UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Protecting Secrets on ESP32: Encryption with PUF-Derived Keys

Marco Oliani

marco.oliani@univr.it

Anno Accademico 2024/25

Learning Objectives

- **Objective**: Demonstrate secure storage and encryption on ESP32 using Arduino IDE.
- **Focus**: Encrypt/decrypt a secret in NVS with AES-CBC, using a PUF-derived key.
- **Why it matters**: IoT devices like ESP32 face resource constraints and security threats.
- **Lab Goals**:
  - Understand symmetric cryptography (AES-CBC).
  - Implement key derivation from PUF.
  - Optimize for ESP32's limited resources (SRAM, CPU).

Why Cryptography in IoT Devices?

- **Motivation**:
  - IoT and embedded systems often handle **sensitive data**: credentials, session keys, tokens, certificates.
  - Devices are frequently **physically accessible** to untrusted users.
  - By default, firmware and memory are **not protected** from direct access or extraction.
  - Common attacks include:
    - **Physical attacks**: Direct access to flash or RAM, Device cloning.
    - **Logical attacks**: Firmware tampering, eavesdropping. reverse engineering.
- **Implications**:
  - Confidentiality cannot be assumed without explicit protection mechanisms.
  - Security solutions must work within the resource constraints of embedded platforms.

ESP32 Architecture and Resource Constraints

- **General characteristics (ESP32 family)**:
  - **SoC by Espressif** designed for wireless-connected embedded systems.
  - **CPU**: Xtensa LX6/LX7 (ESP32, ESP32-S3) or RISC-V (ESP32-C3, ESP32-C6).
  - Single or dual-core, up to 240+ MHz.
  - **On-chip SRAM**: typically between 320 KB and 512 KB, shared across code and data.
  - **Flash memory**: typically 2 MB to 16 MB (introduces higher latency).
  - Optional **external PSRAM**: 2–8 MB on selected modules (e.g., ESP32-WROVER, ESP32-S3).

ESP32 Architecture and Resource Constraints (1)

- **Hardware Constraints**:
  - **Limited on-chip memory**: Small SRAM (e.g., 520 KB on ESP32) demands efficient buffer and heap management for cryptographic operations (e.g., AES, IVs, padding), especially in resource-intensive protocols like TLS.
  - **Limited secure hardware**: No dedicated secure enclave or TPM (*Trusted Platform Module*), though eFuse provides hardware-backed key storage in ESP32.
  - **Physical accessibility**: Embedded systems are often exposed to attackers with direct hardware access (e.g., via SPI or JTAG).
  - **No default memory isolation**: Flash and RAM (including PSRAM) are readable unless protected by hardware (e.g., Flash Encryption) or software mechanisms (e.g., using mbedtls with AES-128-CBC).

ESP32 Architecture and Resource Constraints (2)

- **Software Constraints**:
  - **Limited OS security**: Embedded operating systems (e.g., FreeRTOS in ESP-IDF or Arduino) offer minimal security primitives, requiring custom implementations.
  - **No default encryption**: Data in flash or PSRAM is unencrypted unless explicitly protected (e.g., via Flash Encryption or software-based encryption).
  - **NVS limitations**: Non-Volatile Storage (NVS) supports persistent key-value storage but lacks built-in confidentiality without encryption.
  - **Application-level protection**: Security mechanisms (e.g., encryption, authentication) must be explicitly implemented in the application code.

ESP32 Architecture and Resource Constraints (3)

- **Design Implications**:
  - **Security-driven architecture**: Embedded security relies on robust software design and key management strategies.
  - **Lightweight security features**: Critical protections must be efficient, low-resource, and resilient to physical and logical attacks.
  - **Key derivation strategies**: Using derived, non-stored keys (e.g., via PUFs or eFuse-based seeds) reduces attack surface but increases complexity.

