Lecture #10

Applied Cryptography for Mobile and IoT Devices

SMA 2025

Title Slide

Applied Cryptography for Mobile and IoT Devices

Implementation, Performance, and Constraints

Today's Agenda

- **Part 1: Cryptographic Fundamentals** - Symmetric vs. Asymmetric, Block Modes, and Hashing.
- **Part 2: The Mobile Constraint** - Battery, CPU, and the need for Hardware Acceleration.
- **Part 3: Asymmetric Deep Dive** - RSA vs. ECC and the importance of Key Exchange.
- **Part 4: Securing Data-in-Transit** - TLS 1.3, Handshakes, and Certificate Pinning.
- **Part 5: Securing Data-at-Rest** - File-Based Encryption, Keystores, and Biometrics.
- **Part 6: IoT Security** - Bootstrapping, MQTT, and Secure Firmware Updates.

Recap from Lecture 9

- **Digital Forensics:** We explored the science of recovering evidence from mobile devices.
- **Acquisition Methods:** We learned about Logical, File System, and Physical acquisition.
- **Artifact Analysis:** We saw how call logs, location data, and timestamps build a timeline of events.
- **The Challenge of Encryption:** We discussed how modern encryption (FBE) makes forensic acquisition incredibly difficult.

Part 1: Cryptographic Fundamentals

The Building Blocks of Security

Category 1: Symmetric Encryption

- **Concept:** The *same* key is used for both encryption and decryption.
- **Analogy:** A physical door key. If you have the key, you can lock the door (encrypt) and unlock it (decrypt).
- **Speed:** Very fast. Designed for processing large amounts of data.
- **Primary Use:** Encrypting files, databases, and the bulk of network traffic.

Common Symmetric Algorithms

- **AES (Advanced Encryption Standard):**
  - The global standard. Trusted by governments and industry.
  - Block size: 128 bits.
  - Key sizes: 128, 192, or 256 bits.
  - A modern stream cipher.
  - Designed to be fast in *software*, making it ideal for mobile devices without dedicated AES hardware.
  - Often paired with Poly1305 for authentication.

