{Enc}(k_1, k_2).$$

Anyone who knows k_1 can recover k_2 by decrypting. Key wrapping lets a module that knows k_2 delegate access to k_2 to a module that knows k_1: the first module hands out the wrapped key, which is useless to anyone who does not know k_1. It also lets a system store many keys (one per file, say) while keeping only a few keys secret.

4.3 *The key hierarchy*

The file-system key. The file system's metadata (directories, inodes, and so on) is encrypted with a single key, K_{fs}. The Secure Enclave stores K_{fs} wrapped under the UID, $\mathsf{Enc}(\text{UID}, K_{\text{fs}})$. At boot time, the Secure Enclave unwraps K_{fs} and hands the OS kernel a copy wrapped under K_e, a key that the Secure Enclave shares with the AES engine: $\mathsf{Enc}(K_e, K_{\text{fs}})$. The kernel can then pass $\mathsf{Enc}(K_e, K_{\text{fs}})$ to the AES engine whenever it accesses metadata, without ever seeing K_{fs} itself.

Why bother with K_{fs} at all?

- It makes it possible to erase all data on the phone quickly, by deleting K_{fs}—or, really, by deleting $\mathsf{Enc}(\text{UID}, K_{\text{fs}})$.
- Since K_{fs} is wrapped under the UID, an adversary cannot move the flash chips into another phone and read them there.

Per-file keys. Each file's contents are encrypted with a different key, K_f. The file key is itself wrapped under a *data-protection key*, K_{dp}, and the wrapped key $\mathsf{Enc}(K_{\text{dp}}, K_f)$ is stored with the file's metadata (which in turn is encrypted under K_{fs}). The data-protection keys are stored only in the Secure Enclave. There are different data-protection keys for different levels of protection, and each file is assigned to one of these levels.

Apple calls the data-protection keys *class keys*, since each one corresponds to a data-protection class.

4.4 Levels of data protection

iOS offers four levels of data protection:

Complete protection. The file can be decrypted only while the phone is unlocked. The Secure Enclave derives K_{dp} for this level from the passcode when the user unlocks the phone, and discards it shortly after the phone locks.

Protected unless open. The file can be written at any time, but read only while the phone is unlocked. This is a special case designed for background downloads, such as email attachments that arrive while the phone is locked.

This level uses public-key cryptography: a file can be encrypted using a *public* data-protection key that is always available, but decrypting it later requires the matching secret key, which is derived from the passcode. Apple's implementation uses Elliptic-Curve Diffie-Hellman over Curve25519.

Protected until first user authentication. The file can be decrypted at any time, provided that the user has unlocked the phone at least once since it last booted. The Secure Enclave derives K_{dp} from the passcode on the first unlock and discards it only when the phone powers off. This is the default level for third-party app data.

No protection. The file can be decrypted at any time. K_{dp} for this level is derived only from the UID, not from the passcode, so it is available from boot. Even so, this level still protects the data from an adversary who moves the flash chips to another device, and it still allows a fast wipe and secure deletion (Section 4.6).

The background-apps challenge is thus solved by choosing the right level for each file: data that a background app must read while the phone is locked uses "protected until first user authentication," and data that it must write but not read uses "protected unless open."

4.5 Using wrapped file keys

The file-system code that runs on the main CPU sees only wrapped keys. When it needs to read or write a file, it asks the Secure Enclave

to unwrap the file's key, and the Secure Enclave does so if it currently holds the corresponding data-protection key.

But the Secure Enclave never gives the OS kernel the raw key! Instead, data is encrypted and decrypted by the hardware AES engine:

1. The kernel sends $\mathsf{Enc}(K_{\text{dp}}, K_f)$ to the Secure Enclave.
2. If it has K_{dp}, the Secure Enclave unwraps K_f and re-wraps it under K_e, the key it shares with the AES engine.
3. The Secure Enclave sends $\mathsf{Enc}(K_e, K_f)$ back to the kernel.
4. The kernel programs the AES engine with $\mathsf{Enc}(K_e, K_f)$ whenever it reads or writes blocks of the file. The AES engine unwraps K_f internally and encrypts or decrypts the data as it moves between memory and flash.

A compromised kernel can ask the AES engine to decrypt files while the phone is unlocked, but it never learns any file key, so it cannot decrypt the files after the phone locks or offline.

On recent devices, the key that the Secure Enclave shares with the AES engine is a fresh, ephemeral key generated at each boot.

Touch ID and Face ID. How does the phone obtain data-protection keys when the user unlocks it with a fingerprint or face rather than the passcode?

- Touch ID and Face ID cannot be used immediately after a reboot; the user must first enter the passcode.
- When the user enters the passcode, the Secure Enclave computes the data-protection keys, and sets aside (encrypted) copies of them for later use with Touch ID or Face ID.
- When the user locks the phone, the Secure Enclave deletes the working copies of the data-protection keys.
- When the sensor later reports a match, the Secure Enclave recovers the keys from the copies it set aside.

4.6 Secure data deletion

There are two goals:

- allow the user to quickly wipe all data on the phone, and
- (optionally) wipe the phone after 10 failed unlock attempts.

