Unlimited Results: Breaking Firmware Encryption of ESP32-V3

Karim M. Abdellatif, Olivier Hériveaux, and Adrian Thillard

Ledger, Donjon

Abstract

Because of the rapid growth of Internet of Things (IoT), embedded systems have become an interesting target for experienced attackers. ESP32 [13] is a low-cost and low-power system on chip (SoC) series created by Espressif Systems. The firmware extraction of such embedded systems is a real threat to the manufacturer as it breaks its intellectual property and raises the risk of creating equivalent systems with less effort and resources. In 2019, LimitedResults [20] published power glitch attacks which resulted in dumping secure boot and flash encryption keys stored in the eFuses of ESP32. Therefore, Espressif patched this vulnerability and then advised its customers to use ESP32-V3, which is an updated SoC revision. This new version is hardened against fault injection attacks in hardware and software as announced by Espressif [12]. In this paper, we present for the first time a deep hardware security evaluation for ESP32-V3. The main goal of this evaluation is to extract the firmware encryption key stored in the eFuses. This evaluation includes Fault Injection (FI) and Side-Channel (SC) attacks. First, we use Electromagnetic FI (EMFI) in order to show that ESP32-V3 doesn't resist EMFI. However, by experimental results, we show that this version contains a revised bootloader compared to ESP32-V1, which hardens dumping the eFuse keys by FI. Second, we perform a full SC analysis on the AES accelerator of ESP32-V3. We show that an attacker with a physical access to the device can extract all the keys of the hardware AES-256 after collecting 60K power measurements during the execution of the AES block. Third, we present another SC analysis for the firmware decryption mechanism, by targeting the decryption operation during the power up. Using this knowledge, we demonstrate that the full 256-bit AES firmware encryption key, which is stored in the eFuses, can be recovered by SC analysis using 300K power measurements. Finally, we apply practically the firmware encryption attack on Jade hardware wallet [4].

Keywords— ESP32-V3, EMFI, Side-Channel Attacks (SCAs), eFuses

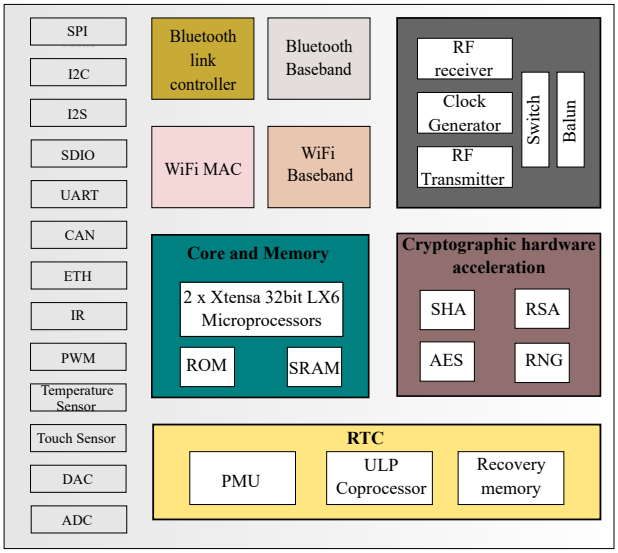


Figure 1: ESP32 architecture [11]

1 Introduction

Embedded systems applications range from smart cards, mobile devices, hardware wallets, to industrial control. ESP32 is one of these chips which are widely used in such systems [11] such as cameras for video streaming, health care applications, and recently ESP32-V3 in security devices such as hardware wallets [4]. It is a single 2.4 GHz Wi-Fi-and-Bluetooth chip designed with the TSMC 40 nm technology. In addition, it has an integrated Wi-Fi solution for Bluetooth IoT applications. According to Espressif, it is designed for mobile devices, wearable electronics, and IoT applications. It is equipped with the following security features as shown in Fig. 1:

- IEEE 802.11 standard security features all supported, including WFA, WPA/WPA2 and WAPI

Figure 2: ESP32-V1

Figure 3: ESP32-V3

- Secure boot
- **Flash memory encryption**
- 1024-bit OTP, up to 768 bits for customers
- Cryptographic hardware accelerators: **AES**, SHA-2, RSA, Elliptic Curve Cryptography (ECC), Random Number Generator (RNG)

Era of IoT makes devices such as ESP32 vulnerable to hardware attacks (e.g., SC and FI attacks). These attacks have been considered as a known threat to embedded systems since more than 20 years. SCAs are a class of vulnerabilities that leverage physical leakage, such as power consumption or electromagnetic emanations, to gain information about sensitive data being manipulated by a device. Since their inception by Paul Kocher in the late 1990s with the discovery of timing attacks [19], several SCAs such as Differential Power Analysis (DPA) [18], Correlation Power Analysis (CPA) [6], and template attacks [7] have shown a very high efficiency against embedded systems. FI attacks are used to perform malicious data modifications by influencing the execution of the chip, in order to bypass sensitive operations such as memory protection and secure boot. In addition, faulted responses during the execution of cryptographic operations can be analyzed using Differential Fault Analysis (DFA) attacks [3] to extract secret keys. These attacks can be performed by injecting faults using laser/optical [23], electromagnetic [8], and glitches (power and clock) [21].

In 2019, LimitedResults [20] presented FI attacks using power glitches on ESP32-V1, which resulted in revealing flash encryption and secure boot keys stored in the eFuses of ESP32-V1. The main idea of this attack is to inject a single power glitch during the power up, when the memory protection bits are manipulated. At the same time, Raelize [22] also presented an EMFI attack to bypass secure boot and flash encryption by targeting the power up using a single EM pulse.

As a positive reaction from Espressif [12], ESP32-V3 has been released in the market since March 2020 as shown in Fig. 3 to replace ESP32-V1 because of the following security enhancements:

- ESP32-V3 is hardened against fault injection attacks in hardware and software. Therefore, it prevents any FI attacks to dump the eFuse keys.
- It supports a new RSA-based secure boot scheme (ESP32 Secure Boot V2) where the eFuse memory contains only the public key unlike the previous version which was based on AES-256.
- It has a new feature to permanently disable the UART Download Mode via eFuse (UART-Disable).

From the above new added features of ESP32-V3, we can conclude that the only sensitive data stored in the eFuses is only the firmware encryption key to protect the firmware.

In this work, we extensively evaluate ESP32-V3 against EMFI and SC attacks in order to extract the firmware encryption key. First, we show by experimental results that ESP32-V3 doesn't contain any hardware countermeasures that could thwart EMFI. This is achieved by faulting a simple application on ESP32-V1 and ESP32-V3 circuits. However, it contains checks on the boot ROM level that harden dumping the eFuse keys using FI. This conclusion is obtained by reproducing the FI attack presented by LimitedResults on ESP32-V1 and following the same attack path on ESP32-V3 using EMFI. Second, we change the attack strategy from FI to SC. A full SCA is presented against the hardware AES accelerator of ESP32-V3. We show that an attacker with a physical access to the device can extract all the keys of the hardware AES-256 accelerator after collecting 60K power measurements. Third, we present a full SCA on the firmware decryption mechanism. This is achieved by locating the firmware decryption process during the power. We confirm after the experimental analysis that the AES-256 core used for flash encryption/decryption is not the same hardware AES accelerator. By using a flash emulator to control the bootloader and collecting 300K traces during the power up, all the 256-AES keys stored in the eFuses are extracted. Finally, we present a practical example of extracting the firmware of Jade hardware wallet [4].

The remainder of this paper is organized as follows. Section 2 provides a detailed review on the security of ESP32-V3. In Section 3, we detail the full EMFI evaluation of ESP32-V3 and the comparison with ESP32-V1. Section 4 presents the SCA on the hardware AES accelerator of ESP32-V3. In Section 5, we highlight the SCA on the firmware decryption mechanism to dump the firmware encryption keys. Section 6 shows a practical example of extracting the firmware of Jade wallet. Section 7 concludes this work.

2 ESP32-V3 Security

ESP32 (including V3) has a 1024-bits eFuse memory (see [13]). It is divided into 4 blocks of 256 bits each as shown in Fig. 4. BLK0 is used entirely for system purposes. BLK1 and BLK2 are used to store flash encryption and secure boot keys, respectively. BLK3 can be reserved for the custom MAC address, or used entirely for user application. Once

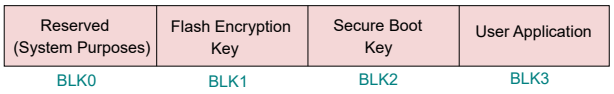


Figure 4: eFuses overview [11]

these keys are written in the eFuses memory, any software running on the ESP32 can not read (or update) them (disabled software readout). Only the ESP32 hardware can read and use those keys for performing secure boot and flash encryption.

