Breaking Espressif's ESP32 V3: Program Counter Control with Computed Values using Fault Injection

Jeroen Delvaux, *Technology Innovation Institute;* Cristofaro Mune, *Raelize;*
Mario Romero, *Technology Innovation Institute;* Niek Timmers, *Raelize*

https://www.usenix.org/conference/woot24/presentation/delvaux

This paper is included in the Proceedings of the
18th USENIX WOOT Conference on Offensive Technologies.

August 12–13, 2024 • Philadelphia, PA, USA

ISBN 978-1-939133-43-4

Open access to the
Proceedings of the 18th USENIX WOOT
Conference on Offensive Technologies
is sponsored by USENIX.

Breaking Espressif's ESP32 V3: Program Counter Control with Computed Values using Fault Injection

Jeroen Delvaux[1], Cristofaro Mune[2], Mario Romero[1], Niek Timmers[2] *
[1] *{Jeroen.Delvaux, Mario.Romero}@tii.ae, Technology Innovation Institute, Abu Dhabi, UAE*
[2] *{cristofaro, niek}@raelize.com, Raelize, Rotterdam, The Netherlands*

Abstract

Espressif introduced the ESP32 V3, a low-cost System-on-Chip (SoC) with wireless connectivity, as a response to earlier hardware revisions that were susceptible to Fault Injection (FI) attacks. Despite its FI countermeasures, we are the first to bypass all security features of the ESP32 V3 with an FI attack, including Secure Boot and Flash Encryption. First, we alter encrypted flash contents to set the 32-bit outcome of a Cyclic Redundancy Check (CRC) on the bootloader signature to an arbitrary value, which we then load into the Program Counter (PC) register of the Central Processing Unit (CPU) using a single Electromagnetic (EM) glitch. This allows us to jump to *Download Mode* in Read-Only Memory (ROM), which provides arbitrary code execution and access to unencrypted flash contents. As far as we know, this is the first successful FI attack, bypassing both Secure Boot and Flash Encryption with a single glitch, on a target with FI countermeasures. As the vulnerabilities are in hardware, they cannot be fixed, and a new hardware revision would be required. In response to our findings, Espressif issued a Security Advisory, AR2023-005, and requested a Common Vulnerabilities and Exposures (CVE) identifier, CVE-2023-35818.

1 Introduction

Espressif's ESP32 is a low-end System-on-Chip (SoC) with Wi-Fi and Bluetooth connectivity, which sparked commercial use in millions of embedded devices. Notable security features such as Secure Boot and Flash Encryption are supported. As shown in Fig. 1, the Secure Boot implements a *chain of trust* where code stored in internal Read-Only Memory (ROM) authenticates bootloader code stored in external Flash. The latter, in turn, authenticates application code stored in Flash. A chain of trust is needed as the ROM is made by Espressif and the flash contents are made by customers of Espressif. Note that Flash is a Multi-Time Programmable (MTP) Non-Volatile Memory (NVM).

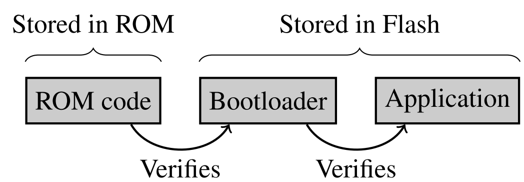


Figure 1: Chain of trust in a Secure Boot.

Vulnerabilities in the ROM code are particularly worrisome because (i) they compromise the entire chain, and (ii) they cannot be fixed by a software patch. The same holds for vulnerabilities that are purely in hardware. In this work, we attain this worst-case scenario of breaking the chain at the root. We leverage several weaknesses in the design of the ROM code through ElectroMagnetic Fault Injection (EMFI), which corrupts the executed instructions. Before listing our precise contributions, we situate our work into a brief history of FI attacks on the ESP32.

1.1 History of FI Attacks on ESP32

The first version of the chip, the ESP32-V1, was released in 2016, and its CPU implements the Xtensa Instruction Set Architecture (ISA) [32]. The following FI attacks were reported:

- In 2019, Riscure and LimitedResults [22] independently disclosed the first FI attack on the ESP32-V1: the digest verification of Secure Boot was skipped through a precisely timed voltage glitch (CVE-2019-15894) [12]. If Flash Encryption is disabled, this allows executing a modified bootloader.
- Still in 2019, LimitedResults [22] reported a second FI attack using supply-voltage glitching (CVE-2019-17391): bits stored in electronic fuses (eFuses), which is One-Time Programmable (OTP) NVM configured by Espressif's customers, are corrupted while being transferred

*The four authors contributed equally and are ordered alphabetically by last name.

to shadow registers. By corrupting read-protection bits stored in eFuses, keys that are also stored in eFuses, can be read out. In 2020, Raelize reproduced this attack using EMFI instead of voltage glitching [27].

- In 2020, Raelize reported an FI attack to bypass Secure Boot with Flash Encryption enabled, leveraging a peculiarity of the ROM to leave the Universal Asynchronous Receiver-Transmitter (UART) bootloader permanently enabled (CVE-2020-13629) [28]. For their attack, they leveraged retained data in the internal SRAM across warm resets, in order to control the PC register of the CPU.

In response to the above FI attacks, Espressif hardened the security design of the ESP32 and released ESP32 Chip Revision v3.0 in 2020 [7]. At the time of writing this paper, this is the latest revision. For the sake of brevity, we refer to this revision as ESP32 V3. Compared to the ESP32 V1, four significant changes are made:

- Secure Boot transitioned from symmetric-key cryptography, *i.e.*, the Advanced Encryption Standard (AES), to public-key cryptography, *i.e.*, Rivest–Shamir–Adleman (RSA) signatures. The ESP32 V3 only stores the public key; the private key is stored externally.
- While analyzing the ROM code, which was publicly released by Espressif as an ELF file [9], we identified the insertion of numerous redundancies, *e.g.*, eFuse bits are read out multiple times. Such redundancies are often used as FI countermeasures [2, 24, 35].
- The UART bootloader can now be disabled using a dedicated eFuse bit.
- Enabling Flash Encryption is encouraged as part of the newly introduced *Release Mode*. Stated otherwise, the security of a chip with Flash Encryption disabled is considered suboptimal.

To the best of our knowledge, Espressif has not made any statements about potential hardware countermeasures. Despite the above FI countermeasures, several FI attacks were reported on the ESP32 V3:

- In 2022, Ledger's Donjon [1] reported the first FI attack on the ESP32 V3, targeting a hardware accelerator of the Advanced Encryption Standard (AES) used for decrypting the Flash contents [14]. Through Body Biasing Injection (BBI), a fault analysis recovered the AES key. The same result was also achieved with a pure Side-Channel Attack (SCA): power-consumption traces were found to be correlated with Hamming distances between consecutive AES states. However, the authors were unsuccessful in retrieving the AES key with EM-FI, likely because of redundancies in the ROM code. More precisely, corrupting multiple OTP transfers with multiple EM pulses was found to be infeasible.
- In 2023, we were the second to report an FI attack on the ESP32 V3, albeit the first to succeed with EM-FI. The benefit is that EM-FI is less invasive than BBI, *i.e.*, the latter technique requires opening the plastic chip packaging so that a microprobe can reach the backside of the die [26]. The prime reason for our attack to succeed is that only a single EM pulse is required, *i.e.*, the complexity of jointly optimizing the glitch parameters of multiple pulses is avoided. Instead of AES and OTP transfers, we target ROM code running on the CPU, shortly before the RSA signature of the Flash contents is verified. This article describes this attack in more detail.

Several new releases of the ESP32 use RISC-V as ISA instead of Xtensa. In 2023, Courdesses [3] combined SCA and FI to achieve arbitrary code execution on two of these releases: the ESP32-C3 and the ESP32-C6 [15]. First, a power analysis recovered the AES key that encrypts the first 128-byte block of the Flash, which allows to insert arbitrary code into this block. Next, a voltage glitch bypasses Secure Boot such that the inserted code is executed. More precisely, the glitch causes a stack buffer to overflow, thereby overwriting a function return address with a pointer to the code.

1.2 Contributions

We present a novel FI attack against the ESP32 V3, which chains multiple vulnerabilities and uses a single EMFI glitch to access the decrypted flash contents. Our attack works on the most secure configuration and bypasses all countermeasures. Using commercially available tooling, our attack can be reproduced in minutes once effective glitch parameters such as timing and location are found.

By modifying the encrypted flash contents, we force the ROM's Cyclic Redundancy Check (CRC32) outcome to an arbitrary 32-bit value, which is then loaded into the CPU's Program Counter (PC) using an EM glitch. This way, we redirect the code execution to the ROM's Download Mode, which provides access to the decrypted flash contents. We are the first to load a computed value into the PC register of a CPU using a glitch. Moreover, as far as we know, this is the first example of a successful bypass of both Secure Boot and Flash Encryption using a single glitch, on a target with FI countermeasures.

1.3 Disclosure Timeline

The attack described in this paper was responsibly disclosed:

- A technical report specifying the attack was sent to Espressif on April 7, 2023.