Introduction to Cryptography on ESP32

Introduction to Cryptography on ESP32

- **Goal**: Protect sensitive data (e.g., credentials, keys, communications) on ESP32.
- **Main Approach**: Software-based cryptography with **mbedtls** (AES, RSA, Elliptic-Curve Crypography).
- **Hardware Support**: Accelerators for AES, SHA-2, RSA, ECC available but not always used.
- **Security Features**:
  - Flash Encryption (AES-256) for flash memory.
  - eFuse for secure key storage.
  - True Random Number Generator (TRNG).
- **Challenge**: Balance security with limited resources in IoT applications.

Resource Consumption in Cryptography

- **Impact of Cryptography** (via mbedtls):
  - **CPU**: Complex operations (e.g., RSA, ECC) consume significant cycles.
  - **Memory**: Buffers for keys, IVs, padding reduce available SRAM.
  - **Energy**: Heavy algorithms increase power usage (critical for battery-powered IoT).
- **With Hardware Acceleration**:
  - AES, RSA, ECC faster with accelerators (e.g., AES-128: ~10 μs vs. ~100 μs in software).
  - Reduces CPU/energy usage but requires specific configuration.
- **Optimizations**:
  - Efficient mbedtls configuration (e.g., minimize dynamic allocations).
  - Use PSRAM for large data, with software-based encryption (e.g., AES-128-CBC).

MbedTLS Library

- Lightweight, **open-source cryptographic library** for embedded systems, supporting AES, RSA, Elliptic-Curve Criptography (ECC), and more.
- **Purpose**: Provides secure encryption and authentication (symmetric and public-key) on ESP32.
- **Supported Algorithms**:
  - **Symmetric**: AES for data encryption.
  - **Public-key**: RSA and ECC for authentication and key exchange.
- **Integration with Arduino IDE**:
  - Compatible with ESP32 boards (e.g., ESP32 Dev Module).
  - Add via manual import or compatible repository.
  - Works with FreeRTOS for IoT task management.
- Note: Balances security with ESP32's limited resources; requires careful configuration.

Symmetric Cryptography (AES with mbedtls)

- **Description**: Uses a shared key for encryption/decryption.
- **Characteristics**:
  - **Algorithm**: AES-128/256 via mbedtls (e.g., mbedtls_aes_crypt_cbc).
  - **Fast**: Ideal for large data volumes (e.g., TLS, PSRAM data).
  - **Trade-off**: Requires secure key distribution.
- Resource Consumption (software):
  - **CPU**: Moderate (e.g., ∼100 µs for 16 bytes with AES-128 on ESP32 at 240 MHz).
  - **Memory**: Small buffers (e.g., 16 bytes for IV, padding).
  - **Energy**: Acceptable consumption, suitable for IoT.
- **Note**: AES hardware accelerator reduces CPU to ∼10 µs, if enabled.
- **Example**: Encrypting PSRAM data with mbedtls AES-128-CBC.

Public Key Cryptography (RSA with mbedtls)

- **Description**: Uses public/private keys for authentication or key exchange.
- **Characteristics**:
  - **Algorithm**: RSA-2048/4096 via mbedtls (e.g., mbedtls_rsa_pkcs1_sign).
  - **Slow**: Complex mathematical operations (e.g., modular exponentiation).
  - **Secure**: Ideal for authentication (e.g., digital signatures) and TLS key exchange.
- **Resource Consumption** (software):
  - **CPU**: High (e.g., ∼1-2 s for RSA-2048 signature on ESP32 at 240 MHz).
  - **Memory**: Large buffers (e.g., 256-512 bytes for keys).
  - **Energy**: High consumption, less suitable for battery-powered IoT.
- **Note**: RSA hardware accelerator reduces time (e.g., ∼500 ms), if enabled.
- **Example**: Digital signature for authentication with mbedtls.