The challenge is that it is hard to delete data from flash storage. Overwriting all of the flash takes a long time. Worse, the flash controller makes copies of data as part of *wear-leveling*: to keep any one part of the chip from wearing out, it writes updates to fresh locations and leaves the old copies behind until it gets around to erasing them. Someone who reads the flash chips directly may be able to recover these old copies.

iOS solves this with *effaceable storage*: a small region of flash that the Secure Enclave accesses directly, using low-level flash operations

that really do erase the data. Effaceable storage holds a few small, critical values, such as $\mathsf{Enc}(\text{UID}, K_{\text{fs}})$. Since every other key is (directly or indirectly) wrapped under K_{fs} or a key derived from the UID, erasing these few values makes all of the data on the phone unrecoverable at once. This is called *crypto-shredding*: rather than deleting the data, we delete the key.

5 Discussion

Places to look for weaknesses. Where might an adversary find a way in?

- **The kernel on the main CPU** sees a lot of sensitive material: the passcode as the user types it, decrypted data, web passwords, and so on. The iOS kernel has had bugs that apps can exploit to gain root or kernel privileges on the main CPU. Luckily, an adversary cannot install new malicious apps on a locked phone. Apple also uses hardware features, such as ARM's pointer authentication codes, to make kernel exploits harder.[3]
- **Boot code** may have bugs that let an adversary boot a kernel that allows passcode guessing. This has happened to the Boot ROM of many iPhone models, to Apple's T2 security chip in Intel-based Macs,[4] and to Intel's own hardware root of trust.[5]
- **The USB, WiFi, and cellular-radio software** probably has bugs, and may be active even when the phone is locked.
- **Apple's private signing keys** must be kept safe, but they must also be used routinely, to sign "personalized" updates for each device. (Apple likely has several or many private keys.)
- **The UID** is supposed to be impossible to extract from the hardware. Can an adversary with enough money extract it anyway, for example by imaging the chip?

Is there a cost to this security?

- The specialized hardware—the Secure Enclave, the fingerprint reader, the AES DMA engine—costs money.
- Having to type a passcode is irritating, and the user needs the passcode in many situations, such as backing up the phone.
- Background activities are awkward while the device is locked.
- If the user forgets the passcode, the data is permanently lost (unless it was backed up).

6 Summary

The iOS design brings together several interesting ideas and techniques:

[3] Jonathan Corbet. *ARM Pointer Authentication*. `https://lwn.net/Articles/718888/`. 2017; Brandon Azad. *Examining Pointer Authentication on the iPhone XS*. `https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html`. 2019.

Pointer authentication attaches a short MAC to pointers (in particular, code pointers), so that a corrupted pointer is detected before it is used.

[4] Dan Goodin. *Apple's T2 Security Chip Has an Unfixable Flaw*. `https://arstechnica.com/information-technology/2020/10/apples-t2-security-chip-has-an-unfixable-flaw/`. 2020.

[5] Mark Ermolov. *Intel x86 Root of Trust: Loss of Trust*. `http://blog.ptsecurity.com/2020/03/intelx86-root-of-trust-loss-of-trust.html`. 2020.

The `checkm8` exploit, used by the `checkra1n` jailbreak, exploits a bug in the USB code in the iPhone Boot ROM. Since the Boot ROM is read-only, Apple cannot fix the bug on affected phones. The exploit must be rerun on every boot, though, and it does not by itself bypass the Secure Enclave's passcode protection.

- **Secure boot**: code signing, and downgrade prevention by personalizing each update to the device's ECID.
- **The Secure Enclave**: a separate processor that keeps all keys away from the main CPU, enforces passcode-guess limits, and uses encrypted and authenticated communication with the sensors.
- **Data protection**: per-file keys, key wrapping, a device key (the UID) that cannot leave the hardware, multiple protection levels, and effaceable storage for fast, secure deletion.

Further reading. Apple's platform-security guide[6] describes these mechanisms, and many others, in much more detail. Android phones face similar considerations; Mayrhofer describes how Android defends against insider attacks, such as an insider who can sign malicious firmware updates.[7]

[6] Apple Inc., *Apple Platform Security*.

[7] René Mayrhofer. *Insider Attack Resistance in the Android Ecosystem*. USENIX Enigma, `https://www.usenix.org/conference/enigma2019/presentation/mayrhofer`. 2019.

References

Apple Inc. *Apple Platform Security*. `https://support.apple.com/guide/security/welcome/web`. 2019.

Azad, Brandon. *Examining Pointer Authentication on the iPhone XS*. `https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html`. 2019.

Corbet, Jonathan. *ARM Pointer Authentication*. `https://lwn.net/Articles/718888/`. 2017.

Ermolov, Mark. *Intel x86 Root of Trust: Loss of Trust*. `http://blog.ptsecurity.com/2020/03/intelx86-root-of-trust-loss-of-trust.html`. 2020.

Frazelle, Jessie. "Securing the Boot Process". In: *ACM Queue* 17.6 (2019).

Goodin, Dan. *Apple's T2 Security Chip Has an Unfixable Flaw*. `https://arstechnica.com/information-technology/2020/10/apples-t2-security-chip-has-an-unfixable-flaw/`. 2020.

Mayrhofer, René. *Insider Attack Resistance in the Android Ecosystem*. USENIX Enigma, `https://www.usenix.org/conference/enigma2019/presentation/mayrhofer`. 2019.