- Espressif requested a Common Vulnerabilities and Exposures (CVE) identifier, which was created as CVE-2023-35818 on June 17, 2023.
- Espressif published Security Advisory AR2023-005 on its website on July 11, 2023 [16].
- Espressif transferred a bug bounty of USD 2229 on September 25, 2023.

1.4 Structure

The remainder of this paper is structured as follows. Section 2 provides preliminaries on the ESP32 V3. Section 3 provides the theory of our attack. Section 4 provides practical experiments. Section 5 concludes this work.

2 Preliminaries on Espressif's ESP32 V3

2.1 System Overview

As is shown in Fig. 2, the ESP32 V3 chip communicates with an external MTP NVM in the form of a Serial Peripheral Interface (SPI) Flash chip. This Flash chip stores the bootloader and the application, which can be signed and/or encrypted. The symmetric encryption key is stored in OTP NVM in the form of fuses. The public key for verifying signatures is stored in Flash, and to protect its integrity, a hash digest of the public key is stored in OTP NVM.

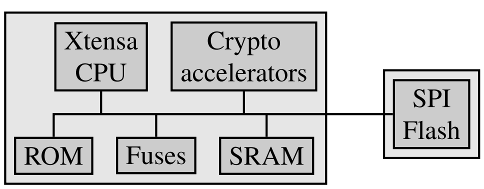


Figure 2: Relevant components of the ESP32 V3.

2.2 Xtensa Instruction Set Architecture

The CPU implements the Xtensa ISA [32]. Instructions are encoded in a 24-bit format, or if it concerns a common use case, in a so-called narrow (n) 16-bit format that can freely be intermixed with the 24-bit format. For example, the 24-bit *move* instruction `movi`, which sets a register to a 12-bit constant, has a 16-bit alternative `movi.n`, which sets a register to a 7-bit constant.

The ISA features 64 general-purpose registers of 32 bits each. However, only 16 registers are visible at any given time through a rotating window, and are labeled `a0` to `a15`. As illustrated in Fig. 3, the window moves back and forth with each function return and function call respectively. For any given subroutine, the return address is stored in register `a0`, the stack pointer is stored in register `a1`, and the input/output operands are stored in registers `a2` to `a7`. Hence, a caller that causes the window to shift with 8 registers, as is the case for the `call8` instruction, passes operands in registers `a10` to `a15` to physically match the subroutine.

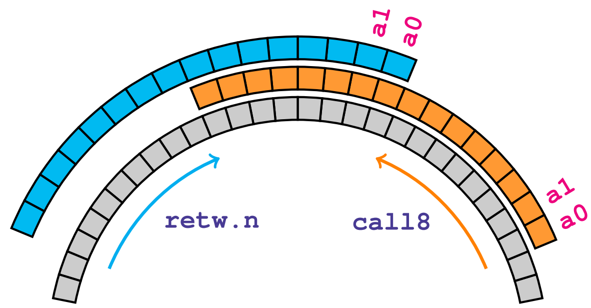


Figure 3: Xtensa rotating window, where the caller and the subprogram are colored orange and cyan respectively.

Given that the shift in window can be either 4, 8, or 12 registers, the two most significant bits of `a0` encode the shift, whereas the 29 least significant bits determine the return address.

2.3 Secure Boot V2

Once an ESP32-based product is fully developed and ready for commercial release, Espressif recommends configuring the chip in Release Mode. Consequentially, Secure Boot and Flash Encryption are both enabled. As illustrated in Fig. 4, the order of operations for constructing the Flash contents is sign-then-encrypt, not encrypt-then-sign. We make abstraction of the application in Fig. 1, given that a forgery of the bootloader inherently compromises the application. Below, the cryptographic algorithms for Secure Boot and Flash Encryption are specified.

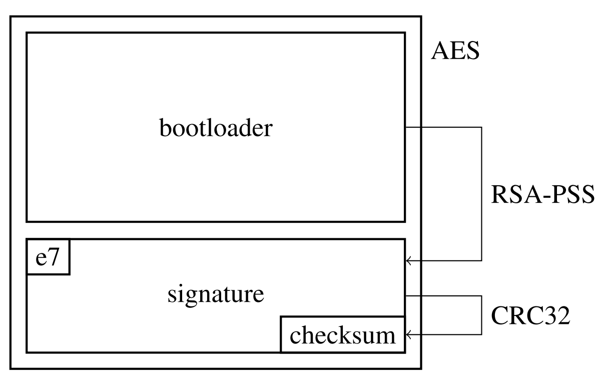


Figure 4: Signed and encrypted Flash data.

2.3.1 Secure Boot: Digital Signatures

Signatures are based on the RSA public-key algorithm, whilst adopting recommendations from Public-Key Cryptography Standards (PKCS) version 2.2, which is published as Request for Comments (RFC) 8017 [21]. More precisely, RSA-3072 is used, in which the public modulus and the signature are each 3072 bits, or 384 bytes. Instead of signing the bootloader image itself, the image is first fed into a Secure Hash Algorithm (SHA) and its digest is signed instead. More precisely, SHA-256 is used, which has a digest of 256 bits, or 32 bytes. The digest is encoded by the Probabilistic Signature Scheme (PSS).

As detailed in Table 1, the produced signature block contains 1216 bytes, starting with the magic byte `0xe7` and ending with a 32-bit checksum. The magic byte is aligned with a 4 KB boundary, *i.e.*, its physical address is an integer multiple of `0x4000`. The public key of RSA is part of the signature block and consists of a modulus, an exponent, and pre-calculated constants that accelerate verification. A SHA-256 digest of the public key is burned into eFuses. The CRC is computed over the 1196 preceding bytes.

Table 1: Signature block format [6].

Offset (bytes)	Size (bytes)	Description
0	1	Magic byte, `0xe7`
1	1	Version number, `0x02`
2	2	Zero padding, `0x0000`
4	32	SHA-256 digest of image
36	384	RSA public modulus
420	4	RSA public exponent
424	384	Pre-calculated constant
808	4	Pre-calculated constant
812	384	Signature
1196	4	CRC32
1200	16	Zero padding, `0x00...00`

As can be seen from the publicly released ROM code [9], verification at boot time consists of five consecutive checks. If any of them fails, an error message is printed via UART, and the device is reset. The first check compares the first byte of the signature block to the magic byte `0xe7`. The second check compares the recomputed CRC-32 checksum to its stored counterpart. The third check compares the recomputed SHA-256 digest of the public key to its counterpart stored in fuses. The fourth check compares the recomputed SHA-256 digest of the bootloader image to its stored counterpart. The fifth and last check is the verification of the RSA signature.

2.3.2 Flash Encryption

Flash encryption [5] relies on AES-256. The 256-bit key is stored in eFuses. Espressif adopted a custom *mode of operation* which is fully parallelizable, *i.e.*, consecutive 128-bit blocks can be encrypted independently of one another, and the same holds for the decryption. Note that this entails random access.

The key for each 128-bit block is derived by XORing the master key stored in eFuses with the physical address of the 256-bit block. Hence, each derived key encrypts two adjacent blocks. For performance reasons: Flash encryption, which is an infrequent operation, uses AES decryption, whereas Flash decryption, which happens on every boot, uses AES encryption.

3 Theory of the Attack

3.1 PC Control Through FI

Originally, PC control through FI was performed in absence of Flash encryption, which is an easier setting than ours. We first describe the original technique, and then introduce our workaround for tackling Flash encryption.

3.1.1 Without Flash Encryption

In 2016, Timmers *et al.* [34] described an FI attack that sets the PC register to a controlled value on CPUs implementing ARM's AArch32 execution state. These controlled values originate from a source that is under control of an attacker, *e.g.*, unencrypted flash. In ARM's AArch32 execution state, the PC register can be used as a destination register for many instructions, which was found to be ideal. Only a single load instruction needs to be corrupted by a glitch in order to load a controlled value directly into the PC register. An effective approach [33] is described below:

- Overwrite the original bootloader in flash with a code payload and sled of pointers. These pointers point to the destination address in executable memory at which the code payload is be copied to.
- As a result, when the device is powered, the ROM code will copy the code payload and the pointers to the same destination as the original bootloader. Then, assuming Secure Boot is enabled, the signature check would fail and the target is reset.
- For the attack, a glitch is injected after the code payload is copied, but while the pointers are being copied. This glitch modifies the destination operand of a load instruction such that a controlled value is loaded into the PC register. This effectively executes the code payload well-before the signature is verified.

The above approach is also possible on CPU architectures where the PC register is not directly addressable, including ARM's AArch64 and Xtensa. For these type of architectures, the PC register can only be controlled indirectly, e.g., by corrupting the operand of a branch, jump or return instruction.

3.1.2 With Flash Encryption

On modern SoC where Flash contents are encrypted, the technique in Section 3.1.1 might still work, on the condition that the CPU operates directly on ciphertext. Then, a controlled value can be loaded into the PC register simply by overwriting the ciphertext.

However, on the ESP32, the flash contents are decrypted on-the-fly by a hardware implementation of AES. This process is done completely transparent to the CPU, which never operates on the encrypted contents, only on the decrypted contents. Therefore, any modification in the external flash will end up in the context of the CPU as *gibberish*. In theory, a brute-force attack on the 32-bit address space might still be possible, *i.e.*, the ciphertext is randomly manipulated until the pointer of interest is found. In practice though, the time needed for performing this search is likely excessive, given that devices typically take a few milliseconds to boot.