2.1 Secure Boot

Secure boot is one of the ESP32 (including V3) security features, which ensures running trusted and signed (by a known entity) applications on the chip. There are two versions of ESP32 secure boot:

Secure Boot V1 : It is used for previous ESP32 versions (prior to ESP32-V3). The digest is generated after hashing the result of encrypting the bootloader with the chip's public key using BLK2 key, as shown in Eq. 1. Then, the digest should be stored at address 0x00 and the bootloader is at 0x1000. During the power up, the boot ROM generates a digest from the bootloader stored in the flash using eFuse BLK2 and compares it with the digest stored at address 0x00 to validate the bootloader as shown in Fig. 5. It can be activated using the command shown in List. 1 through the *espefuse.py* script provided by Espressif. After the fault injection attacks reported in [20][22], Espressif designed secure boot V2 which is based on RSA signature verification, in order to avoid storing the secure boot key in the chip.

$$Digest = \text{SHA-512}(\text{AES-256}((Bootloader \,||\, publickey), BLK2))) \quad (1)$$

```
espefuse.py burn_efuse ABS_DONE_0
```

Listing 1: Enabling secure boot V1

Secure Boot V2 : In order to avoid storing the secure boot key in the eFuses, V2 was proposed. It is based on a public key signature verification using RSA. The bootloader

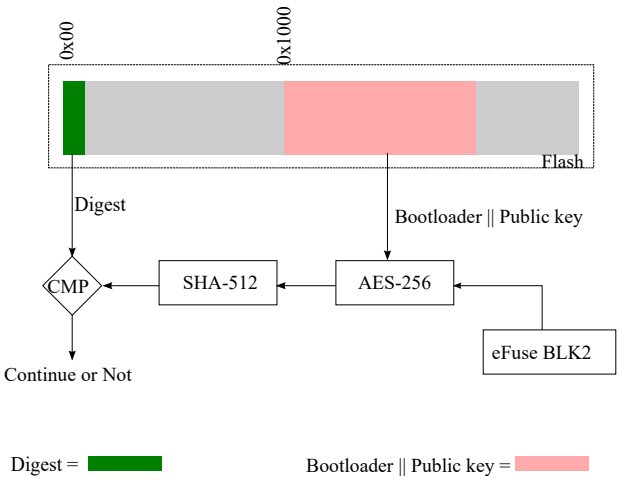


Figure 5: Signature verification of V1

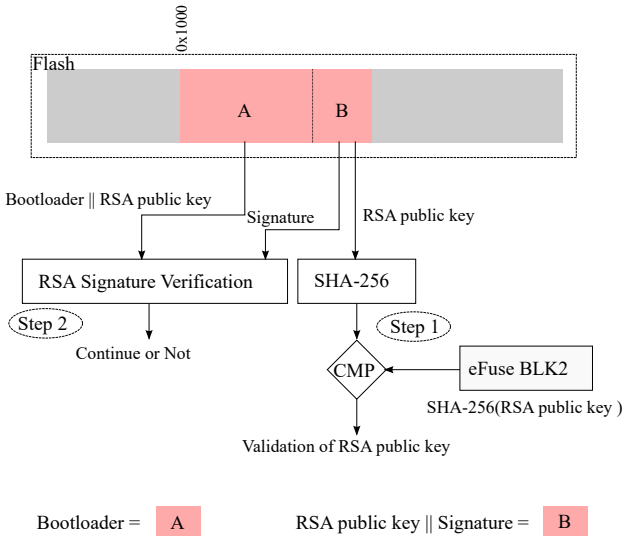


Figure 6: Signature verification of V2

image stored in the flash contains the main bootloader, an RSA public key, and an RSA signature. Before activating this mode, the bootloader is signed by a user private key using the esptool program and stored at address 0x1000. In addition, eFuse BLK2 stores the SHA-256 digest of the RSA public key, to ensure the correctness of the public key generated from the main private key. During the power up, the boot ROM looks up the public key stored in the bootloader's image, and validates its SHA-256 digest with the value stored in eFuse BLK2. After the successful validation, the RSA signature verification is executed as shown in Fig. 6. Also, it can be activated using the command shown in List. 2 through the esptool program.

```
espefuse.py burn_efuse ABS_DONE_1
```

Listing 2: Enabling secure boot V2

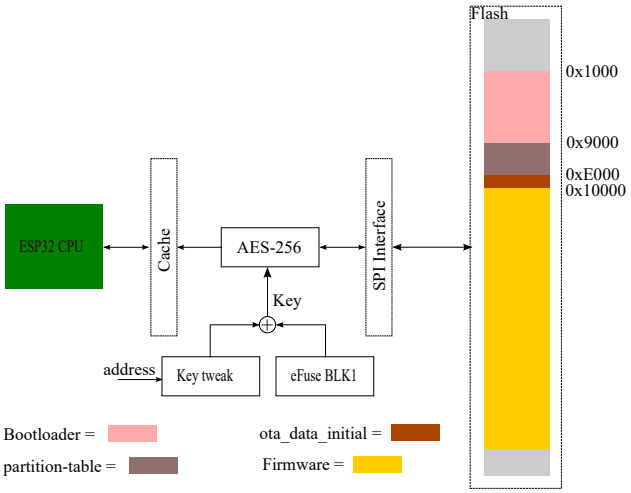


Figure 7: Flash encryption of ESP32

2.2 Flash Encryption

When the ESP32 (including V3) is used, any firmware or data is stored in the external SPI flash memory. It may contain proprietary firmware and sensitive user data, such as user keys, credentials for gaining access to a private network, etc. The flash encryption block encrypts all the flash content using AES-256 and writes encrypted code to the external flash memory for enhanced security. When the CPU reads the external flash through the cache, the flash decryption block can automatically decrypt instructions and data read from the external flash (see Fig. 7), thus providing hardware-based security for application code. According to Espressif [13], flash encryption uses AES-256 decryption and flash decryption uses AES-256 encryption.

During the power up, the firmware bootloader reads the *FLASH_CRYPT_CNT* eFuse value. If the value is an even number of bits sets, it configures and enables the flash encryption block to decrypt the flash content. The flash encryption key is stored in the eFuses BLK1 internal to the chip and, by default, is protected from software access. This key is "tweaked" with the offset address of each 32 bytes block of flash (key tweak). This means that every 32 bytes block (two consecutive 16 bytes AES blocks) is encrypted with a unique key derived from the flash encryption key.

The generated tweak depends on the *FLASH_CRYPT_CONFIG* eFuse setting. It is a 4 bits eFuse slot where each bit enables XORing of a particular range of the key bits where:

- Bit 0, bits 0-66 of the key are XORed
- Bit 1, bits 67-131 of the key are XORed
- Bit 2, bits 132-194 of the key are XORed
- Bit 3, bits 195-256 of the key are XORed

According to Espressif, It is recommended that *FLASH_CRYPT_CONFIG* is always left at the default value 0xF, so that all key bits are XORed with the block offset. More details about *FLASH_CRYPT_CONFIG* are shown in [15]. In order to enable the flash encryption as recommended by Espressif, the following commands are executed through the esptool program:

```
espefuse.py --port PORT burn_key flash_encryption
    flash_encryption_key.bin
espefuse.py --port PORT burn_efuse
    FLASH_CRYPT_CONFIG 0xf
espefuse.py --port PORT burn_efuse FLASH_CRYPT_CNT
```

Listing 3: Enabling flash encryption

2.3 Disabling JTAG and UART Bootloader

The eFuse has one-time programmable bit fields that allow the user to disable support for JTAG debugging and UART bootloader. By default, enabling flash encryption and/or secure boot will disable JTAG debugging. On first boot, the bootloader will burn an eFuse bit to permanently disable JTAG at the same time it enables the other features. As a countermeasure to the previous physical attacks [20][22], Espressif added another countermeasure to ESP32-V3, in order to disable the UART bootloader and disable any eFuse read commands. To activate this feature, the following command is executed through the esptool program:

```
espefuse.py --port PORT burn_efuse
    UART_DOWNLOAD_DIS 1
```

Listing 4: Enabling UART-disable

3 EMFI Evaluation

After highlighting the security of ESP32-V3 in the previous section, we evaluate these countermeasures against EMFI. In order to understand clearly the impact of such countermeasures on ESP32-V3, reproducing the previous attack of [20] is an important step to understand clearly the countermeasures added in ESP32-V3. Therefore, the evaluation methodology that we will follow is to first study the EMFI on ESP32-V1 and then ESP32-V3.

3.1 Setup

In order to obtain a stable setup, ESP32-V1 and ESP32-V3 circuits were soldered on a fabricated PCB with the minimum features for the normal operation as shown in Fig. 8.