Public Key Cryptography (ECC with mbedtls)

- **Description**: Uses elliptic curves for smaller keys than RSA.
- **Characteristics**:
  - **Algorithm**: ECC (e.g., ECDSA, ECDH) via mbedtls (e.g., mbedtls_ecdsa_sign).
  - **Faster than RSA**: More efficient operations (e.g., NIST P-256 curve).
  - **Secure**: Same cryptographic strength with shorter keys (e.g., 256 bits vs. 2048 bits RSA).
- Resource **Consumption** (software):
  - **CPU**: Moderate (e.g., ∼100-200 ms for ECDSA P-256 signature on ESP32).
  - **Memory**: Smaller buffers (e.g., 32-64 bytes for keys).
  - **Energy**: More efficient than RSA, suitable for IoT.
- **Note**: ECC hardware accelerator reduces time (e.g., ∼50 ms), if enabled.
- **Example**: TLS authentication with **ECDSA** via mbedtls, **Blockchain**.

Memory Optimization in Cryptography on ESP32 (1)

- **Efficient Use of mbedTLS**:
  - Select lightweight algorithms: Prefer AES-128 over AES-256, ECC over RSA for lower memory usage.
  - Minimize buffer allocations: Reuse buffers for keys, IVs, and temporary data.
- **Memory Management**:
  - Limit dynamic memory: Avoid malloc/free in mbedtls; use static arrays when possible.
  - Leverage PSRAM for large data: Encrypt/decrypt in chunks to reduce SRAM usage.

Memory Optimization in Cryptography on ESP32 (2)

- **Code Optimization**:
  - Process data in small blocks: Use streaming APIs (e.g., mbedtls_aes_crypt_cbc) for large datasets.
  - Reduce stack usage: Keep function calls lean to avoid stack overflow on limited SRAM.
- **Practical Tips for Exercises**:
  - Test memory usage: Monitor SRAM with Arduino IDE's serial output or profiling tools.
  - Prioritize ECC for authentication: Smaller key sizes (e.g., 32-64 bytes for NIST P-256).
- **Note**: Careful memory optimization ensures secure cryptography within ESP32's resource constraints.

ESP32 Cryptography Lab

Lab Overview

- **Task**: Store a secret in NVS, encrypt/decrypt it using AES-CBC with mbedtls.
- **Key Components**:
  - ESP32 board (e.g., ESP32 Dev Module).
  - Arduino IDE with mbedtls library.
  - NVS for persistent storage, PUF for key derivation.
- **Challenges**:
  - Limited SRAM (520 KB) and CPU (240 MHz).
  - Secure key management without dedicated hardware.
- Outcome: Secure, efficient IoT application.

Understanding AES-CBC

- **What is AES-CBC**: Symmetric encryption with Advanced Encryption Standard in **Cipher Block Chaining** mode.
- Key **Features**:
  - Uses a single key (e.g., 128-bit) for encryption/decryption.
  - IV (Initialization Vector) ensures unique ciphertexts.
  - Block size: 16 bytes, requires padding for non-aligned data.
- **Why use it**: Fast, suitable for ESP32's limited resources.

Non-Volatile Storage (NVS) and Partitions on ESP32

- **What is NVS**: Key-value storage system in ESP32's flash for persistent data.
- **Key Features**:
  - Stores data (e.g., encrypted secrets) in a dedicated flash partition.
  - Accessed via Arduino's Preferences.h library in Arduino IDE.
  - Supports strings, numbers, and binary data (e.g., AES-encrypted secrets).
- **Partitions Overview**:
  - ESP32 flash is divided into partitions (e.g., app, NVS, data).
  - NVS partition: Reserved for key-value pairs, typically 96 KB.
  - Configurable via partition table in Arduino IDE or ESP-IDF.

Non-Volatile Storage (NVS) and Partitions on ESP32

- **Use in Lab**:
  - Store encrypted secrets (e.g., sensor data).
  - Retrieve and decrypt for secure IoT applications.
- **Resource Considerations**:
  - Minimal SRAM usage (1-2 KB for NVS operations).
  - Flash wear: Limit frequent writes to extend lifespan.
- **Note**: No built-in encryption; use mbedtls for confidentiality.