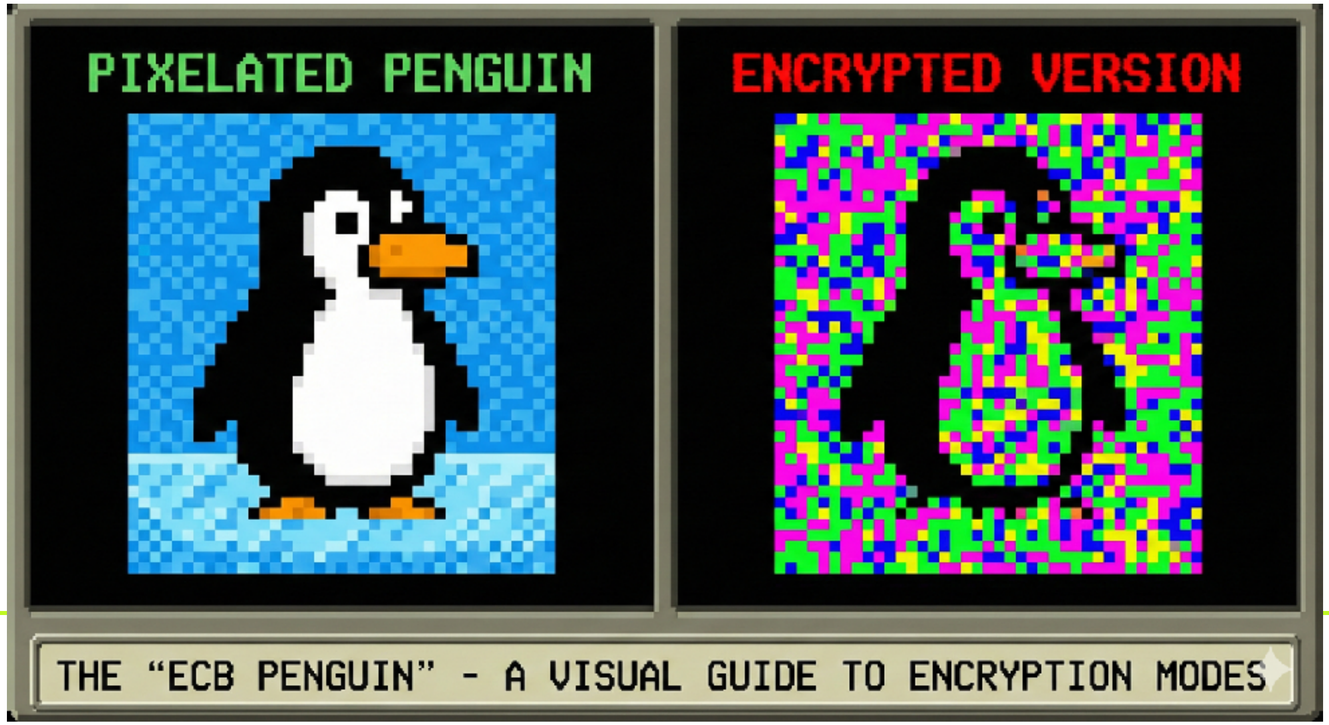
Block Cipher Modes of Operation

AES encrypts data in fixed-size blocks (128 bits). But most files are larger than 128 bits. How do we encrypt the whole file? We use a "Mode of Operation."

- **ECB (Electronic Codebook):** The naive approach. Encrypts each block independently. **INSECURE.**
- **CBC (Cipher Block Chaining):** Each block depends on the previous one. Better, but hard to implement correctly.
- **GCM (Galois/Counter Mode):** The modern standard. Provides both encryption *and* integrity (authentication).

Why ECB is Dangerous

- **The Problem:** In ECB mode, identical blocks of plaintext produce identical blocks of ciphertext.
- **The Result:** Patterns in the data are preserved. An attacker can "see" the structure of the data even if they can't read the exact bytes.



Authenticated Encryption (AEAD)

- **Confidentiality:** Keeping data secret (Encryption).
- **Integrity:** Ensuring data hasn't been tampered with (hashing/MAC).
- **AEAD (Authenticated Encryption with Associated Data):** Combines both into a single, efficient operation.
- **AES-GCM:** The most common AEAD mode. It ensures that if an attacker modifies even one bit of the encrypted file, the decryption will fail completely.

Category 2: Hash Functions

- **Concept:** A one-way mathematical function that takes any amount of data and produces a fixed-size string (the "digest" or "hash").
- **Characteristics:**
  - **Deterministic:** Same input always equals same output.
  - **One-Way:** You cannot reverse the hash to get the original data.
  - **Collision Resistant:** It should be impossible to find two different inputs that produce the same hash.

Common Hash Algorithms

- **MD5:** Broken. Do not use.
- **SHA-1:** Broken. Do not use.
- **SHA-2 (SHA-256):** The current industry standard. Secure and widely supported.
- **SHA-3:** The newest standard, built on a different mathematical structure.

Category 3: Asymmetric Cryptography

- **Concept:** Uses a *pair* of keys.
- **Public Key:** Shared with the world. Used to encrypt messages or verify signatures.
- **Private Key:** Kept secret. Used to decrypt messages or sign data.

Part 2: The Mobile Constraint

Why Crypto is Hard on Phones

The Resource Triad

- **Battery:**
  - Crypto = Math. Math = CPU cycles. CPU cycles = Power drain.
  - A poorly written crypto loop can drain a battery in hours.
- Mobile CPUs are powerful, but they throttle (slow down) when they get hot.
- Heavy crypto can cause the UI to stutter or freeze.
- IoT devices might have only kilobytes of RAM.
- Some older algorithms require large memory tables, which is bad for these devices.

Hardware Acceleration

To solve the performance problem, we use special hardware.

- **AES-NI / ARMv8 Crypto Extensions:**
  - Special CPU instructions designed just for encryption.
  - They run 10x-100x faster than software code.
  - They use significantly less battery.