Therefore, we decided to find another method for slipping in one or more controlled values, which we intend to load into the PC register using a glitch. On the ESP32 V1, Raelize [28] leveraged the UART bootloader, which could not be disabled. However, Espressif patched this vulnerability on the ESP32 V3. We decided to slip in a controlled 32-bit value into the context of the CPU by tampering with the CRC operation that is performed over the signature block. Setting the PC register of the CPU to the result of this CRC32 operation using a glitch is the main novelty of this paper. Gratchoff [19] previously described this as a potential approach, however, to the best of our knowledge, this has never been performed in practice.

3.2 Modifying the Flash

We demonstrate our attack on a bootloader which prints "Hello, World!". There is no application, as shown in Fig. 1, because being able to execute a modified bootloader compromises the application by default. The boot log observed on the UART is given in Fig. 5. Additional line breaks have been inserted to accommodate the two-column format of this paper.

The ROM code reports explicitly that secure boot is enabled and the secure boot verification succeeded. Even though not specifically reported, flash encryption is enabled as well. Any change to the bootloader or its signature block, both of which are stored encrypted in flash, causes an error message to be displayed in the boot log. If the signature block is modified, the checksum verification fails, and the error message in Fig. 6 is printed. The key observation is that that the checksum verification is done in the plaintext domain.

```
ets Jul 29 2019 12:21:46
rst:0x1 (POWERON_RESET),boot:0x13
(SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,
hd_drv:0x00,wp_drv:0x00
mode:2, clock div:2
secure boot v2 enabled
secure boot verification succeeded
load:0x3fff0020 len:0xc8c
load:0x40078000 len:0x2020
load:0x40080400 len:0xeac
entry 0x40080640
I (41) boot: ESP-IDF v5.0.1-397-g3050ea656f 2nd
stage bootloader
I (41) boot: compile time 16:51:07
I (41) boot: chip revision: v3.0
I (45) boot.esp32: SPI Speed      : 40MHz
I (50) boot.esp32: SPI Mode       : DIO
I (54) boot.esp32: SPI Flash Size : 2MB
I (59) boot: Enabling RNG early entropy source...
Hello, World!
```

Figure 5: UART for a bootloader that prints "Hello, World!".

```
...
secure boot v2 enabled
Sig block 0 invalid: Stored CRC 0xbaaeaf78
calculated 0xdeadbeef
secure boot verification failed
```

Figure 6: UART boot log where the CRC fails.

We refrain from corrupting the first 16-byte block of the signature as this includes a byte at offset 0 which is used as a magic value. Whenever this value is not **`0xe7`**, the signature block is not considered a signature block and the checksum verification is not performed. The error message that is printed when the magic value is modified is shown in Fig. 7.

```
...
secure boot v2 enabled
No signature block magic byte found at signature
sector (found 0xc3 not 0xe7). Image not V2 signed?
secure boot verification failed
```

Figure 7: UART boot log where the magic byte is corrupted.

By performing manipulations of the ciphertext, we can solve a system of linear equations and set the recomputed checksum to any 32-bit value of choice. In fact, the hexspeak value **`0xdeadbeef`** in Fig. 6 is no coincidence, and serves to demonstrate this ability. For our attack, we modify this hexpspeak value into a pointer, *i.e.*, a memory address, which we then load into PC using a glitch.

3.3 Solving Equations

We now specify how the system of linear equations is constructed. As this section is mathematical, unlike the rest of this paper, a notation system is introduced. Variables and constants are denoted by characters from the Latin and Greek alphabets respectively. Orthogonal to this convention: scalars are denoted by regular lowercase characters, binary vectors are denoted by bold lowercase characters, and binary matrices are denoted by bold uppercase characters. All vectors are column vectors.

We leverage that CRC-32 is an affine function, as formalized in Eq. (1), where $\oplus$ denotes XORing and where constant $\boldsymbol{\gamma} \in \{0,1\}^{32}$ only depends on the size of the input $\boldsymbol{x}$. For inputs $\boldsymbol{x} \in \{0,1\}^{9568}$, which corresponds to the first 1196 bytes of the signature block, it holds that $\boldsymbol{\gamma} =$ `0x6b691bc6`. Given that $\boldsymbol{\gamma}$, eventually, cancels out, its value is inconsequential.

$$\text{CRC-32}(\boldsymbol{x}_1 \oplus \boldsymbol{x}_2) = \text{CRC-32}(\boldsymbol{x}_1) \oplus \text{CRC-32}(\boldsymbol{x}_2) \oplus \boldsymbol{\gamma}. \quad (1)$$

Input $\boldsymbol{x} \in \{0,1\}^{9568}$ spans 75 AES plaintext blocks $\boldsymbol{p} \in \{0,1\}^{128}$, as formalized in Eq. (2) where $\|$ is the concatenation operator. The first block, $\boldsymbol{p}_0$, contains the magic byte `0xe7`. The last block, $\boldsymbol{p}_{74}$, shares only 96 bits with $\boldsymbol{x}$, and the 32 excluded bits comprise the stored checksum $\boldsymbol{s}_{\text{stored}} = \text{CRC-32}(\boldsymbol{x})$.

$$\boldsymbol{x} \triangleq \boldsymbol{p}_0 \,\|\, \boldsymbol{p}_1 \,\|\, \cdots \,\|\, \boldsymbol{p}_{73} \,\|\, (\boldsymbol{p}_{74} \bmod 2^{96}). \quad (2)$$

The external Flash contains the corresponding ciphertexts $\boldsymbol{c}_0, \boldsymbol{c}_1, \cdots, \boldsymbol{c}_{74}$, which we can alter. The first block, $\boldsymbol{c}_0$, is unaltered. Otherwise, the magic byte is not found with probability $255/256$, and the UART boot log is uninformative, as shown in Fig. 7.

Instead, we consecutively alter blocks $\boldsymbol{c}_1$ to $\boldsymbol{c}_{32}$. In the first iteration, we overwrite $\boldsymbol{c}_1$ with a value $\boldsymbol{c}_1^\star$ that is selected uniformly at random from $\{0,1\}^{128}$. Consequentially, the corresponding plaintext block $\boldsymbol{p}_1$ changes to an unknown value $\boldsymbol{p}_1^\star \triangleq \boldsymbol{p}_1 \oplus \boldsymbol{e}_1$. By booting the ESP32 with this modification, and parsing the UART log, we obtain the checksum difference $\boldsymbol{d}_1 \triangleq \boldsymbol{s}_{\text{calculated},1} \oplus \boldsymbol{s}_{\text{stored}}$. From linearity in Eq. (1), it follows that the difference $\boldsymbol{d}_1$ only depends on plaintext error $\boldsymbol{e}_1$, as specified in Eq. (3). Nevertheless, $\boldsymbol{e}_1 \in \{0,1\}^{128}$ cannot be recovered from $\boldsymbol{d}_1 \in \{0,1\}^{32}$ due to the 96-bit difference in length, *i.e.*, there are many $\boldsymbol{e}_1$'s that result in the same $\boldsymbol{d}_1$. This is fine: $\boldsymbol{e}_1$ does not need to be recovered, and we merely store the pair $(\boldsymbol{c}_1^\star, \boldsymbol{d}_1)$ for further use.

$$\boldsymbol{d}_1 = \text{CRC-32}(0_{128} \,\|\, \boldsymbol{e}_1 \,\|\, 0_{9312}) \oplus \boldsymbol{\gamma}. \quad (3)$$

Now, the same principle is repeated to obtain pairs $(\boldsymbol{c}_2^\star, \boldsymbol{d}_2)$ until $(\boldsymbol{c}_{32}^\star, \boldsymbol{d}_{32})$, as formalized in Eq. (4). Again, recovery of $\boldsymbol{e}_2$ until $\boldsymbol{e}_{32}$ is unnecessary.

$$\begin{aligned} \boldsymbol{d}_2 &= \text{CRC-32}(0_{256} \,\|\, \boldsymbol{e}_2 \,\|\, 0_{9184}) \oplus \boldsymbol{\gamma}. \\ &\vdots \\ \boldsymbol{d}_{32} &= \text{CRC-32}(0_{4096} \,\|\, \boldsymbol{e}_{32} \,\|\, 0_{5344}) \oplus \boldsymbol{\gamma}. \end{aligned} \quad (4)$$

Instead, we linearly combine the known differences $\boldsymbol{d}_1$ until $\boldsymbol{d}_{32}$ into a desired difference $\boldsymbol{d} \triangleq \boldsymbol{s}_{\text{pointer}} \oplus \boldsymbol{s}_{\text{stored}}$, where $\boldsymbol{s}_{\text{pointer}}$ is the memory address we want to jump to. This is achieved by solving the system of linear equations in Eq. (5) for $\boldsymbol{z} \in \{0,1\}^{32}$. Note the absence of constant $\boldsymbol{\gamma}$. Each bit z_i of $\boldsymbol{z}$, where $i \in [1,32]$, determines whether or not the corresponding ciphertext block should be corrupted: if $z_i = 0$, the original cipertext is $\boldsymbol{c}_i$ remains in place, otherwise, the random value $\boldsymbol{c}_i^\star$ is used.