Physically Unclonable Functions (PUFs)

- **What is a PUF**: Hardware-based unique identifier leveraging chip variations.
- **Role in Key Derivation**:
  - Provides unique output (e.g., ESP32 chip ID or SRAM pattern).
  - Output processed with **PBKDF2-HMAC** to generate seed.
  - Seed used to derive AES key (via SHA-256).
- **Benefits**:
  - Enables device-specific keys, **reducing attack surface**.
  - Prevents cloning by tying keys to hardware.
- Resource **Impact**: Minimal SRAM usage (32 bytes for PUF output/seed).

Key Derivation Process

- **Steps**:
  - Obtain PUF output.
  - Generate seed using PBKDF2-HMAC.
  - Derive AES key from seed using SHA-256.
- **Key Management**:
  - Key not stored: Regenerated each time from PUF for security.
  - Avoids permanent storage in flash or NVS.
- **Why Derive Keys**:
  - Enhances security with dynamic, device-specific keys.
  - Reduces attack surface by avoiding hardcoded keys.
- **Resource Impact**: Minimal SRAM usage.

Lab Setup

- **Hardware**: ESP32 Dev Module (or similar).
- **Software**: Arduino IDE (latest version).
- **Libraries**: Preferences.h (NVS), mbedtls (manual import).
- **Prerequisites**:
  - Install ESP32 board support in Arduino IDE.
  - Configure mbedtls for AES-CBC and SHA-256.

Code Overview

- **Components**:
  - Initialize NVS to store/retrieve secrets.
  - Generate PUF-based seed and derive AES key.
  - Encrypt/decrypt secret with AES-CBC using mbedtls.
- **Structure**:
  - Setup: Initialize NVS, generate key, encrypt secret.
  - Loop: Decrypt and verify secret (optional).
- **Optimization**: Use static buffers, minimize dynamic memory.

Sample Code – NVS Initialization

```
#include <Preferences.h>
Preferences nvs;

void setup() {
    uint8_t *data; // Assume it'd initialized with bytes
    size_t len;

    nvs.begin("my-app", false); // Initialize NVS namespace
    // Store or retrieve secret
    nvs.putBytes("secret", data, len);
    nvs.end();
}
```

Sample Code - NVS Read/Write (1)

- Read Bytes:

```

#include <Preferences.h>
Preferences prefs; // Manages NVS

// Create a NVS namespace called mynamespace
prefs.begin(nvs_namespace, false);
size_t len = prefs.getBytesLength(nvs_key);

if (len > 0) {
  uint8_t buffer[len];
  prefs.getBytes(nvs_key, buffer, len);
}

prefs.end();

```

Sample Code - NVS Read/Write (2)

- Read Strings:

```

#include <Preferences.h>
Preferences prefs; // Manages NVS

// Create a NVS namespace called mynamespace
prefs.begin(nvs_namespace, false);
String nvs_string = prefs.getString(nvs_key);

prefs.end();

```

Sample Code - NVS Read/Write (3)

- Write Bytes:

```
#include <Preferences.h>
Preferences prefs; // Manages NVS

// Create a NVS namespace called mynamespace
prefs.begin(nvs_namespace, false);

prefs.putBytes(nvs_key, (uint8_t *)item, size);
prefs.end();
```

Sample Code - NVS Read/Write (4)

- Write Strings:

```

#include <Preferences.h>
Preferences prefs; // Manages NVS

// Create a NVS namespace called mynamespace
prefs.begin(nvs_namespace, false);

prefs.putString(string);
prefs.end();