Trusted Execution Environment (TEE)

- **What is it?** A secure area of the main processor. It runs a separate, secure operating system.
- **Android:** TrustZone.
- **iOS:** Secure Enclave.
- **Purpose:** To perform sensitive operations (like unlocking the device or processing payments) in an isolated environment that even the main OS (Android/iOS) cannot tamper with.

The Secure Enclave (Apple)

- A dedicated hardware coprocessor.
- **Key Feature:** It stores cryptographic keys that *never leave the hardware*.
- **How it works:**
  - The App asks the Enclave to create a key.
  - The Enclave creates it and stores it internally.
  - The App asks the Enclave to sign a piece of data.
  - The Enclave signs it and returns the signature.
  - The App *never* sees the private key.

Part 3: Asymmetric Deep Dive

RSA vs. ECC: The Battle for Mobile

RSA (Rivest–Shamir–Adleman)

- **The Old Guard:** Invented in 1977.
- **Math:** Based on the difficulty of factoring large prime numbers.
- **Key Size:** Requires very large keys to be secure (2048 or 3072 bits).
- **Performance:** Fast at verifying signatures, but slow at generating keys and signing.

ECC (Elliptic Curve Cryptography)

- **The Challenger:** Gained popularity in the 2000s.
- **Math:** Based on the algebraic structure of elliptic curves over finite fields.
- **Key Size:** Tiny! A 256-bit ECC key is as strong as a 3072-bit RSA key.
- **Performance:** Much faster at signing and key generation.

Why ECC Wins on Mobile

- **Smaller Keys:**
  - Less storage space needed.
  - Less bandwidth needed to send certificates over the network.
  - Less CPU usage = Longer battery life.
  - Faster handshakes = App loads faster.
  - IoT devices with tiny memory can handle ECC but often choke on RSA.

Digital Signatures

- **Goal:** To prove authenticity and integrity.
- **Process:**
  - Sender hashes the message (SHA-256).
  - Sender encrypts the hash with their **Private Key**. This is the "Signature".
  - Receiver decrypts the signature with the Sender's **Public Key**.
  - Receiver hashes the message themselves.
  - If the two hashes match, the signature is valid.

Key Exchange (Diffie-Hellman)

- **The Problem:** How do two strangers agree on a secret symmetric key over a public network without anyone else seeing it?
- **The Solution:** ECDH (Elliptic Curve Diffie-Hellman).
- **The Analogy:** Mixing paint.
  - Alice and Bob mix their secret colors with a public color.
  - They exchange the mixtures.
  - They add their secret color to the mixture they received.
  - They end up with the exact same final color (the shared key), but an eavesdropper can't figure it out.

Part 4: Securing Data-in-Transit

TLS and Network Security

What is TLS?

- **Transport Layer Security (TLS):** The successor to SSL.
- **Function:** It creates a secure, encrypted tunnel between the client (app) and the server.
- **Properties:**
  - **Confidentiality:** Nobody can read the data.
  - **Integrity:** Nobody can modify the data.
  - **Authentication:** You know you are talking to the real server.

The TLS Handshake (Simplified)

- **Client Hello:** "Hi, I speak TLS 1.3 and I support these algorithms."
- **Server Hello:** "Hi, let's use TLS 1.3 and AES-GCM. Here is my Certificate."
- **Key Exchange:** The client and server perform ECDH to generate a shared session key.
- **Finished:** "Let's switch to encryption."
- **Secure Data Transfer:** All subsequent data is encrypted with the shared key.

TLS 1.3 Improvements

- **Faster:** 1-RTT (Round Trip Time) handshake.
- **Safer:** Removed insecure algorithms (no more RSA key exchange, no more CBC mode).
- **0-RTT:** If you've visited the site before, you can send data immediately (Zero Round Trip).

The Chain of Trust

- **Certificate Authority (CA):** A trusted organization (like DigiCert or Let's Encrypt) that issues digital certificates.
- **Root Store:** Your phone comes pre-installed with a list of trusted CAs (the "Root Store").
- **Validation:**
  - Server sends its certificate.
  - Phone checks: "Is this signed by a CA in my Root Store?"
  - If yes, the connection is trusted.