$$\boldsymbol{D}\boldsymbol{z} = \boldsymbol{d}, \quad \text{where } \boldsymbol{D} = \begin{pmatrix} \boldsymbol{d}_1 & \boldsymbol{d}_2 & \cdots & \boldsymbol{d}_{32} \end{pmatrix}. \quad (5)$$

One problem remains though: the matrix $\boldsymbol{D} \in \{0,1\}^{32\times 32}$ is not necessarily invertible. Under the assumption that $\boldsymbol{D}$ is selected uniformly at random from $\{0,1\}^{32\times 32}$, which is a reasonable abstraction, the probability that $\boldsymbol{D}$ is invertible given in Eq. (6). The proof is straightforward and imagines that columns are added one-by-one [25]: if the previous $i-1$ columns are linearly independent, the addition of column i causes linear dependence with probability 2^{i-33}. For example, the first and last columns cause linear dependence with probability $1/2^{32}$ and $1/2$ respectively. Logarithms help with numerical evaluation, and result in a probability of around 28%.

$$\begin{aligned} \Pr(\text{rank}(\boldsymbol{D}) = 32) &= \prod_{i=1}^{32} (1 - 2^{-i}) \\ &= \exp\left(\sum_{i=1}^{32} \left(\log(2^i - 1) - \log(2^i) \right) \right) \approx 28\%. \end{aligned} \quad (6)$$

To ensure that $\boldsymbol{D}$ is invertible, we check whether its rank increases for each column $\boldsymbol{d}_i$ that is added, as formalized in Algorithm 1. If the rank does not increase, a new corrupted ciphertext $\boldsymbol{c}_i^\star$ is selected uniformly at random. As can be seen from the invertibility proof [25], it are usually the last few columns that require a retry. Observe that there is no need to retake measurements if we would want to build images for more than one pointer of interest.

Alternatives to Algorithm 1 could be devised. For example, instead of gathering pairs $(\boldsymbol{c}_i^\star, \boldsymbol{d}_i)$ for 32 AES blocks, pairs could be gathered for, say, 40, blocks. From these 40 blocks, 32 blocks that result in an invertible $\boldsymbol{D}$ are then retained.

Although solving a system of equations is the canonical approach, it is only possible because the signature block happens to be long. Originally, we disclosed an alternative method to Espressif that would also have worked for small signature blocks, at the minor inconvenience of a 32-bit brute-force

Algorithm 1: Measurement for CRC insertion

Input: Original bootloader, $\boldsymbol{b} \in \{0,1\}^*$
Input: Index of magic byte, $m \in \mathbb{N}$
Input: Pointer of interest, $\boldsymbol{s}_{\text{pointer}} \in \{0,1\}^{32}$
Output: Modified bootloader, $\boldsymbol{b}^\star \in \{0,1\}^*$

$\boldsymbol{C} \leftarrow \boldsymbol{0}_{32\times128}$
$\boldsymbol{D} \leftarrow \boldsymbol{0}_{32\times32}$
for $i \leftarrow 1$ **to** 32 **do**
  $\boldsymbol{b}^\star \leftarrow \boldsymbol{b}$
  do
    $\boldsymbol{c}_i \leftarrow \{0,1\}^{128}$
    $\boldsymbol{b}^\star[m+i128 : m+i128+127] \leftarrow \boldsymbol{c}_i$
    Program $\boldsymbol{b}^\star$
    Fetch $\boldsymbol{s}_{\text{calculated}}$ and $\boldsymbol{s}_{\text{stored}}$ from UART
    $\boldsymbol{D}[:,i] \leftarrow \boldsymbol{s}_{\text{calculated}} \oplus \boldsymbol{s}_{\text{stored}}$
  while rank$(\boldsymbol{D}) \neq i$
  $\boldsymbol{C}[:,i] \leftarrow \boldsymbol{c}_i$
$\boldsymbol{b}^\star \leftarrow \boldsymbol{b}$
$\boldsymbol{z} \leftarrow \boldsymbol{D}^{-1}(\boldsymbol{s}_{\text{pointer}} \oplus \boldsymbol{s}_{\text{stored}})$
for $i \leftarrow 1$ **to** 32 **do**
  if $z[i]$ **then**
    $\boldsymbol{b}^\star[m+i128 : m+i128+127] \leftarrow \boldsymbol{C}[:,i]$

search. In this method, we perturb $\eta \geq 4$ blocks of the signature, and for each block, we select $\lambda \geq 2$ ciphertexts $\boldsymbol{c}^\star$ uniformly at random from $\{0,1\}^{128}$, where $\lambda^\eta > 2^{32}$. We store pairs $(\boldsymbol{c}^\star_{i,j}, \boldsymbol{d}_{i,j})$, where $i \in [1,\eta]$ and $j \in [1,\lambda]$, and where $\boldsymbol{d}_{i,j}$ is given in Eq. (7).

$$\boldsymbol{d}_{i,j} = \text{CRC-32}(0_{128\cdot i} \,\|\, \boldsymbol{e}_{i,j} \,\|\, 0_{9440-128\cdot i}) \oplus \boldsymbol{\gamma}. \tag{7}$$

Next, the goal is to find indices $j_1, j_2, \cdots j_\eta \in [1,\lambda]$ such that $\boldsymbol{d}_{1,j_1} \oplus \boldsymbol{d}_{2,j_2} \oplus \cdots \oplus \boldsymbol{d}_{\eta,j_\eta} = \boldsymbol{d}$. This search took less than one hour on a laptop. The corresponding ciphertexts $\boldsymbol{c}^\star_{1,j_1}, \boldsymbol{c}^\star_{2,j_2}, \cdots, \boldsymbol{c}^\star_{\eta,j_\eta}$ are applied.

3.4 Attack Surface for FI

The result of the checksum operation, *i.e.*, the pointer of interest, propagates through several CPU registers before the chip, eventually, resets. This propagation path can be followed with relative ease, given that Espressif published the ROM code in ELF format [9]. If the ROM code would not have been published, the code would have to be extracted from the device through either delayering [17, 18] or an exploit [4, 31]. The ELF file is loaded in Ghidra, which is reverse-engineering software that decompiles assembly instructions into C, among other features. Our analysis reveals that the proverbial *attack surface* for FI comprises three subroutines.

The first subroutine is `crc32_le`, which computes the checksum, and is shown in Fig. 8. The XOR operation at address `0x4005d019` writes the computed checksum into register `a2`. If this instruction could be corrupted such that the destination register `a2` changes to the return address `a0`, as formalized in Corruption 1, the pointer of interest would be loaded into the PC register of the CPU. Given that `a2` and `a0` are encoded as four-bit fields `0x2` and `0x0` respectively, this would only require a single bit flip.

```
crc32_le()
0x4005cfec  entry    a1, 0x20
0x4005cfef  movi.n   a8, 0xff
0x4005cff1  xor      a2, a8, a2
0x4005cff4  l32r     a9, 0x4005cfe8
0x4005cff7  movi.n   a8, 0x0
0x4005cff9  j        0x4005d014
0x4005cffc  add.n    a10, a3, a8
0x4005cffe  l8ui     a10, a10, 0x0
0x4005d001  addi.n   a8, a8, 0x1
0x4005d003  xor      a10, a10, a2
0x4005d006  extui    a10, a10, 0x0, 0x8
0x4005d009  addx4    a10, a10, a9
0x4005d00c  l32i.n   a10, a10, 0x0
0x4005d00e  srli     a2, a2, 0x8
0x4005d011  xor      a2, a10, a2
0x4005d014  bne      a8, a4, 0x4005cffc
0x4005d017  movi.n   a3, 0xff
0x4005d019  xor      a2, a3, a2
0x4005d01c  retw.n
```

Figure 8: ROM code of `crc32_le`.

Corruption 1. *At address* `0x4005d019`*, the instruction* `xor a2, a3, a2` *with encoding* `0x302320` *is corrupted into* `xor a0, a3, a2` *with encoding* `0x300320`*.*

The second potential target for FI is subroutine `ets_secure_boot_verify_signature`, which is the caller of `crc32_le`, and the relevant part is shown in Fig. 9. Due to the shifting window, the pointer is returned in register `a10` after the `call8` instruction at address `0x4006547e`, and is copied to register `a13` at address `0x40065485`. If the contents of `a10` differ from the stored checksum in register `a12`, a branch is taken at address `0x40065488` to resume normal operation. Otherwise, the subroutine `ets_printf` is called at address `0x40065491` to print the CRC error message. The unconditional jump at address `0x40065565` results in a reset.