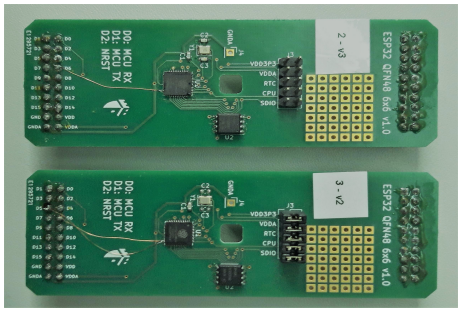


Figure 8: Fabricated PCB for ESP32

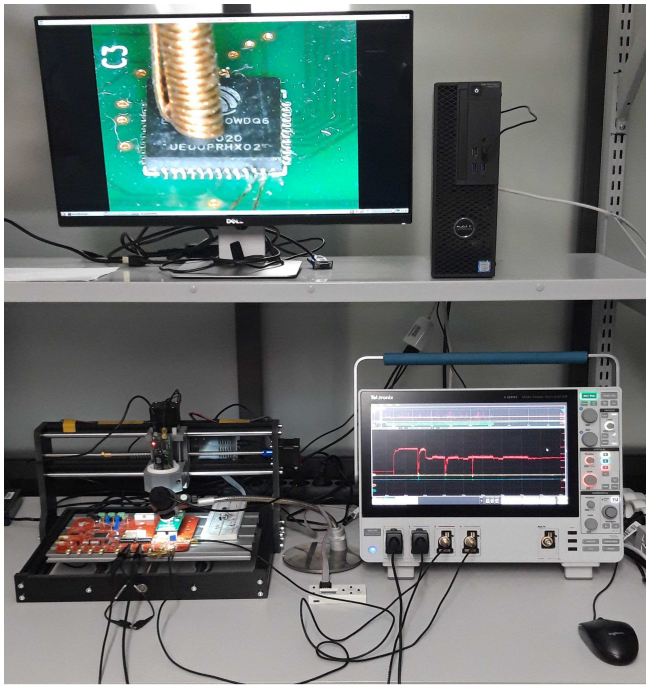


Figure 9: EMFI setup

We used Scaffold [17], which is an open source hardware evaluation board, in order to communicate with ESP32 chips. SiliconToaster [1] was used to inject EM pulses up to 1 kV. A Tektronix MSO44 200 MHz digital oscilloscope with a maximum sampling rate of 6.25 GS/s, was used to measure and capture the instantaneous power consumption of the DUT. The Scaffold board is fixed on an XYZ table in order to scan the chip during the experiment. The overall setup is shown in Fig. 9 and Fig. 10.

```
digitalWrite(4, HIGH); //Trigger High
for (i = 0; i < 500; i++) {}
digitalWrite(4, LOW); //Trigger Low
Serial.print(i);
if(i != 500){
  Serial.print("Faulted");
} else{
  Serial.print("Ok");
}
```

Listing 5: Glitchable application

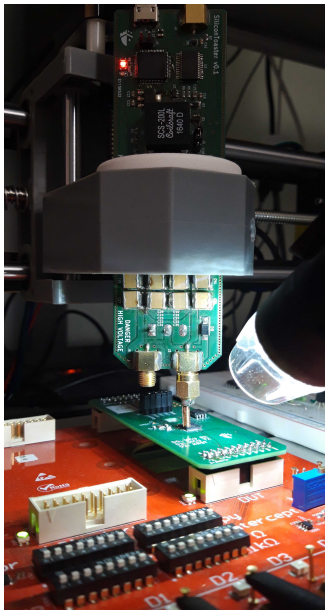


Figure 10: EM probe on ESP32

Algorithm 1: eFuse attack methodology

```
Data: N = Nb trials = 50
i = 0;
while True do
    PrepareFault();
    ChipRestart();
    CaptureTrace();
    ReadeFuse();
    i += 1;
    if (i == N) then
        break;
        MoveProbe()
```

3.2 Attack Methodology

As we mentioned before, in order to understand the new security enhancements of ESP32-V3, it is relevant to reproduce the attack of [20]. Two main fault injection attempts are performed on ESP32-V1 and ESP32-V3. We start with a glitchable application for the basic characterization of the target as shown in List. 5. The main idea of chip characterization is to reduce the fault parameters research such as pulse width and find out the sensitive area to EMFI. After that, the eFuse attack is performed following Algorithm 1. It shows

the overall scenario of the attack for each physical spot of the scanned area. `PrepareFault` function includes fault parameter preparation such as pulse width and offset. The chip is faulted during the power up, when the eFuse protection bits are manipulated (during `ChipRestart()`).

3.3 EMFI on ESP32-V1

The initial characterization was done with a simple "loop" application as shown in List. 5, which is commonly used as an initial evaluation step. We adjusted the SiliconToaster settings to inject EM pulses with 450V and the EM probe scanned all the chip surface using **0.25mm** motor step in order to break the glitchable application. Several physical spots have been found vulnerable to EM pulses as shown in Fig. 11.

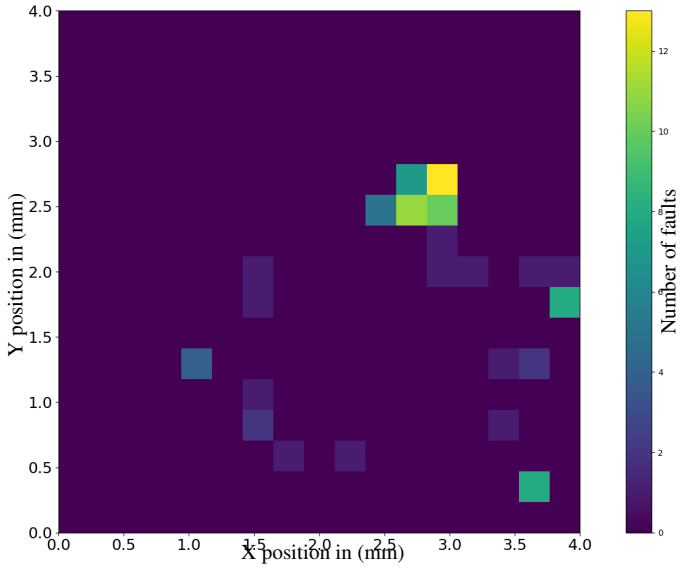


Figure 11: Successful fault map of the glitchable application

The following step is to attack the eFuse memory using the same methodology. The main idea is to target the power up process when the eFuse protection bits are manipulated. Fig. 12 shows the power consumption of ESP32-V1 during the power up. We scanned the chip using smaller motor step (0.1mm). For each physical spot, Algorithm 1 was executed. We found that the first chip activity after the power up is the interesting shooting zone for the successful faults. Fig. 13 shows the the power consumption of the chip in case of a successful fault. The fault map that shows the physical spots of successful faults is presented in Fig. 14. The following subsection will highlight the same methodology on ESP32-V3.

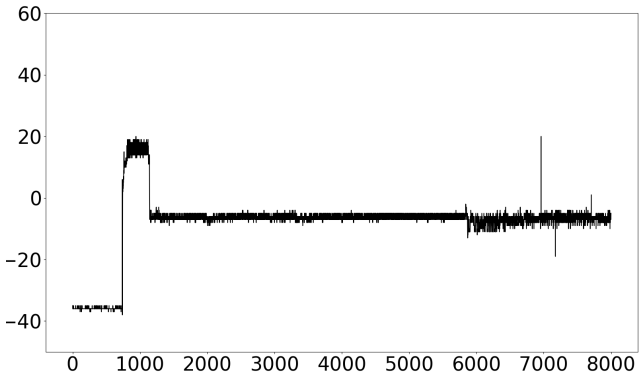


Figure 12: Power up of ESP32-V1

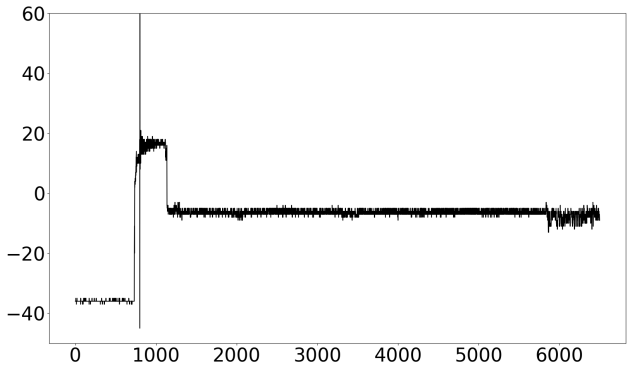


Figure 13: Power consumption in case of a successful fault of ESP32-V1

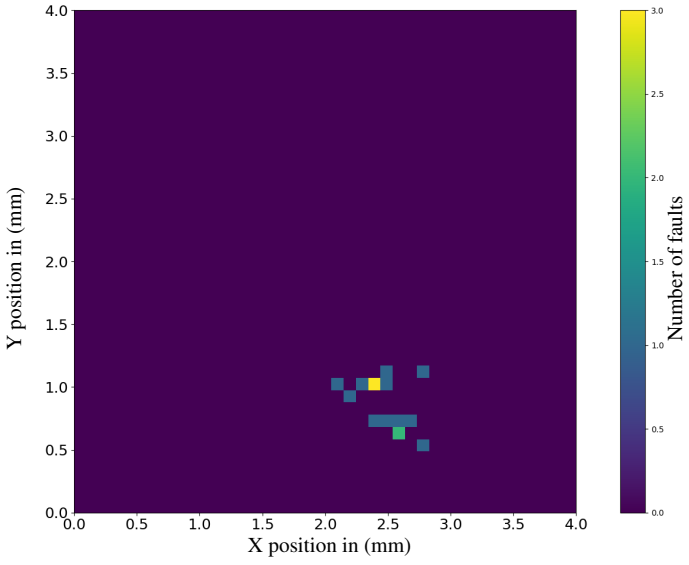


Figure 14: Successful fault map on ESP32-V1

3.4 EMFI on ESP32-V3

In order to verify that ESP32-V3 is not equipped with any hardware against EMFI, we evaluated the chip using the glitchable application. We used the same parameters of ESP32-V1 evaluation such as motor step, pulse voltage, timing, and pulse width. Fig. 15 shows the success map of the EMFI effect in case of using the glitchable application. Therefore, we can conclude that ESP32-V3 has no any hardware countermeasure against EMFI.