The Weakness of the CA System

- **Too Many CAs:** There are hundreds of trusted CAs.
- **The Risk:** If *any* one of them is hacked or goes rogue, they can issue a fake certificate for *google.com* or *your-bank.com*.
- **The Result:** A Man-in-the-Middle (MitM) attacker can intercept your traffic, and your phone will think it's secure.

Certificate Pinning

- **The Solution:** Don't trust the whole Root Store. Trust only **one** specific certificate (or public key).
- **How it works:** You hardcode the hash of your server's public key inside your app.
- **The Check:** When the app connects, it checks if the server's key matches the hardcoded hash. If not, it kills the connection.

Implementing Pinning (Android)

Android makes this easy with XML configuration.

```
<network-security-config>
    <domain-config>
        <domain includeSubdomains="true">api.myapp.com</domain>
        <pin-set expiration="“2026-01-01">
            <pin digest="SHA-256">7HIpactkIAq2Y49orFOOQKurWxmmSFZhBCoQYcRhJ3Y=</pin>
            <pin digest="SHA-256">fwza0LRMXouZHRC8Ei+4PyuldWDURappCnOgQWO9s1L=</pin>
        </pin-set>
    </domain-config>
</network-security-config>
```

Implementing Pinning (iOS)

On iOS, you can use *Info.plist* or a library like TrustKit.

```
let trustKitConfig = [
    kTSKSwizzleNetworkDelegates: true,
    kTSKPinnedDomains: [
        "api.myapp.com": [
            kTSKPublicKeyHashes: [
                "7HIpactkIAq2Y49orFOOQKurWxmmSFZhBCoQYcRhJ3Y=",
                "fwza0LRMXouZHRC8Ei+4PyuldWDURappCnOgQWO9s1L="
            ],
            kTSKEnforcePinning: true
        ]
    ]
]
TrustKit.initSharedInstance(withConfiguration: trustKitConfig)
```

Part 5: Securing Data-at-Rest

Encryption on the Device

Evolution of Mobile Storage Security

- **No Encryption:** Early phones. If you lost it, anyone could read the data.
- **Full Disk Encryption (FDE):** One key for the whole drive.
- **Problem:** Phone is useless until unlocked after reboot.

How FBE (File-Based Encryption) Works

- **Credential Encrypted Storage:** Available only after the user unlocks the device. (e.g., User photos, emails).
- **Device Encrypted Storage:** Available immediately on boot. (e.g., Alarm clock settings, incoming call handlers).
- **Keys:** The keys for Credential Storage are derived from the user's passcode.

Storing Keys: The Keystore

Where do we store the keys that encrypt the data? We can't just put them in a text file.

- **Android Keystore System:** A system service that stores cryptographic keys in the TEE (Trusted Execution Environment).
- **iOS Keychain:** The Apple equivalent.
- **Benefit:** The key material is never exposed to the application process. You ask the Keystore to "sign this data" or "decrypt this key," and it does it inside the secure hardware.

Using the Android Keystore

```
KeyGenerator keyGenerator = KeyGenerator.getInstance(
        KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

keyGenerator.init(new KeyGenParameterSpec.Builder("MyKeyAlias",
        KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
        .build());

SecretKey key = keyGenerator.generateKey();
```

Using the iOS Keychain

```
let attributes: [String: Any] = [
    kSecClass as String: kSecClassGenericPassword,
    kSecAttrAccount as String: "MySecretKey",
    kSecValueData as String: secretData,
    kSecAttrAccessible as String: kSecAttrAccessibleWhenUnlockedThisDeviceOnly
]

SecItemAdd(attributes as CFDictionary, nil)
```

Biometric Authentication

We can bind keys to biometrics.

- **Concept:** The key in the Keystore is "locked" and can only be used if the user successfully authenticates with their fingerprint or face.
- **Flow:**
  - App requests to use the key.
  - Keystore prompts user for Biometric.
  - User touches sensor.
  - Hardware verifies match.
  - Keystore unlocks key and performs operation.

Secure Boot

How do we know the OS itself hasn't been hacked?

- **Chain of Trust:**
  - **Boot ROM:** Burned into the chip at the factory. Cannot be changed. Verifies the Bootloader.
  - **Bootloader:** Verifies the Kernel.
  - **Kernel:** Verifies the OS System Partition.