Again, taking PC control by overwriting the return address `a0` is plausible. Most notably, the move instruction at address `0x40065485` could be corrupted such that the destination register changes from `a13` to `a0`, as formalized in Corruption 2. Although this entails three bit flips, the probability of

```
0x40065474 movi    a12, 0x4ac
0x40065477 movi.n  a10, 0x0
0x40065479 mov.n   a11, a6
0x4006547b movi    a2, 0x4ac
0x4006547e call8   crc32_le
0x40065481 add.n   a2, a6, a2
0x40065483 l32i.n  a12, a2, 0x0
0x40065485 mov     a13, a10
0x40065488 beq     a10, a12, 0x40065498
0x4006548b l32r    a10, 0x40065428
0x4006548e mov     a11, a7
0x40065491 call8   ets_printf
0x40065494 j       0x40065565
```

Figure 9: ROM-code fragment of `ets_secure_boot_verify_signature`.

occurrence could be significant depending on the unknown *fault model*: all flips are of the type $1 \rightarrow 0$, and they all occur within a single 4-bit field. Different fields are processed by different circuits, so there is no reason why setting an entire field to zero would be unrealistic.

Corruption 2. *At address* `0x40065485`*, the instruction* `mov a13, a10` *with encoding* `0x20daa0` *is corrupted into* `mov a0, a10` *with encoding* `0x200aa0`*.*

Alternatively, it might be possible to corrupt the opcode of the move instruction and turn it into an unconditional register jump `jx`, as formalized in Corruption 3. Although this entails four bit flips, it equates to setting two out of six fields to zero.

Corruption 3. *At address* `0x40065485`*, the instruction* `mov a13, a10` *with encoding* `0x20daa0` *is corrupted into* `jx a10` *with encoding* `0x000aa0`*.*

The third and last subroutine for potential FI is `ets_printf`, which prints a formatted string similar to its C counterpart *printf*. The pointer of interest is passed as an argument through register `a13`. The ROM code is not analyzed here due to its length.

3.5 Pointers of Interest

As listed in Table 2, we jump to two ROM functions. The first function, `ets_fatal_exception_handler`, prepares a formatted string and calls `ets_printf`, as shown in Fig. 10. The relative simplicity of a print enables us to efficiently tune EM-FI glitch parameters later-on: the delay, the power, and the XY coordinates. Furthermore, because the value of five registers is printed, useful insights about the injected fault can potentially be gained.

Once suitable glitch parameter values are found, we change the Flash image of our target device and jump to *Download Mode* instead, *i.e.*, the ROM function `UartDwnLdProc`. The latter jump is more restrictive than `ets_fatal_exception_handler` because three input parameters should have proper values.

Given that the windowing mechanism of the Xtensa ISA has a crucial role in this matter, we experiment with addresses of the form `0x4XXXXXXX` and `0x8XXXXXXX`. The return instruction `retw.n` uses the two most significant bits of `a0` are to determine the shift in window, whereas the 29 least significant bits determine the next PC.

Table 2: Pointers of interest in the ROM code.

Address	Function
`0x40006864` `0x80006864`	`ets_fatal_exception_handler`
`0x40008ceb` `0x80008ceb`	`UartDwnLdProc`

```
0x40006864 l32r    a10, 0x3ff9e820
0x40006867 mov.n   a11, a6
0x40006869 mov.n   a12, a5
0x4000686b mov.n   a13, a4
0x4000686d mov.n   a14, a3
0x4000686f mov.n   a15, a2
0x40006871 call8   ets_printf
```

Figure 10: ROM-code fragment of `ets_fatal_exception_handler`.

3.6 Simulating Faults with GDB

Before building the FI setup and performing the attack in practice, we simulated the desired faults with Espressif's GNU Debugger (GDB) to confirm their effect. For this purpose, we prepared a bootloader where the signature block is corrupted, and the recomputed checksum is, consequentially, incorrect. Upon flashing this bootloader, Corruption 1 and Corruption 2 are simulated as shown in Fig. 11a and Fig. 11b respectively. In both simulations, we set a *hardware breakpoint* at the targeted instruction and overwrite register `a0` with the desired pointer during the break.

To determine whether or not `ets_fatal_exception_handler` is reached, we merely need to observe the UART output, and check whether or not the string is printed. For Download Mode, there is no welcome message, but we can check whether or not an additional hardware breakpoint deep within this mode is reached. Our conclusion is that pointers of the form `0x8XXXXXXX` result in a successful jump, whereas pointers of the form `0x4XXXXXXX` do not.

For Corruption 3, we performed similar GDB experiments.

```
hbreak *0x4005d01c
continue
set $a0 = 0x80006864
continue
```

(a) Corruption 1

```
hbreak *0x40065485
continue
set $a0 = 0x80006864
continue
```

(b) Corruption 2

Figure 11: Simulation of (a) Corruption 1 in **crc32_le** and (b) Corruption 2 in **ets_secure_boot_verify_signature** with GDB.

However, the conclusion is different: pointers of the forms **0x4XXXXXXX** and **0x8XXXXXXX** both result in a successful jump.

4 Practical Experiments

4.1 Target Preparation

We target an ESP32-DevKitC V4 [8] with an ESP32-WROOM-32E module [13], which is a small-sized and commercially available development board produced by Espressif. To enable EM-FI, we removed the metal shield that covers both the ESP32 V3 chip and the SPI flash chip with a KADA 852D+ hot air gun. No-clean flux is applied to facilitate this process.

Upon confirming that the board survived the hot air, we manually enable the security features of the ESP32 V3 by burning eFuses. Although Espressif provides a partially automated process, the manual approach is more convenient for developing an attack: the security features can be enabled one by one, instead of altogether automatically. Figure 12a shows the eFuses for enabling Secure Boot. The SHA-256 digest of the RSA public key is obtained from the file **rsa.pem**. Figure 12b shows the eFuses for enabling Flash encryption. The 256-bit AES key is contained in the binary file **aes.bin**. Figure 12c shows the eFuses for enabling Release Mode.

Burning the eFuses for enabling Flash Encryption, as shown in Fig. 12b, is postponed as long as possible. Although our attack works equally well with and without Flash Encryption, this allows us to gradually develop the attack and compare timing in the two cases.

Likewise, burning the eFuse for disabling Download Mode, as shown in Fig. 12d, is postponed as long as possible. Although this security feature does not preclude our attack, in which we enter Download Mode by directly jumping to address **0x40008ceb**, one cannot easily program the external Flash anymore after the eFuse is burned. Recall from Algorithm 1 that at least 33 manipulated Flash images need to be programmed to set the recomputed checksum to an arbitrary pointer. Starting from a valid signed and encrypted image, where the bootloader prints "Hello, World!", we created four images that correspond to the four interesting jump locations in Table 2. Only after tuning the glitch parameters, we burn

```
$ espefuse.py burn_key_digest rsa.pem
$ espefuse.py burn_efuse ABS_DONE_1 1
```

(a) Secure Boot.

```
$ espefuse.py burn_key flash_encryption aes.bin
$ espefuse.py burn_efuse FLASH_CRYPT_CNT 1
$ espefuse.py burn_efuse FLASH_CRYPT_CONFIG 15
```

(b) Flash Encryption.

```
$ espefuse.py burn_efuse DISABLE_DL_ENCRYPT 1
$ espefuse.py burn_efuse DISABLE_DL_DECRYPT 1
$ espefuse.py burn_efuse DISABLE_DL_CACHE 1
$ espefuse.py write_protect_efuse FLASH_CRYPT_CNT
```

(c) Release Mode.

```
$ espefuse.py burn_efuse UART_DOWNLOAD_DIS 1
```

(d) Download Mode.

Figure 12: Burning eFuses for (a) enabling Secure Boot, (b) enabling Flash Encryption, (c) enabling Release Mode, and (d) disabling Download Mode.

the eFuse. Because the Flash is external, programming in principle remains possible, at the minor inconvenience of soldering an SPI programmer to the chip.

4.2 EM-FI Setup

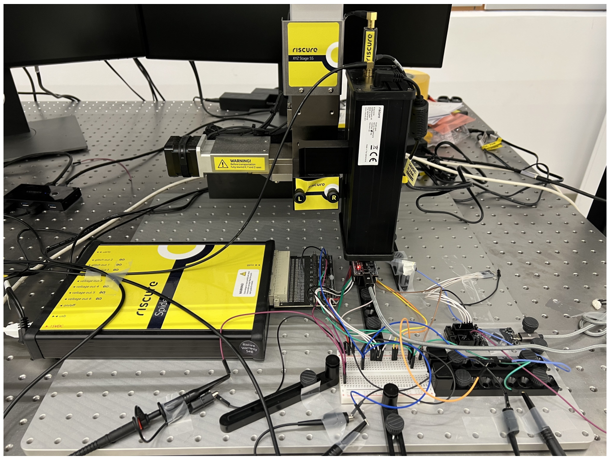
We used Riscure's EM-FI setup [29]. The motorized XYZ stage is shown in Fig. 13a. We used the large red Classic probe tip, which has a diameter of 4 mm. The targeted ESP32 V3 board is stabilized with double-sided tape. A desktop computer communicates with the board via the Micro-USB connector; a Windows COM port provides the serial interface. As is shown in Fig. 13b, an electric wire is attached to the *chip-enable* pin from the SPI Flash chip, thereby providing a timing reference (*i.e.*trigger) for the EM glitches.