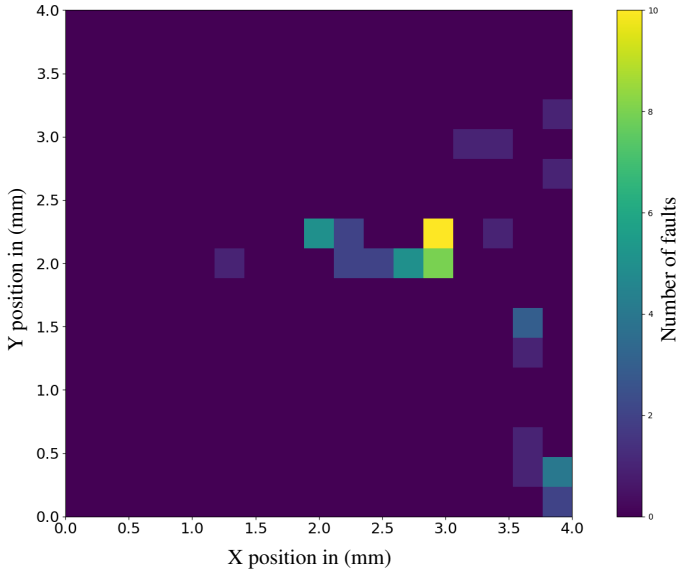


Figure 15: Successful EMFI effect on the glitchable application of ESP32-V3

The next step is to dive into the eFuse attack using EMFI. We followed the same strategy of ESP32-V1. The first step is to understand the power up trace and compare it with ESP32-V1 to see if there is a difference between them. Therefore, we measured the power consumption of the chip during the power up as shown in Fig. 16. It is clear from this power trace that ESP32-V3 contains three additional activities compared to ESP32-V1 that reflects the fault countermeasures on the boot ROM level.

We followed the methodology of injecting multiple faults during the boot ROM countermeasures as shown in Fig. 17. After scanning the chip several times using random number of pulses with random offsets during the boot ROM countermeasures, we haven't obtained successful faults but we obtained several cases where the chip is crashed because of the multiple fault pulses. Fig. 18 shows the physical spots where the chip is crashed.

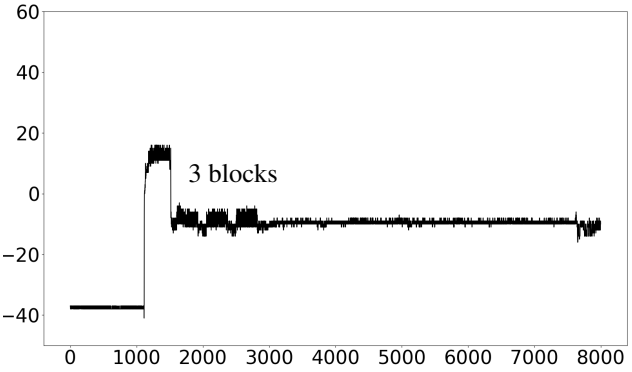


Figure 16: Power up of ESP32-V3

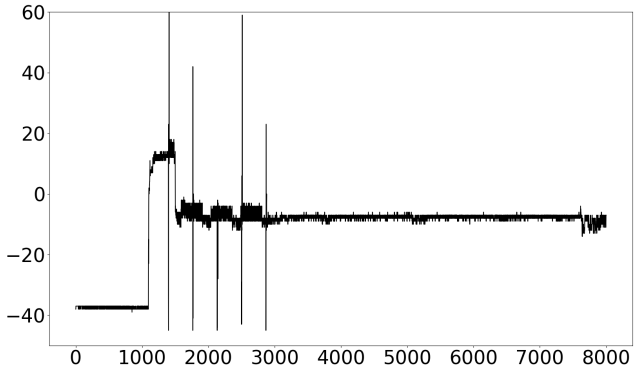


Figure 17: Multiple EM pulses on ESP32-V3

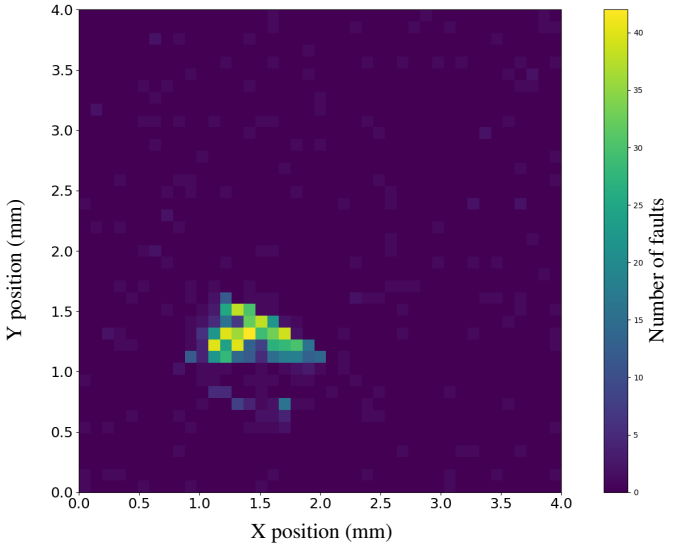


Figure 18: Chip crash map

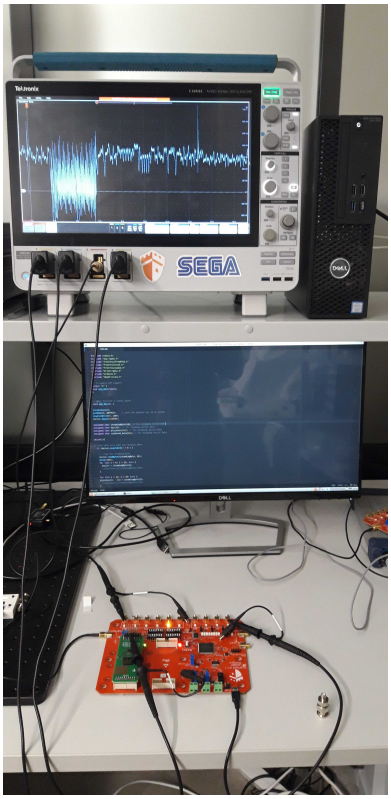


Figure 19: Overall setup

3.5 Discussion

In this section we have shown the difference between V3 and V1 of ESP32. It was clear that the major difference between them is the boot ROM countermeasures that confirms V3 is patched against FI by adding verification countermeasures during the power up. However, V3 doesn't resist EMFI which was proven by breaking the glitchable application. Moreover, multiple EMFI pulses haven't yet succeeded against V3.

4 Breaking AES Core by SCAs

In this section, we present a detailed SC analysis of the hardware AES accelerator deployed in ESP32-V3. For the sake of clarity, the following techniques will be used in order to evaluate the AES core:

Correlation Power Analysis (CPA) [6]: A type of SCAs that has been demonstrated to be a realistic threat to many critical embedded systems. It aims at finding correlation between an estimated leakage model (Hamming weight (HW) or Hamming Distance (HD)) and the actual power output of a device. The attack success depends on the accuracy of the leakage model (such as Hamming Weight (HW) or Hamming Distance (HD)) which is mathematically correlated with the actual power measurements. The correlation coefficient ρ between the actual power consumption and the predicted model can be calculated using Eq. 2. X represents a set of real power consumption values while Y is a set of the predicted leakage model values. The Cov is the covariance operation while Var is the variance operation.

$$\rho = \frac{Cov(X,Y)}{\sqrt{Var(X)}\sqrt{Var(Y)}} \tag{2}$$

Leakage detection: A methodology to identify leakage moments which contain sensitive information. It reduces the computation complexity of security evaluation and improves the efficiency of the SCAs. Several methods have been used to identify the amount of leakage such as T-test [16] and NICV [2]. In this paper, we chose to use the signal-to-noise ratio (SNR) to detect these points of interest. The formula of SNR is given in Eq. 3. It gives the ratio between the deterministic data-dependent leakage and the remaining noise. X represents the random variable associated to the real power traces, Y is the label which is determined, E is the expectation, and Var is the variance of a random variable.

$$SNR = \frac{Var(E(X|Y))}{E(Var(X|Y))} \tag{3}$$

4.1 Setup

We used the same PCB shown in Fig. 8 in order to be easily plugged into Scaffold [17]. Fig. 19 shows the overall setup which contains the following components:

- A Scaffold board with the device under test (DUT).
- A PC to communicate with the Scaffold-based DUT.
- A Tektronix MSO54 1 GHz digital oscilloscope with a maximum sampling rate of 6.25 GS/s, used to measure and capture the instantaneous power consumption of the DUT during the execution of the AES.