Part 6: IoT Security

The Wild West of Cryptography

The "Headless" Problem

- **Scenario:** You buy a smart lightbulb. It has no screen, no keyboard, and no internet connection.
- **Goal:** Connect it to your home Wi-Fi.
- **Challenge:** How do you tell it the Wi-Fi password securely?

Bootstrapping Method 1: SoftAP

- **How it works:** The device creates its own Wi-Fi network (e.g., "Bulb-Setup"). You connect your phone to it.
- **The Flaw:** This connection is often unencrypted HTTP.
- **The Attack:** An attacker nearby can sniff the traffic and steal your home Wi-Fi password as you send it to the bulb.

Bootstrapping Method 2: BLE (Bluetooth Low Energy)

- **How it works:** The phone connects to the device via BLE.
- **The Advantage:** BLE supports pairing and encryption protocols.
- **The Flow:**
  - Phone pairs with Bulb via BLE.
  - Phone sends Wi-Fi credentials over encrypted BLE channel.
  - Bulb connects to Wi-Fi.

IoT Protocols: MQTT

- **HTTP:** Good for web pages, heavy for IoT.
- **MQTT (Message Queuing Telemetry Transport):**
  - Lightweight publish/subscribe protocol.
  - Perfect for unreliable networks and low-power devices.

IoT Authentication: Mutual TLS (mTLS)

- **Standard TLS:** Client verifies Server.
- **Mutual TLS:** Client verifies Server AND Server verifies Client.
- **IoT Use Case:**
  - Every lightbulb has a unique client certificate burned in at the factory.
  - The cloud server refuses to talk to any device that doesn't present a valid certificate signed by the manufacturer.

Firmware Updates (OTA)

- **The Risk:** If an attacker can push a malicious firmware update, they own the device forever.
- **The Defense:** Digital Signatures.
- Manufacturer signs the firmware update with their **Private Key**.
- Device has the Manufacturer's **Public Key** stored in Read-Only Memory.
- Device verifies the signature before installing the update.

The Mirai Botnet

- **What happened:** In 2016, millions of IoT devices (cameras, routers) were hacked.
- **The Vulnerability:** Default passwords (admin/admin) and open telnet ports.
- **The Impact:** The botnet launched massive DDoS attacks, taking down major parts of the internet.
- **Lesson:** IoT security affects everyone.

Case Study: Smart Lock Security

Let's design a secure smart lock system.

- **Components:** Lock (BLE), Gateway (Wi-Fi), Cloud, Mobile App.
- **Threat Model:**
  - Attacker sniffing Bluetooth.
  - Attacker hacking the Cloud.
  - Attacker stealing the Phone.

Smart Lock: BLE Security

- **Challenge:** Replay Attacks. An attacker records the "unlock" signal and plays it back later.
- **Solution:** Challenge-Response or Rolling Codes.
- Lock sends a random number (Nonce).
- Phone signs the Nonce with a shared key.
- Lock verifies signature.
- The "unlock" command is unique every time.

Smart Lock: Guest Access

- **Scenario:** You want to let a guest in for one hour.
- **Mechanism:** Ephemeral Keys.
- App generates a temporary key valid for 1 hour.
- App sends key to Guest's phone.
- Guest uses key to unlock.
- Lock enforces the time limit.

Performance Tuning for IoT

- **Problem:** RSA-2048 is too slow for a cheap microcontroller.
- **Solution:** Use Curve25519 (ECC).
  - Extremely fast.
  - Small code size.
  - High security.

Random Number Generation (RNG)

- **The Hidden Danger:** Cryptography relies on good randomness.
- **IoT Issue:** Cheap devices often lack a good source of entropy (randomness).
- **Result:** Predictable keys. If the RNG is predictable, the crypto is broken.
- **Fix:** Use hardware True Random Number Generators (TRNG) built into modern microcontrollers.

Side-Channel Attacks

- **Concept:** Attacking the implementation, not the math.
- **Power Analysis:** Monitoring the power consumption of the device to guess the key bits.
- **Timing Attacks:** Measuring how long an operation takes to guess the key.
- **Defense:** Constant-time algorithms (code that takes the exact same time regardless of the input).