4.3 Tuning Glitch Parameters

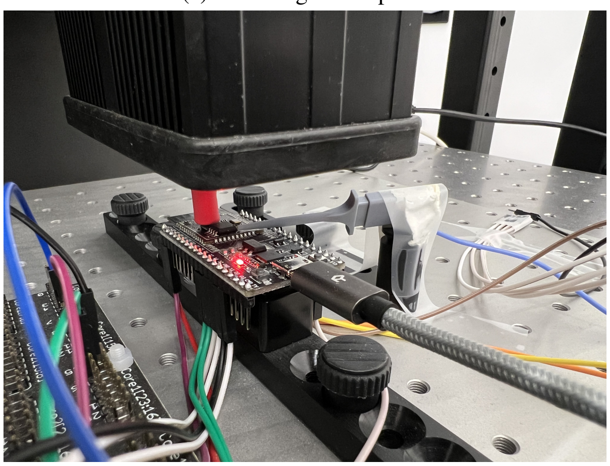
4.3.1 Coarse Timing: Execution Trace

The most crucial glitch parameter to be tuned is the timing. As in prior work, we use the so-called *chip-enable* signal of the Flash chip as a timing reference. As shown in Fig. 14, the *chip-enable* signal is observed to consist of five relatively large blocks where data is copied from Flash to SRAM. We used a Teledyne LeCroy WavePro 804HD oscilloscope to take these measurements.

Next, we determine when **crc32_le** is executed with respect to these five copy blocks. For this purpose, we use Espressif's GDB to trace the program execution. We created a



(a) XYZ stage and Spider



(b) Target and glitch amplifier

Figure 13: Riscure EM FI setup.

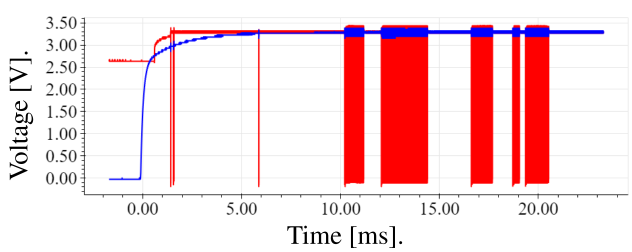


Figure 14: Blocks of data copied from Flash to SRAM. The reset signal is colored blue; the chip-enable signal is colored red.

Python script that starts from a hardware breakpoint in `main`, and then pauses at each instruction with `stepi` until the program ends. At each pause, we log the function name, the value of the program counter, and the value of registers `a0` to `a15`. This whole process takes less than two hours. As shown in Fig. 15, `crc32_le` is executed shortly after block #5.

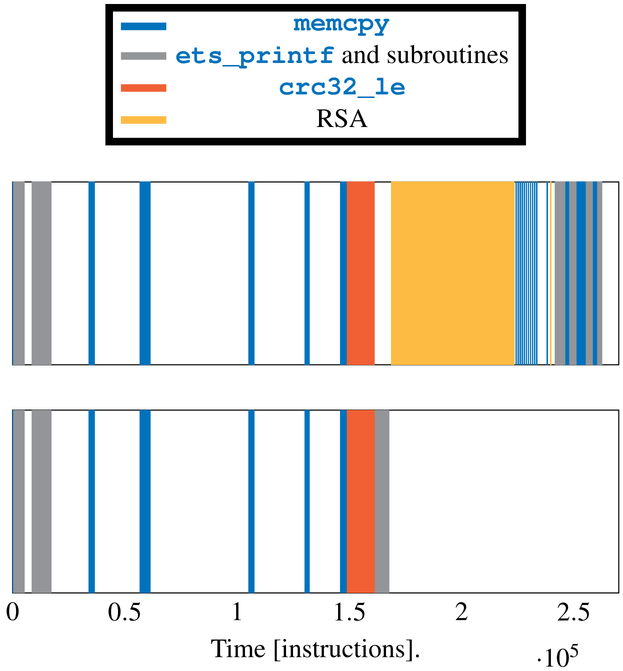


Figure 15: Execution trace when the CRC is correct (top) and wrong (bottom).

Remark that execution traces are only one possible method to obtain course timing information. An unexplored alternative is SCA, *e.g.*, by taking power-consumption measurements. In this approach, power traces are collected for the two classes, where the recomputed checksum is correct and wrong for the first and second class respectively. Initially, the two classes should be quasi indistinguishable, and shortly after `crc32_le`, the classes should diverge drastically.

4.3.2 Refined Timing: FI as Virtual Oscilloscope

Next, we refine the timing by using EM-FI as a virtual oscilloscope [20, 23, 30]. We inject glitches in a large time interval while fetching the CRC error string from UART, both with and without Flash Encryption. Because only tiny differences could be observed, we only show results obtained with Flash Encryption enabled in Fig. 16 and all subsequent scatter plots.

For Fig. 16 and all subsequent plots, we adopt the color legend from Table 3. Green dots represent the baseline, *i.e.*, the fault has no observable effect in the UART output, and the device eventually resets because the stored and recomputed checksums are different. Yellow dots represent fault-induced crashes, *i.e.*, the fault and not the checksum difference causes the target device to reset. Cyan dots indicate that the CRC error string is deformed, e.g., characters are missing or corrupted. Orange dots indicate that the CRC error string is well-formed, but the recomputed checksum is corrupted. Purple dots indicate that the stored checksum is corrupted instead. Pink dots indicate that the recomputed and stored checksums

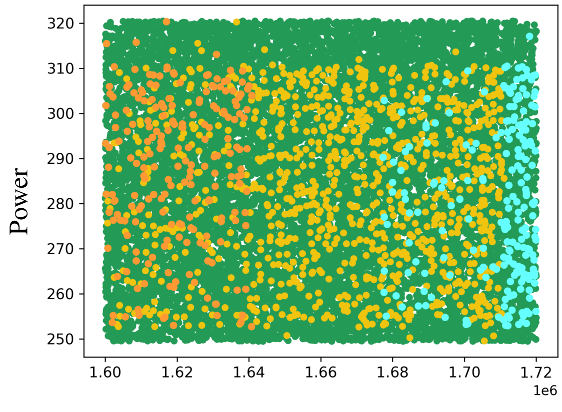


Figure 16: FI as an oscilloscope.

are both corrupted. Red dots indicate successful PC control.

Table 3: Color legend for the scatter plots in Figs. 16 to 19.

green	Nominal response.
yellow	No response, *i.e.*, a crash.
cyan	Deformed CRC error string.
orange	Altered recomputed checksum.
purple	Altered stored checksum.
pink	Altered recomputed and stored checksums.
red	Successful jump.

For improved visibility, three measures are taken for all scatter plots. Firstly, small random errors are added to the shown pair of variables, which is also a common practice in Riscure's own visualization software, Spotlight. Otherwise, most dots would coincide. Secondly, dots are drawn in the order of the legend in Table 3. Otherwise, a small number of red dots could be obscured by large numbers of green and yellow dots, for example. Thirdly, dots with different colors might be drawn with different diameters.

In Fig. 16, two regions are of particular interest. The region with the orange dots corresponds to `crc32_le`. The region with cyan dots is part of `ets_printf`. From the ROM code execution trace analysis, we know that `ets_printf` starts well before the cyan dots appear. The function `ets_secure_boot_verify_signature` is likely executed in the region between the orange and the cyan dots. This is not visible because of the very small number of instructions the function is composed of.

4.3.3 XY-Coordinates and Power

The XYZ-stage is used to scan the surface of the ESP32 chip. Although the surface is approximately square, the EM-FI probe is partially blocked by the neighboring Flash chip and is free to move in a rectangular area of roughly 5 mm × 2 mm. Because the Flash chip has only eight pins, displacement though soldering is possible, but is unnecessary for the attack to succeed. Within the rectangular area, the probe moves in a 30-by-30 grid.

Fig. 17 shows the result of our surface scan. For clarity, only the green, yellow, and red dots are shown. Red dots represent successes, *i.e.*, the string in `ets_fatal_exception_handler` is successfully printed. For this particular scan, we used the `0x8XXXXXXX` address. The key takeaway of Fig. 17 is that the probe can be placed inside at a relatively large fraction of the chip's surface in order to inject a successful glitch. Stated otherwise, finding the proverbial *needle in a haystack* primarily applies to time, not space. This is unsurprising because the CPU is relatively large and, arguably, the centerpiece of the chip.

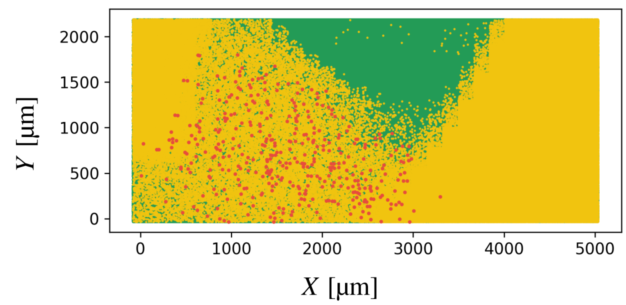


Figure 17: Scan of the chip surface.

Figure 18 shows a similar scatter plot, now pairing the glitch delay and the glitch power. The power randomly varies between 20% and 100% of the physical maximum; 500 is merely a scaling factor configured in software. The key takeaway of the plot is that two different instruction corruptions result in the desired jump.