4.2 Calling AES accelerator

In order to evaluate the AES accelerator, a custom firmware that triggers an AES encryption using a controlled key and plaintext has been developed (see List. 6). It has been designed and compiled using ESP-IDF [10]. To call the AES accelerator, we configured the following options in the toolchain build settings:

- *Enabling the hardware AES Accelerator* was selected from `Component-config` → `mbedTLS`.
- *Setting the frequency to 160 MHz* by setting `Component-config` → `ESP32-specific` → `CPU frequency` to 160 MHz.

```
while (1) {
  if (Serial.available() > 0) {
    Serial.readBytes(incomingBytes, 48);
    delay(100);
    for (int i = 0; i < 32; i++) {
      key[i] = incomingBytes[i];
    }
    for (int i = 32; i < 48; i++) {
      plaintext[i - 32] = incomingBytes[i];
    }
    delay(100);
    esp_aes_context aes;
    esp_aes_init(&aes);
    esp_aes_setkey( &aes, key, 256);
    digitalWrite(4, HIGH); // Trigger = 1
    // Start AES-256
    esp_aes_crypt_ecb(&aes, ESP_AES_ENCRYPT,
    plaintext, ciphered_data);
    digitalWrite(4, LOW);  // Trigger = 0
    esp_aes_free(&aes);
    delay(100);
    for (int i = 0; i < 16; i++) {
      char st[3];
      sprintf(st, "%02x", (int)ciphered_data[i]);
      Serial.print(str);
    }
  }
}
```

Listing 6: Calling AES-256

4.3 Side-Channel Analysis

In order to evaluate the AES accelerator called in List. 6 against CPA attacks, 400K traces were collected during the execution of the AES under the variation of the plaintext **P** and the key **K** as shown in Fig. 20. To improve the statistical analysis, power traces were aligned using cross-correlation, as shown in Fig. 21(a).

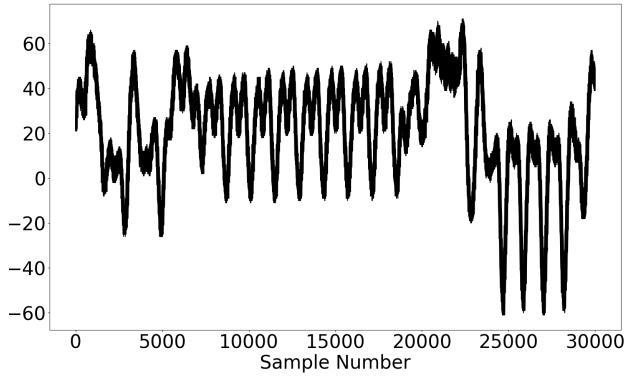


Figure 20: Power traces during the execution of List. 6 (50 traces without alignment)

4.4 Locating the AES encryption

According to Espressif's specification [13], the AES encryption takes 15 cycles for 256 bits key. Running at 160MHz, we hence expect the encryption time to be close to 93ns. Sampling to the maximal frequency of the used oscilloscope scope (6.25GS/s) would hence imply that slightly more than 585 points are measured during the running time of the whole AES.

To measure the amount of data leakage in the traces, we computed the signal-to-noise ratio (SNR), whose computation is given in Eq. 3. We calculated the SNR for the values of **P**, **K**, the ciphertext **C**, and **S-box** output, in order to locate the AES precisely before performing the CPA attack. Fig. 21(a) shows a sample of aligned traces, Fig. 21(b) shows the SNR for **K** (in blue), **P** (in red) and **C** (in yellow), and Fig. 21(c) presents the SNR of the S-boxes related to the first two rounds.

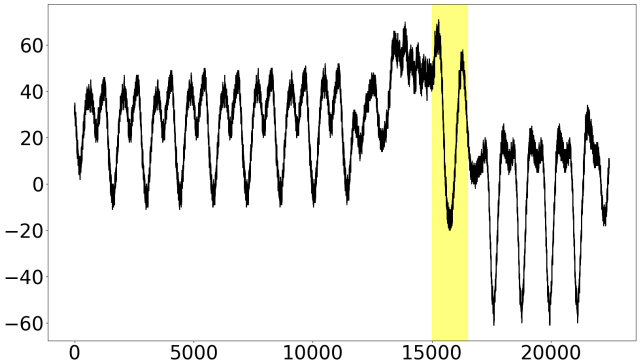
We can conclude that the 32 bytes of **K** are manipulated at the beginning, **P** bytes are then loaded and finally, **C** bytes are stored. This scenario is matched with the code shown in List. 7 [9]. After locating the leakage of the first two rounds and the leakage of **C**, the AES can be located easily: it starts from the leakage of the first two rounds, and ends with the leakage of **C** (the yellow zone on the trace).

```
int esp_aes_crypt_ecb( esp_aes_context *ctx, int
    mode, const unsigned char input[16], unsigned
    char output[16] )
{
    esp_aes_acquire_hardware();
    // Load Key bytes
    esp_aes_setkey_hardware(ctx, mode);
    // Load P and execute the AES encryption
    ets_aes_crypt(input, output);
    esp_aes_release_hardware();
    return 0;
}
```

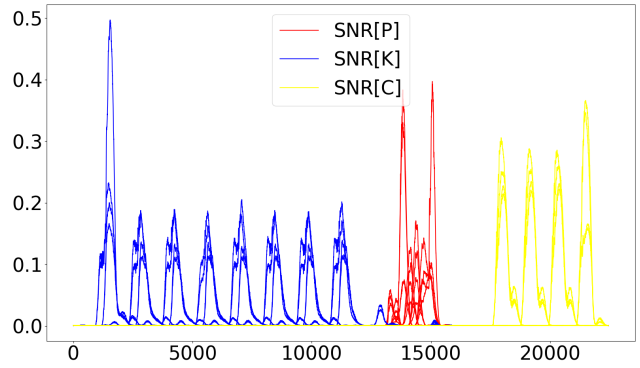
Listing 7: Execution of the hardware AES

4.5 AES Attack

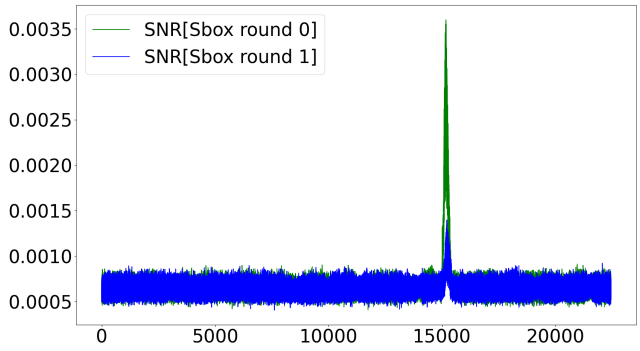
After identifying the AES leakage, we tested our setup in an attack context. 70K traces were acquired with a fixed key **K** (supposed unknown) and variable plaintexts **P**, as shown in Fig. 22 using better acquisition vertical resolution. AES-256 relies on a 256-bits key, and hence requires the knowledge of two round keys. Therefore, CPA was performed on the first two rounds. The leakage models for first round $round_0$ and second round $round_1$ are defined in Eq. 4 and Eq. 5, respectively. The rank of the correct key is used as a success metric for the attack. It is computed in function of the number of traces. After performing the success rate of all the keys using the previous leakage models, all the 32 bytes of the AES-256 key are recovered within 60K traces as shown in Fig. 23.



(a) 50 traces after alignment



(b) SNR of **P, K,** and **C**



(c) SNR of the first and second rounds

Figure 21: AES leakage detection

$$Model_{round_0}[i] = HW(Sbox[P[i] \oplus guess]) \quad (4)$$

$$Model_{round_1}[i] = HW(Sbox[State_1[i] \oplus guess] \oplus Sbox[P[i] \oplus K[i]]) \quad (5)$$

where HW is the Hamming Weight.

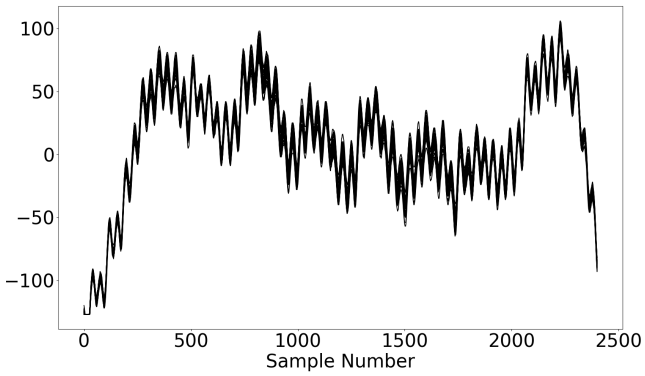


Figure 22: Zoom on the AES execution

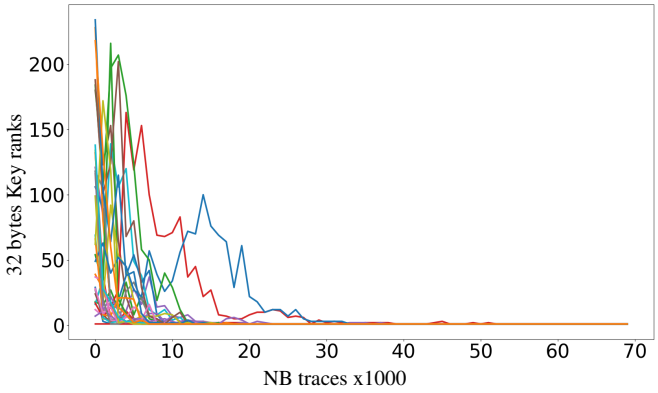


Figure 23: 32 bytes Key ranks

4.6 Discussion

In this section, we presented a detailed SCA on the AES accelerator of ESP32-V3. It shows that the hardware AES core integrated in ESP32-V3 is not resistant against SC attacks. An attacker can extract the whole AES-256 key using only 60K power traces.

5 Breaking Firmware Encryption of ESP32-V3 by SCAs

In this section, we present a detailed side-channel analysis of the flash encryption mechanism deployed in ESP32-V3. This is achieved by targeting the power up of ESP32-V3 in order to locate the firmware decryption.