```

Sample Code - Seed generation from PUF

```
#include <mbedtls/pkcs5.h>
#include <mbedtls/sha256.h>

void setup() {
  String puf = "YOUR_PUF_HERE";
  uint8_t seed[32]; // 32-byte seed
  const unsigned char *salt = (const unsigned char *)"my-salt";
  const uint32_t iterations = 2048;

  // Generate seed with PBKDF2-HMAC-SHA256
  mbedtls_pkcs5_pbkdf2_hmac_ext(
    MBEDTLS_MD_SHA256,
    (const unsigned char *)puf.c_str(), puf.length(),
    salt, strlen(salt),
    iterations,
    sizeof(seed), seed);
}

```

Sample Code - AES Key derivation

```
#include <mbedtls/sha256.h>

void setup() {
  uint8_t seed[16] = { /* Pre-filled seed from PBKDF2 */ }; // Example seed
  uint8_t key[32]; // 32-byte key (SHA-256 output)

  // Derive key with SHA-256
  mbedtls_sha256(seed, 16, key, 0); // Simple SHA-256 hash

  // Print key in hex
  Serial.print("Key: ");
  for (size_t i = 0; i < 32; i++) {
    if (key[i] < 16) Serial.print("0");
    Serial.print(key[i], HEX);
  }
  Serial.println();
}
```

Sample Code - Set AES Key

```
#include <mbedtls/aes.h>

void setup() {
    uint8_t seed[32];
    uint8_t derived_aes_key[32];
    mbedtls_aes_context ctx;
    mbedtls_aes_init(&ctx);
    // CREATE AND DERIVE KEY HERE

    // keybit is the number of BITS of the key
    mbedtls_aes_setkey_enc(&ctx, derived_aes_key, keybit);
    //mbedtls_aes_setkey_dec(&ctx, derived_aes_key, keybit);

    mbedtls_aes_free(&ctx);
}
```

Sample Code – AES-CBC Encryption (1)

```c
#include <mbedtls/aes.h>

void setup() {
    // Padding is not necessary if mbedtls is used.
    // MbedTLS pads input data automatically.
    size_t padded_len = (len + 15) / 16 * 16;
    uint8_t plaintext[padded_len];
    memset(plaintext, 0, padded_len);
    memcpy(plaintext, buffer, len);

    uint8_t aes_key[32];
    mbedtls_aes_context ctx_aes;
    mbedtls_aes_init(&ctx_aes);

    // SET_ENCRYPTION_KEY_HERE

```

Sample Code – AES-CBC Encryption (2)

```
        uint8_t encrypted[padded_len];
        uint8_t iv_copy[16];
        memcpy(iv_copy, iv, sizeof(iv));

        mbedtls_aes_crypt_cbc(
                &ctx_aes,
                MBEDTLS_AES_ENCRYPT,
                sizeof(plaintext),
                iv_copy, plaintext,
                encrypted
        );

        mbedtls_aes_free(&ctx_aes);
    }

```

Sample Code – AES-CBC Encryption (1)

```
#include <mbedtls/aes.h>

void setup() {
    // Retrieve secret from NVS first. You can save it
    // to enc_secret[];
    uint8_t aes_key[32];
    mbedtls_aes_context ctx_aes;
    mbedtls_aes_init(&ctx_aes);

    // Output buffer.
    // len previously retrived from read secret in NVS
    uint8_t decrypted[len];

    uint8_t aes_key[32];
    esp_aes_context ctx_aes;
    esp_aes_init(&ctx_aes);

    // SET_DECRYPTION_KEY_HERE;

```

Sample Code – AES-CBC Decryption (2)

```
        uint8_t iv_copy[16];
        memcpy(iv_copy, iv, sizeof(iv));

        mbedtls_aes_crypt_cbc(
                    &ctx_aes,
                    MBEDTLS_AES_DECRYPT,
                    len,
                    iv_copy,
                    enc_secret,
                    decrypted
        );

        mbedtls_aes_free(&ctx_aes);
    }

```

Source Code of the Project

- https://github.com/marcooliani/arduino_esp32_mbedtls
- Sorry for the ugliness of the code!

Thank you for your attention!