Our setup performs around 3.4 attempts per second. The red dots can be reproduced with success rates of around 2%, upon fixing the position (X,Y), the delay, and the power.

4.4 Root Cause Analysis

As for virtually all FI attacks described in the literature, there is no absolute certainty about the exact instruction corruption that caused the attack to succeed. Nevertheless, clues can be obtained.

The easiest available source of clues is the UART log. Recall that `ets_fatal_exception_handler` prints registers `a2` to `a6`. By matching the printed values to the GDB execution trace, we conclude that `a2` to `a6` from `ets_secure_boot_verify_signature` are printed. The addition `add.n a2, a6, a2` at address `0x40065481` is confirmed to take place, *i.e.*, the instruction corruptions happen from `0x40065483` onwards. Another clue obtained from UART is that for the second cluster of red dots in Fig. 18, the CRC error string is printed, whereas for the first cluster,

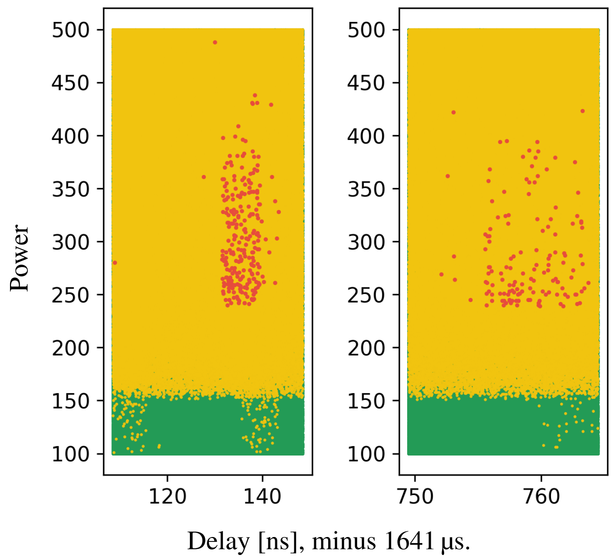


Figure 18: Delay versus power.

this print is missing. Based on the above observations, we set forth a hypothesis for each cluster:

1. For the first cluster, we corrupt an instruction in `ets_secure_boot_verify_signature` between addresses `0x40065483` and `0x40065491`. This corruption causes a jump to `ets_fatal_exception_handler` with immediate effect, and without shifting the register window. The above behavior is consistent with `jx` in Corruption 3, but inconsistent with overwriting the return address `a0` in Corruption 2.

2. For the second cluster, we corrupt an instruction in the beginning of `ets_printf`. This corruption causes a jump to `ets_fatal_exception_handler` with a delayed effect, and rotates back the window with eight registers. This behavior is consistent with overwriting the return address `a0`.

A second source of clues is the aforementioned notion of using FI as an oscilloscope. Figure 19 covers a narrow time internal around the two clusters of red dots. Purple dots indicate that the stored checksum is wrong, whereas the recomputed checksum is correct. Pink dots indicate that both checksums are wrong. We see a stripe pattern with a period of around 25 ns. This corresponds to a frequency of 40 MHz, which is also the frequency of the external crystal oscillator.

The first cluster of reds dots is located within a purple region, which is consistent with Corruption 3. Note that the stored checksum is loaded right before.

4.5 Jumping to Download Mode

After having tuned our parameters, we prepare the `0x80008ceb` image for Download Mode and burn the eFuse in Fig. 12d. If successful, we can leverage this mode to read and write memory, and execute arbitrary code.

To verify that we are successful in getting into *Download Mode*, we use UART to send the packet below, which is a command for reading memory. As defined in Espressif's Serial Line Internet Protocol (SLIP) [11], each packet begins and ends with byte `0xc0`. The second byte is `0x00` and indicates that the packet is a request. The third byte is `0x0a` and indicates the nature of the request: reading data from a memory address. Byte 4 and 5, with value `0x0400`, indicate that four bytes of data are attached to this packet, *i.e.*, the memory address. Bytes 6 to 9, with value `0x00000000`, are unused. Bytes 10 to 13 encode the memory address `0x3f401000` in *little endian*. This virtual address is mapped to physical address `0x1000` of the external Flash, where the firmware file header is written [10], starting with a magic byte `0xe9`.

`c0000a04000000000010403fc0`.

The ESP32 responds with the packet below. Unlike before, the second byte is `0x01` and indicates that the packet is a response. The third byte is still `0x0a`, repeating the nature of the request. Again, byte 4 and 5, with value `0x0400`, indicate that four bytes of data are attached. Byte 6 to 9, with value `e9030210`, are decrypted Flash contents. Figure 20 displays the Flash contents before and after encryption, which confirms the match.

`c0010a0400e903021000000000c0`.

The success rate for jumping to Download Mode is the same as for jumping to `ets_fatal_exception_handler`: roughly 2%. Because an attacker only needs to succeed once, further optimizing this success rate is unnecessary.

5 Conclusion

Our work demonstrates that the ESP32 V3, even though it is specifically hardened against FI attacks, is still vulnerable. Using a single EM glitch, we were able to bypass the SoC's most significant security features, *i.e.*, *Secure Boot V2*, *Flash Encryption*, the disabling of *Download Mode* by burning fuses, and the enabling of *Release Mode* by burning fuses. We have no reasons to believe that a skilled and resourceful attacker would be unable to perform this attack on a commercial product that incorporates an ESP32 V3 chip.

Moreover, we believe to have demonstrated an FI technique that is versatile enough to be applied to various architectures, which includes vendors other than Espressif. Our approach

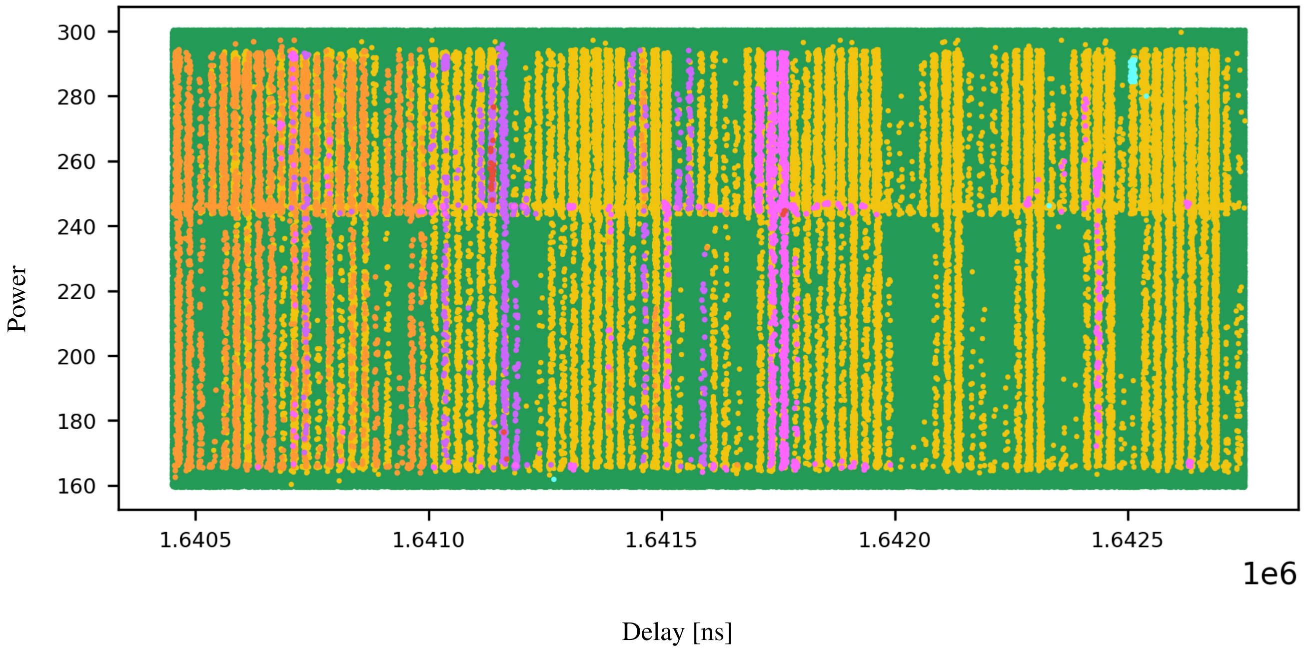


Figure 19: FI as an oscilloscope, revisited.

```
00000000 E9 03 02 10  3C 06 08 40  EE 00 00 00  00 00 03 00  ....<..@........
00000010 00 FF FF 00  00 00 00 01  20 00 FF 3F  C4 0C 00 00  ........ ..?....
00000020 FF FF FF FF  28 50 04 00  FF AC 00 00  01 00 00 00  ....(P..........
00000030 00 F0 F5 3F  00 00 00 00  04 00 00 00  05 00 00 00  ...?............
00000040 06 00 00 00  07 00 00 00  41 73 73 65  72 74 20 66  ........Assert f
00000005 61 69 6C 65  64 20 69 6E  20 25 73 2C  20 25 73 3A  ailed in %s, %s:
```

(a) Unencrypted.

```
00000000 BB C3 FC 39  C1 52 A1 1B  05 D8 E9 FF  A2 4E D3 64  ...9.R.......N.d
00000010 7C 55 95 FC  DC 5C AA BB  AC 81 38 A1  0F 99 62 42  |U...\....8...bB
00000020 98 D1 9C 13  66 1C 49 D1  E4 C4 42 6F  D9 76 24 55  ....f.I...Bo.v$U
00000030 DD 4A C4 ED  FB 01 05 18  29 02 4A 7A  F4 01 4E 52  .J......).Jz..NR
00000040 C1 2C B9 02  77 6F DE 4B  72 24 1A DB  2D A9 1D 3E  .,..wo.Kr$..-..>
00000005 39 E1 0D BB  A3 6F BA B1  DA E5 02 A0  27 76 00 64  9....o......'v.d
```

(b) Encrypted.