5.1 Power Up During Flash Encryption

After burning the encryption key, the flash encryption feature was activated as shown in List. 3. We looked at the power consumption of the chip during the power up. In addition,

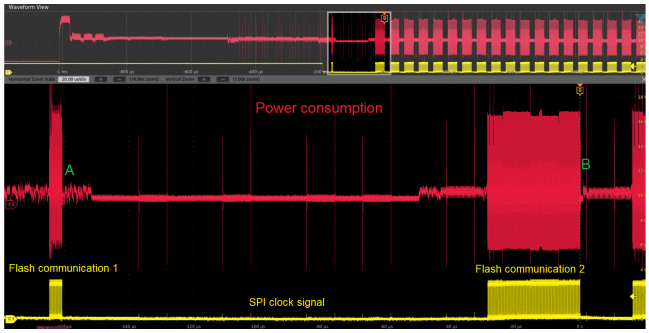
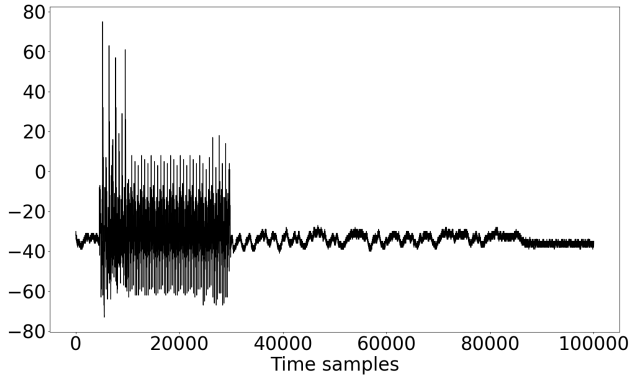


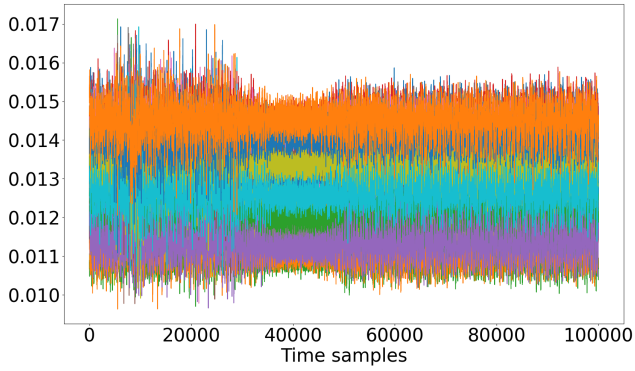
Figure 24: Power up during flash encryption

Algorithm 2: Traces measurement sequence

```
Data: N = NB traces = 50000
i = 0;
while True do
    FlashData = Random(32);
    EraseFlash();
    WriteFlash(FlashData,address = 0x1000);
    ChipRestart();
    CaptureTrace();
    i += 1;
    if (i == N) then
        break;
```



(a) Power trace on zone A



(b) SNR of the first 32 bytes on zone A

Figure 25: Leakage detection during zone A

we monitored the SPI clock signal of the external flash to find where flash data are manipulated as shown in Fig. 24.

According to the reference manual [13], data are loaded from the flash as 32 bytes block which are decrypted using the same key in case of activating the flash encryption. From Fig. 24, there are two zones where the flash data is manipulated (flash communication 1 and flash communication 2). Therefore, the firmware decryption should be located in zone A or zone B.

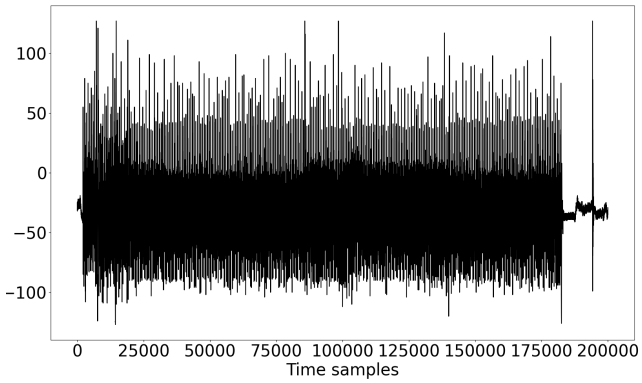
During the power up, the ROM bootloader loads the firmware bootloader which is stored at address 0x1000 as shown in [14]. Hence, the first 32 bytes stored at 0x1000 should be manipulated during flash communication 1 or flash communication 2. To locate the manipulation of the firmware bootloader data, we collected traces during the two zones under the scenario shown in Algorithm 2. After collecting 50K traces, we performed the SNR of the first 32 bytes stored at 0x1000, on the two zones (flash communication 1 and flash communication 2) to identify which zone is relevant.

Fig. 25 shows the SNR of the first 32 bytes stored at address 0x1000 on flash communication 1 zone. We can observe that the SNR is very low, and that no point of interest can be identified in this zone. In addition, by spying on the SPI interface during this period, we found that the flash ID is communicated during this time. However, the SNR on the second zone (flash communication 2) is very high as shown in Fig. 26 and it highlights 32 different leakages related to the first 32 bytes stored at address 0x1000. Therefore, the AES decryption should be located around Zone B (after the manipulation of the bootloader data).

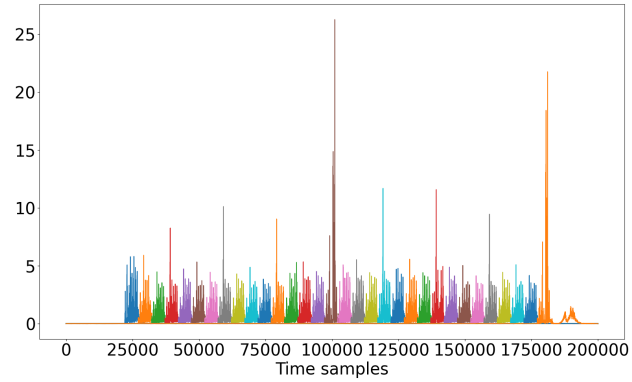
5.2 AES Encryption

After being more confident regarding the zone of the AES decryption, we captured 100K traces with better vertical resolution around zone B after the flash read. As we know the eFuse BLK1 and the tweak value at address 0x1000, we performed the SNR on the Ciphertext (C) of the first 16 bytes (C0) and the last 16 bytes (C1). In Fig. 27 that shows the SNR of C0 and C1, there are some peaks resulting from the leakage of C0 and C1 bytes. The leakage of C1 bytes can tell that the AES-256 decryption of the last 16 bytes is executed before the leakage of the first C1 bytes (before the time sample 1750).

Because of the flash limitation of writing and erasing, we changed the flash and collected another 100K traces around the zone where we suspect the AES-256 decryption is performed. We observed the CPA results by monitoring the



(a) Power trace during zone B



(b) SNR of the first 32 bytes during zone B

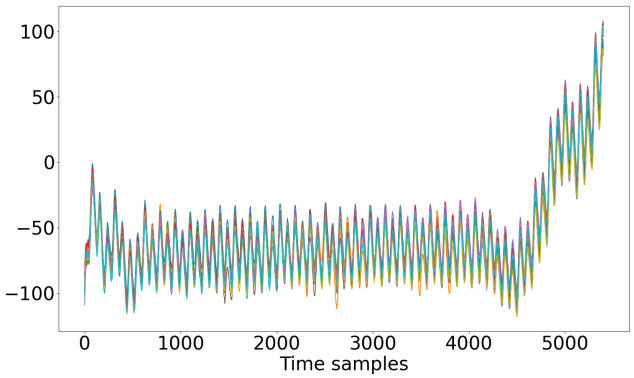
Figure 26: Leakage detection during zone B

correlation scores of all the keys including the correct key. Fig. 28 shows an example of attacking Key[3] by CPA. It shows the correlation scores of all the estimated keys including the correct key[3] in green. It is clear that the correct key has maximum peaks during the yellow zones.

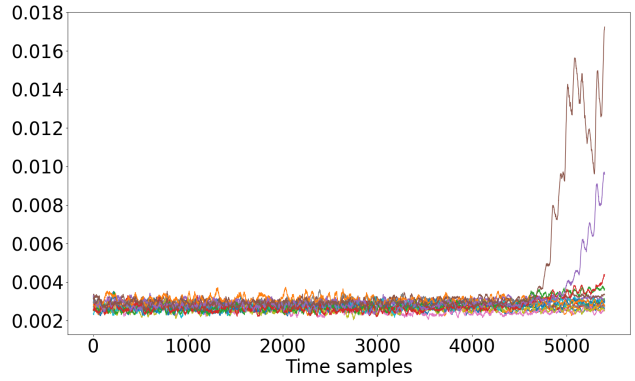
5.3 Flash Emulator

The initial ESP32 testing circuit had a GD25Q32C integrated circuit as the external flash memory. According to the datasheet, this circuit guarantees 100K program/erase cycles. Going beyond is not recommended, and we indeed got flash memory corruption quickly after this limit was reached. This limits the number of traces with different flash memory content, that can be acquired for a SC attack, and replacing the circuit may hardly be possible as a different circuit may have slightly different timing or power behavior.