Quantum Computing Threat

- **The Future:** Quantum computers could theoretically break RSA and ECC.
- **Timeline:** Unclear (10-30 years?).
- **Post-Quantum Cryptography (PQC):** New algorithms (Lattice-based) that are resistant to quantum attacks.
- **Relevance:** "Store Now, Decrypt Later" attacks. Encrypted data stolen today could be decrypted in 20 years.

Best Practices Checklist (1/3)

- **Don't Roll Your Own Crypto:** Use standard libraries (Bouncy Castle, Sodium, OS APIs).
- **Use Hardware Acceleration:** AES-NI, ARM Crypto Extensions.
- **Secure Key Storage:** Use the TEE / Secure Enclave. Never store keys in code or shared preferences.

Best Practices Checklist (2/3)

- **Use TLS 1.3:** Enforce it on your servers.
- **Pin Certificates:** For high-security apps.
- **Encrypt Data at Rest:** Use FBE / EncryptedSharedPreferences.

Best Practices Checklist (3/3)

- **Authenticate Everything:** mTLS for IoT, OAuth for Apps.
- **Sign Firmware:** Secure Boot and Secure OTA.
- **Plan for Updates:** Crypto agility. Be able to swap algorithms if one breaks.

Tools for Testing

- **Wireshark:** Analyze network traffic.
- **mitmproxy:** Intercept and inspect TLS traffic (useful for debugging).
- **MobSF (Mobile Security Framework):** Static analysis of Android/iOS apps to find crypto issues.
- **OpenSSL:** Command line tool for generating keys and testing connections.

Part 7: Practical Code Examples

Implementing Crypto on Android and iOS

Android: EncryptedSharedPreferences (Kotlin)

```kotlin
// The Master Key (stored in Keystore)
val masterKey = MasterKey.Builder(context)
    .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
    .build()

// The Encrypted Shared Preferences
val sharedPreferences = EncryptedSharedPreferences.create(
    context,
    "secret_shared_prefs",
    masterKey,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)

// Usage is identical to standard SharedPreferences
sharedPreferences.edit().putString("auth_token", "xyz123").apply()
```

iOS: Signing with Secure Enclave (Swift)

```swift
import CryptoKit

// 1. Generate a Private Key in the Secure Enclave
let privateKey = try SecureEnclave.P256.Signing.PrivateKey()

// 2. The Public Key (safe to send to server)
let publicKeyData = privateKey.publicKey.compactRepresentation!

// 3. Sign some data
let dataToSign = "Unlock Door".data(using: .utf8)!
let signature = try privateKey.signature(for: dataToSign)

print("Signature: \(signature)")
```

Android: Hashing Data (Kotlin)

```kotlin
import java.security.MessageDigest

fun hashString(input: String): String {
    val bytes = input.toByteArray()
    val md = MessageDigest.getInstance("SHA-256")
    val digest = md.digest(bytes)

    // Convert byte array to hex string
    return digest.fold("") { str, it -> str + "%02x".format(it) }
}

val hash = hashString("password123")
```

Lecture Wrap-Up

- **Applied Crypto is Engineering:** It's about constraints, trade-offs, and implementation details.
- **Mobile is Special:** Battery and UI responsiveness drive our choices (ECC > RSA).
- **IoT is Dangerous:** The physical world brings new threats (Side-channels, bad RNG).
- **The Goal:** To build systems that are secure by design and resilient to attack.

Next Lecture Preview

Lecture 11: Network Security and Traffic Analysis

- We will take the tools we learned today (Wireshark, TLS) and use them to analyze real-world traffic.
- We will learn how to spot malicious traffic patterns.
- We will discuss VPNs, Proxies, and Tor.

Additional Resources

- **Book:** "Serious Cryptography" by Jean-Philippe Aumasson.
- **Standard:** NIST Digital Identity Guidelines.
- **Docs:** Android Keystore System Training.
- **Docs:** Apple CryptoKit Documentation.

Q&A

Questions?