Figure 20: Hexadecimal dump of a Flash image (a) before encryption and (b) after encryption.

marks the first successful demonstration of loading an arbitrary value into the PC register of a CPU without being able to directly control the value. Modifying ciphertext in order to load the result of a computation on the plaintext into the PC using a single glitch represents a previously unseen level of complexity for such attacks.

The vulnerabilities we exploited on the ESP32 V3 require a new hardware revision as they cannot be mitigated by a software patch. If such a revision would be made, the attack could be mitigated by simply not printing the checksum values on the serial interface. However, given that variations on our FI technique are not limited to the checksum operation, the printing of any information on the serial interface should be carefully assessed. Either way, Espressif indicated that the attack presented in this article does not apply to the ESP32-S2, ESP32-C3, ESP32-S3, and future chips. We did not investigate what is different for those chips that would yield our attack inapplicable.

Acknowledgments

We thank Espressif for establishing a smooth vulnerability-disclosure process.

References

[1] Karim M. Abdellatif, Olivier Hériveaux, and Adrian Thillard. Unlimited results: Breaking firmware encryption of ESP32-V3. Cryptology ePrint Archive, Paper 2023/090, January 2023. https://eprint.iacr.org/2023/090.

[2] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[3] Kévin Courdesses (Courk). Fault injection attacks against the ESP32-C3 and ESP32-C6, January 2024. https://courk.cc/esp32-c3-c6-fault-injection#esp32-c3-c6-fault-injection [Accessed: Feb 9, 2024].

[4] derrek, nedwill, and naehrwert. Nintendo hacking 2016 – game over, December 2016. 33rd Chaos Communication Congress (33C3), https://media.ccc.de/v/33c3-8344-nintendo_hacking_2016 [Accessed: Mar 11, 2024].

[5] Espressif. ESP-IDF programming guide – flash encryption. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/flash-encryption.html [Accessed: Mar 12, 2024].

[6] Espressif. ESP-IDF programming guide – secure boot v2. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/secure-boot-v2.html [Accessed: Mar 12, 2024].

[7] Espressif. ESP32 chip revision v3.0 – user guide. https://www.espressif.com/sites/default/files/documentation/esp32_chip_revision_v3_0_user_guide_en.pdf [Accessed: Mar 12, 2024].

[8] Espressif. ESP32-DevKitC V4 getting started guide. https://docs.espressif.com/projects/esp-idf/en/stable/esp32/hw-reference/esp32/get-started-devkitc.html [Accessed: Mar 12, 2024].

[9] Espressif. ESP32 series ROM ELF files. https://github.com/espressif/esp-rom-elfs/releases [Accessed: Mar 12, 2024].

[10] Espressif. Firmware image format. https://docs.espressif.com/projects/esptool/en/latest/esp32/advanced-topics/firmware-image-format.html [Accessed: Mar 12, 2024].

[11] Espressif. Serial protocol. https://docs.espressif.com/projects/esptool/en/latest/esp32/advanced-topics/serial-protocol.html [Accessed: Mar 12, 2024].

[12] Espressif. Espressif security advisory concerning fault injection and secure boot (cve-2019-15894), 2019. https://www.espressif.com/en/news/Espressif_Security_Advisory_Concerning_Fault_Injection_and_Secure_Boot [Accessed: Mar 12, 2024].

[13] Espressif. ESP32-WROOM-32E – ESP32-WROOM-32UE – Datasheet v1.6, 2023. https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf [Accessed: Mar 12, 2024].

[14] Espressif. Security advisory concerning breaking the hardware AES core and firmware encryption of ESP32 chip revision v3.0. Technical report, 2023. https://www.espressif.com/sites/default/files/advisory_downloads/AR2022-003%20Security%20Advisory%20Concerning%20Breaking%20the%20Hardware%20AES%20Core%20and%20Firmware%20Encryption%20of%20ESP32%20Chip%20Revision%20v3.0%20-%20V2.0%20EN.pdf [Accessed: Mar 12, 2024].

[15] Espressif. Security advisory concerning bypassing secure boot and flash encryption using CPA and FI attack on ESP32-C3 and ESP32-C6. Technical report, 2023. https://www.espressif.com/sites/default/files/advisory_downloads/AR2023-007%20Security%20Advisory%20Concerning%20Bypassing%20Secure%20Boot%20and%20Flash%20Encryption%20using%20CPA%20and%20FI%20attack%20on%20ESP32-C3%20and%20ESP32-C6%20EN.pdf [Accessed: Mar 12, 2024].

[16] Espressif. Security advisory concerning bypassing secure boot and flash encryption using EMFI. Technical report, 2023. https://www.espressif.com/sites/default/files/advisory_downloads/AR2023-005%20Security%20Advisory%20Concerning%20Bypassing%20Secure%20Boot%20and%20Flash%20Encryption%20Using%20EMFI%20EN.pdf [Accessed: Sep 12, 2023].

[17] Travis Goodspeed. GameBoy ROM tutorial, March 2023. https://github.com/travisgoodspeed/gbrom-tutorial [Accessed: Mar 12, 2024].

[18] Travis Goodspeed. Thread on X by @travisgoodspeed, November 2023. https://threadreaderapp.com/thread/1728420233050747287.html [Accessed: Mar 12, 2024].

[19] James Gratchoff. Proving the wild jungle jump. Technical report, University of Amsterdam, July 2015. https://www.os3.nl/_media/2014-2015/courses/rp2/p48_report.pdf.

[20] Tim Hummel. Exploring effects of electromagnetic fault injection on a 32-bit high speed embedded device microprocessor. Master Thesis, University of Twente, July 2014.

[21] Internet Engineering Task Force (IETF). RFC 8017 – PKCS #1: RSA cryptography specifications version 2.2. https://datatracker.ietf.org/doc/html/rfc8017 [Accessed: Mar 12, 2024].

[22] LimitedResults. Fatal fury on ESP32: Time to release hardware exploits. BlackHat Europe 2019, December 2019. https://www.blackhat.com/eu-19/briefings/schedule/#fatal-fury-on-esp-time-to-release-hardware-exploits-17336.

[23] Nourdin Aït El Mehdi. Analyzing the resilience of modern smartphones against fault injection attacks. Master Thesis, Delft University of Technology, June 2019.

[24] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4:145–156, 2014.

[25] The On-Line Encyclopedia of Integer Sequences. A048651, July 2007. https://oeis.org/A048651 [Accessed: Mar 12, 2024].

[26] Colin O'Flynn. Low-cost body biasing injection (BBI) attacks on WLCSP devices. In Pierre-Yvan Liardet and Nele Mentens, editors, *19th Conference on Smart Card Research and Advanced Applications (CARDIS 2020)*, volume 12609 of *Lecture Notes in Computer Science*, pages 166–180. Springer, November 2020. https://eprint.iacr.org/2020/1228.pdf.

[27] Raelize. Breaking SoC security by glitching OTP data transfers, 2020. https://hardwear.io/usa-2022/speakers/cristofaro-mune.php.

[28] Raelize. Espressif ESP32: Bypassing encrypted secure boot (CVE-2020-13629), September 2020. https://raelize.com/blog/espressif-esp32-bypassing-encrypted-secure-boot-cve-2020-13629/.

[29] Riscure. EM-FI transient probe. https://www.riscure.com/products/em-fi-transient-probe/ [Accessed: Mar 12, 2024].

[30] Albert Spruyt, Alyssa Milburn, and Łukasz Chmielewski. Fault injection as an oscilloscope: Fault correlation analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):192–216, Dec. 2020.

[31] Michael Steil. 17 mistakes Microsoft made in the Xbox security system. In *22nd Chaos Communication Congress*, 2005.

[32] Tensilica, Inc. Xtensa instruction set architecture (ISA) – Reference manual, April 2010. https://0x04.net/~mwk/doc/xtensa.pdf [Accessed: Mar 12, 2024].

[33] Niek Timmers and Cristofaro Mune. Using fault injection to turn data transfers into arbitrary execution, 2019. https://powerofcommunity.net/poc2019/Niek.pdf.

[34] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM using fault injection. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2016)*, pages 25–35. IEEE Computer Society, August 2016.

[35] Marc Witteman and Martijn Oostdijk. Secure application programming in the presence of side channel attacks. In *RSA conference*, volume 2008, 2008.