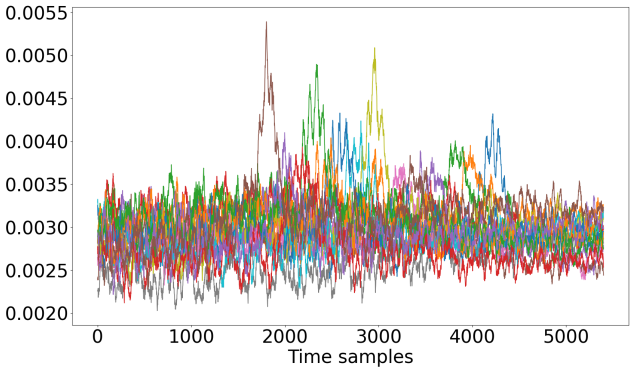
To leverage this flash write limit for the SC traces acquisitions, we decided to replace the external flash memory device with a flash memory emulation. During the SC acquisition, the ESP32-V3 chip starts by sending to the flash memory the *Release from Deep Power-Down or High Performance Mode and Read Device ID* command (`0xab000000`). This



(a) A sample trace after Zone B



(b) SNR of C0 bytes after zone B



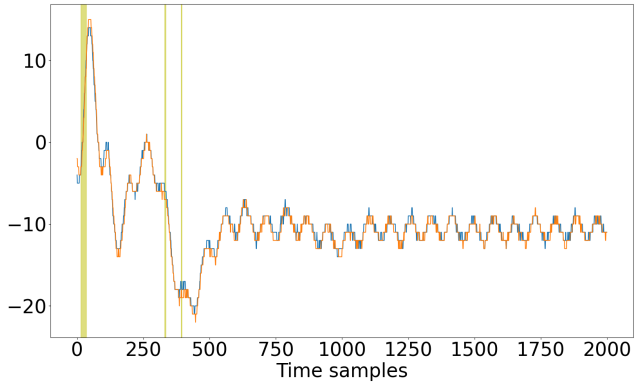
(c) SNR of C1 bytes after zone B

Figure 27: SNR on C0 and C1

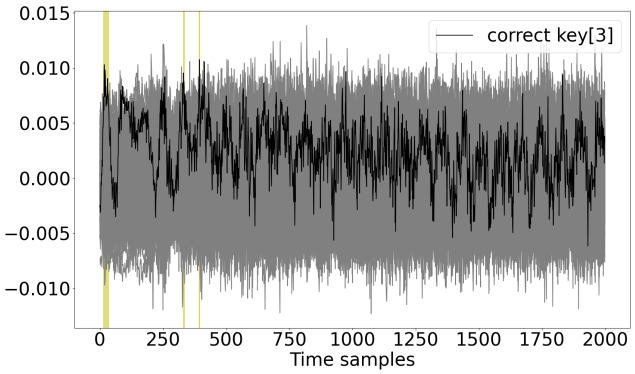
command is acknowledged by the circuit with a `0x15` byte response.

The MCU then sends the *Read Data Bytes* command to read the content of the memory at address `0x1000` (command bytes: `0x03001000`). The flash memory responds by sending the 32 bytes of data stored at the requested address. The MCU decrypts this block as soon as received. The MCU then read more blocks from the flash memory, which are not relevant for the attack.

That is, the communication between the MCU and the

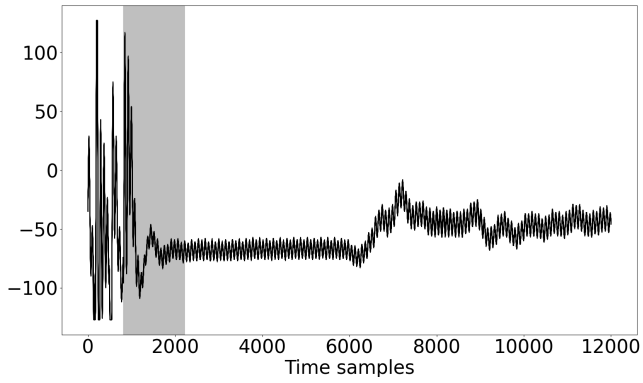


(a) A sample trace after Zone B

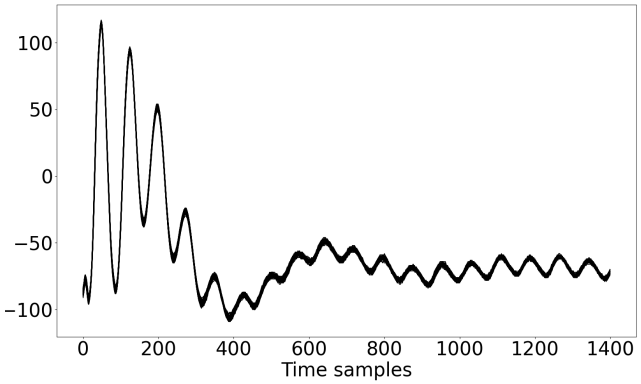


(b) Correlation of the correct third key byte compared to other estimations

Figure 28: Correlation score of the correct K[3]



(a) Sample of non aligned 100 traces



(b) Sample of 100 traces after alignment

Figure 29: Power measurements in case of the flash emulator

flash memory can be predicted before booting, and it is easy to write a small flash emulator which will return a pre-programmed response. We implemented this directly into the Scaffold board FPGA by modifying the original VHDL architecture to add a custom SPI slave module. This is designed as a simple shift register synchronized to the SPI master clock and chip select signals. Flash content data is loaded into the shift register through Scaffold API, which is faster than erasing and programming a real Flash device, thus allowing capturing traces at faster rate.

Furthermore, it happens that our SPI flash emulation has smaller response data jitter than the initial GD25Q32C flash memory. We found received data blocks are decrypted by the ESP32-V3 right after the last SPI bit is transmitted, and as a result, the SPI communication bus has an important influence on the power consumption of the ESP32-V3. Reducing the communication jitter by replacing the Flash memory with our Flash emulation reduced significantly the number of traces required to recover the flash encryption key with the SCA.

5.4 Attacking Firmware Encryption Using Flash Emulator

Besides the flash encryption, we activated all the security features of the chip, which are: secure boot (V1 and V2), flash encryption, and UART-disable. The common point between all the cases is that the bootloader which is stored at address 0x1000 is manipulated at the beginning during the power up. Regarding the UART-disable countermeasure, it doesn't have any impact since the device will read and decrypt the external flash memory whatever the UART disable setting is. We registered 2M traces during the power up. Fig. 29 shows the power traces in case of not using the external flash and replacing it with the Scaffold FPGA shift register. We concentrated on the grey zone where the AES-256 decryption is performed. In order to improve the SCA efficiency, traces were aligned by cross-correlation as shown in Fig. 29.

We tried several leakage models for attacking the first and the second round of the AES-256 by CPA. We found that using the same leakage models as in the previous section (Eq. 4 and Eq. 5), has given the best key ranks compared to other leakage models. Our success metric (key rank) indicates that with 300K traces, all the key bytes except k[0], are recov-

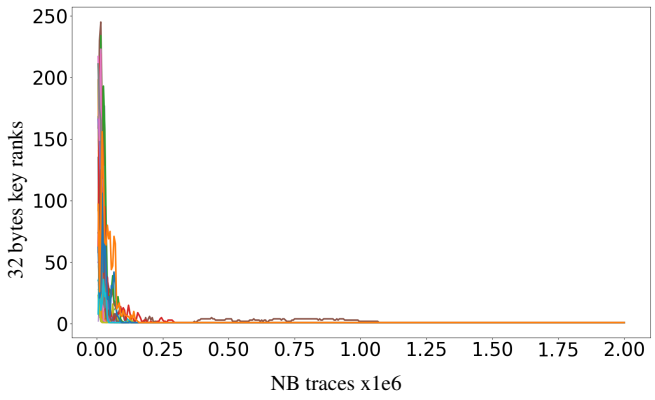


Figure 30: 32 bytes key ranks

Figure 31: Jade wallet before opening the package

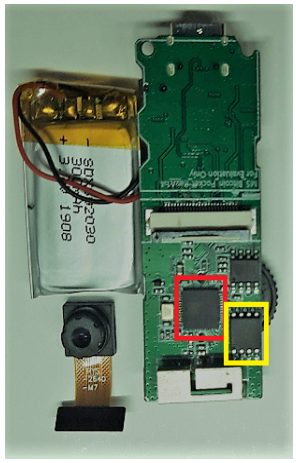


Figure 32: Jade wallet after opening the package

```
7010h: 27 E6 F3 D4 43 F1 95 38 79 CF E3 4C D9 4B 8B 6F  'æóOCñ•8yIãLUK‹o
7020h: 38 7C 28 AC 1D 9E CE D5 EB F2 9A 3F 95 0F CE E8  8|(¬.žÎÕëòš?•.Îè
7030h: 6C 7E 30 FF B9 CA 09 1C BA EE D6 8E EA BD 46 8B  l~0ÿ¹Ê..ºîÖŽê½F‹
7040h: 20 0C 87 BC C1 E2 F7 66 8F F3 82 E9 BF BF 89 61   .‡¼Áâ÷f.ó,é¿¿‰a
7050h: DE 56 0E C4 D1 16 E2 71 9A AA BE D7 29 A5 E5 2B  ÞV.ÄÑ.âqšª¾×)¥å+
7060h: C8 C0 D3 AF 50 4E E1 21 6E 7E 21 F2 EB F5 DB 91  ÈÀÓ¯PNá!n~!òëõÛ'
7070h: 86 AE A0 D0 86 D4 D3 C4 4C 3A B7 D1 70 5E 69 82  †® Ð†ÔÓÄL:·Ñp^i,
7080h: 3A DB 1B A7 64 7D 81 EA 15 12 E3 C0 B2 2F 38 B0  :Û.§d}.ê..ãÀ²/8°
7090h: 17 7E 61 FB 99 70 99 4E 6D B2 61 32 8E EF CF 93  .~aû™p™Nm²a2ŽïÏ“
70A0h: 50 43 44 FE 2A B4 27 65 F5 EA F1 C9 A2 8F 05 3D  PCDþ*´'eõêñÉ¢..=
70B0h: E4 C6 8E A1 B4 81 EB 7F DB EA C8 45 B3 1D C8 A8  äÆŽ¡´.ë.ÛêÈE³.È¨
70C0h: 4E 3D E7 BE 61 9E 30 1E 13 CD 62 2A F2 94 85 2F  N=ç¾až0..Íb*ò"…/
```

(a) Encrypted firmware

```
7010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
7020h: AA 50 01 00 00 D0 00 00 00 20 00 00 6F 74 61 64  ªP...Ð... ..otad
7030h: 61 74 61 00 00 00 00 00 00 00 00 00 01 00 00 00  ata.............
7040h: AA 50 01 01 00 F0 00 00 00 10 00 00 70 68 79 5F  ªP...ð......phy_
7050h: 69 6E 69 74 00 00 00 00 00 00 00 00 01 00 00 00  init............
7060h: AA 50 00 10 00 00 01 00 00 70 17 00 6F 74 61 5F  ªP.......p..ota_
7070h: 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  0...............
7080h: AA 50 00 11 00 00 19 00 00 70 17 00 6F 74 61 5F  ªP.......p..ota_
7090h: 31 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  1...............
70A0h: AA 50 01 04 00 70 30 00 00 10 00 00 6E 76 73 5F  ªP...p0.....nvs_
70B0h: 6B 65 79 00 00 00 00 00 00 00 00 00 01 00 00 00  key.............
70C0h: EB EB FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ëëÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

(b) Decrypted firmware

Figure 33: Jade's firmware before and after decryption

ered as shown in Fig. 30. K[0] can be recovered either using more side-channel traces or through a brute-force. Note that the side-channel literature have proposed so-called Key Enumeration Algorithm methods [24] to efficiently solve such occurrences. These methods might be used to even further reduce the number of required traces. After attacking the decryption key, it should be XORed with the tweak value generated from the offset 0x1000 (which is a known value), in order to obtain the eFuses BLK1. We can see that the power traces shown in Fig. 29 are different from the case of the hardware AES accelerator shown in Fig. 22. In addition, more traces were needed for the firmware encryption case in order to attack all the keys. As a result, we can estimate that the two hardware solutions are different.

5.5 Discussion

This section detailed the SCA of the flash encryption mechanism deployed in ESP32-V3. It proves that this security feature is not protected against SCAs. An attacker can dump the eFuse BLK1 which is responsible for flash encryption after collecting 300K traces during the power up. As a result, all the external flash content (firmware including the IP and the sensitive user data) can be decrypted. Hence, firmwares of ESP32-V3 protected by the encryption mechanism can fall into the hands of attackers or competitors.

6 Practical example

Blockstream Jade [4] is a hardware wallet dedicated to securing cryptocurrencies and its transactions. It supports Bitcoin and its leading sidechains. This hardware wallet is an open-source and open-hardware project [5]. Fig. 31 and Fig. 32 show the Jade wallet before and after opening its plastic package, respectively. The wallet has ESP32-V3 (in red) as the main microcontroller and an external flash memory (in yellow) to store the encrypted firmware. It doesn't store the user PIN in the external flash. The PIN verification is performed remotely on the Blockstream's server. However, the external flash contains the user's private and public keys to communicate with this server. Therefore, decrypting the encrypted firmware stored in the external flash leads to clone the wallet and raises the risk of injecting a backdoor to perform transactions to substituted addresses, which can be called as an evil maid attack.

As a practical application to the proposed SCA on the firmware encryption, we applied this attack on the case of Jade. First, we dumped the encrypted firmware from the external flash using Scaffold. Then, we removed the ESP32-V3 from the PCB of Jade and soldered it on the PCB shown in Fig. 8. We collected 300K traces for attacking all the keys except K[0] which was brute-forced. Finally, we succeeded to decrypt the Jade's encrypted firmware, which is stored in

the external flash. Fig. 33 shows the firmware stored in the external flash before and after the decryption using the attacked key.

7 Acknowledgment

The authors would like to thank Espressif for their fast and positive reply regarding the two attacks highlighted in this paper (hardware AES accelerator and firmware encryption). We reported the two attacks to them in October 2021. They awarded the authors two bounties for the two attacks.

8 Conclusion

In this paper, we presented a deep hardware security evaluation for ESP32-V3. First, we used EMFI in order to attack the firmware encryption key stored in the eFuses. We discovered that this chip version is patched against FI attacks by adding verification blocks on the bootloader level. Therefore, we changed the attack path from FI to SC. We evaluated the hardware AES accelerator of the chip against SCAs. After using a high end oscilloscope with 6.25GS/s and careful alignment, it was broken by collecting only 60K traces. As a next step, we showed another SC analysis for the firmware decryption mechanism. After locating this process during the power up and using a flash emulator, we demonstrated that the full 256-bit AES firmware encryption key, which is stored in the eFuses, can be recovered by SCAs using 300K power measurements. Finally, we applied this attack practically on Jade hardware wallet [4].

References

[1] ABDELLATIF, K. M., AND HERIVEAUX, O. Silicontoaster: a cheap and programmable em injector for extracting secrets. 35–40.

[2] BHASIN, S., DANGER, J.-L., GUILLEY, S., AND NAJM, Z. Nicv: normalized inter-class variance for detection of side-channel leakage. In *2014 International Symposium on Electromagnetic Compatibility, Tokyo* (2014), IEEE, pp. 310–313.

[3] BIHAM, E., AND SHAMIR, A. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference* (1997), Springer, pp. 513–525.

[4] BLOCKSTREAM. Jade wallet. `https://blockstream.com/jade/`.

[5] BLOCKSTREAM. OPen Source of Jade wallet. `https://github.com/Blockstream/Jade`.

[6] BRIER, E., CLAVIER, C., AND OLIVIER, F. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems* (2004), Springer, pp. 16–29.

[7] CHARI, S., RAO, J. R., AND ROHATGI, P. Template Attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2002), Springer, pp. 13–28.

[8] DEHBAOUI, A., DUTERTRE, J.-M., ROBISSON, B., AND TRIA, A. Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography* (2012), IEEE, pp. 7–15.

[9] ESPRESSIF. AES.c. `https://github.com/pycom/esp-idf-2.0/blob/master/components/esp32/hwcrypto/aes.c`.

[10] ESPRESSIF. ESP-IDF. `https://github.com/espressif/esp-idf`.

[11] ESPRESSIF. ESP32 Architecture. `http://esp32.net/#Features/`.

[12] ESPRESSIF. ESP32 Fault Injection Vulnerability - Impact Analysis. `https://www.espressif.com/en/news/ESP32_FIA_Analysis/`.

[13] ESPRESSIF. ESP32 Technical Reference Manual. `https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf`.

[14] ESPRESSIF. Flash Encryption. `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/flash-encryption.html`.

[15] ESPTOOL. Flash Encryption. `https://github.com/espressif/esptool/blob/master/espsecure.py/`.

[16] GILBERT GOODWILL, B. J., JAFFE, J., ROHATGI, P., ET AL. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop* (2011), vol. 7, pp. 115–136.

[17] HERIVEAUX, O. Scaffold. `https://github.com/Ledger-Donjon/scaffold?files=1/`.

[18] KOCHER, P., JAFFE, J., AND JUN, B. Differential power analysis. In *Annual international cryptology conference* (1999), Springer, pp. 388–397.

[19] KOCHER, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference* (1996), Springer, pp. 104–113.

[20] LIMITEDRESULTS. Pwn the ESP32 Forever: Flash Encryption and Sec. Boot Keys Extraction. https://limitedresults.com/2019/11/.

[21] O'FLYNN, C. Fault Injection using Crowbars on Embedded Systems. *IACR Cryptol. ePrint Arch. 2016* (2016), 810.

[22] RAELIZE. Pwn the ESP32 Forever: Flash Encryption and Sec. Boot Keys Extraction. https://raelize.com/blog/espressif-systems-esp32-bypassing-sb-using-emfi/.

[23] SKOROBOGATOV, S. P., AND ANDERSON, R. J. Optical Fault Induction Attacks. In *International workshop on cryptographic hardware and embedded systems* (2002), Springer, pp. 2–12.

[24] VEYRAT-CHARVILLON, N., GÉRARD, B., RENAULD, M., AND STANDAERT, F.-X. An optimal key enumeration algorithm and its application to side-channel attacks. In *International Conference on Selected Areas in Cryptography* (2012), Springer, pp. 390–406.