ESP32 Flash 加密指南

版本 1.0

关于本手册

本文介绍了 ESP32 flash 加解密的原理以及如何使用和关闭加密功能。

发布说明

日期	版本	发布说明
2017.07	V1.0	首次发布。

文档变更通知

用户可通过乐鑫官网订阅技术文档变更的电子邮件通知。

目录

1. 概述

ESP32 flash 加密功能用于加密 ESP32 SPI flash 里的内容。启用 flash 加密时，大部分 flash 内容可以防止物理读取。

Flash 加密功能与安全启动 (secure boot) 功能是分开的，用户可以直接使用 flash 加密功能。但是，如果要保证安全的使用环境，建议将这两个功能一起使用。

> 📖 **说明：**
>
> 启用 *flash* 加密会限制后续对 *flash* 内容的更新。使用 *flash* 加密功能前请必阅读本文档。

2. Flash 加密机制

- Flash 的内容是使用 256-bit 密钥的 AES 进行加密的。Flash 加密密钥存储在芯片内部的 eFuse 中，并且（在默认情况下）软件不能对其读写。
- 通过 ESP32 的 flash 缓存映射功能可以对 flash 进行透明读写，映射到地址空间的任意 flash 区域在读取时都会被透明解密。
- 用户将明文数据烧录到 ESP32 来启动加密功能，引导加载程序会在首次启动时对数据在 flash 原处进行加密。
- 并非所有的 flash 数据都会被加密。被加密 flash 数据包括：
  - 引导加载程序
  - secure boot 引导加载程序摘要（如果启用了 secure boot）
  - 分区表
  - 所有“app”类型的分区
  - OTA data 分区（OTA data 最多可以有一个，当需要使用 OTA 功能时会有该分区）
  - 分区表中所有标有“encrypt”的分区

    一些数据分区可能需要保持不加密以便于访问，或者可以使用在数据加密时无效的 flash 更新算法。用于非易失性存储的 NVS 分区不能加密。
- Flash 加密密钥存储在 ESP32 芯片内部的 eFuse 密钥块 1 中。默认情况下，该密钥是读写保护的，软件无法访问或更改。
- Flash 加密的算法是 AES-256，密钥会根据 flash 的每 32 字节块的偏移地址进行加密。即 flash 的每 32 字节块（两个连续的 16 字节 AES 块）使用一个唯一的密钥进行加密，这个唯一的密钥是由 flash 加密密钥生成的。
- 虽然在芯片上运行的软件可以透明解密 flash 内容，但默认情况下，如果启动了 flash 加密，则 UART 引导加载程序无法解密（或加密）数据。
- 所以，如果要使用 flash 加密的功能，在编写代码时必须谨慎。

3. Flash 加密初始化

本章介绍了默认（推荐）的 flash 加密初始化的过程。这个过程可以为开发或其他目的定制。详细信息请参阅章节 ***Flash* 加密高级功能**。

> ⚠ **注意：**
>
> 一旦在第一次启动时使用了 *flash* 加密功能，则硬件最多允许后续 *3* 次通过串口更新 *flash* 内容。串口升级需要遵循特定的程序，具体请参考章节**串口烧写**。

启用 flash 加密的过程如下：

1. 编译引导加载程序时必须确保支持 flash 加密功能。在 ***make menuconfig*** 中，导航到 ***Security Features***，将 ***Enable flash encryption on boot*** 勾选为 ***Yes***。
2. 如果要使用 Secure Boot 功能，则最好同时勾选这些选项。请先阅读 Secure Boot 的文档。
3. 构建并烧录引导程序、分区表和出厂 app image。这些分区最初被写入 flash 时未加密。
4. 第一次启动时，引导加载程序会读出 `FLASH_CRYPT_CNT` eFuse 的值为 0（出厂默认值），然后会使用硬件随机数生成器生成 flash 加密密钥。此密钥存储在 eFuse 中。此密钥被读写保护，防止软件进一步访问。
5. 所有加密分区由引导加载程序直接就地加密。就地加密可能需要一些时间（大的分区可能需要一分钟）。

> ⚠ **注意：**
>
> 第一次启动加密正在运行时不要中断 *ESP32* 的电源。如果电源中断，*flash* 内容会被破坏，需要再次使用未加密的数据进行烧写。重新烧写不会计入烧写的限制总数。

6. 烧写完成后，默认烧写 eFuse 来防止加密的 flash在 UART 引导加载程序运行时被访问。
7. `FLASH_CRYPT_CONFIG` eFuse 也烧写为最大值 (0xF)，以最大化 flash 算法中被修改的密钥的比特数。更多信息请参阅章节**设置 *FLASH_CRYPT_CONFIG***。
8. 最后，`FLASH_CRYPT_CNT` eFuse 被烧写为初始值 1。正是这个 eFuse 激活了透明 flash 加密层，并限制了后续串口重烧 flash 的次数。有关 `FLASH_CRYPT_CNT` eFuse 的详细信息，请参阅章节**升级加密 *Flash***。
9. 引导加载程序从新加密的 flash 中自动重启。

4. 使用 Flash 加密

ESP32 app 代码可以通过调用 `esp_flash_encryption_enabled` 检查 flash 加密是否启用。启用 flash 加密后，从代码访问 flash 内容时需要注意以下事项。

4.1. Flash 解密的范围

每当 `FLASH_CRYPT_CNT` eFuse 的奇数个比特被置为 1 时，通过 MMU的 flash 缓存访问的所有 flash 内容都将被透明地解密。这些 flash 内容包括：

- flash (IROM) 中的可执行 app 代码
- 存储在 flash (DROM) 中的所有只读数据。
- 通过 `esp_spi_flash_mmap` 访问的数据
- 由 ROM 引导加载程序读取的软件 bootloader image。

> ⚠ **注意：**
>
> *MMU flash* 缓存无条件地解密所有数据。在 *flash* 中未加密存储的数据会被“透明解密”，并且在软件看来是乱码。

4.2. 读加密 Flash

要读取数据而不使用 flash 缓存 MMU 映射，我们建议使用分区读取函数 `esp_partition_read`。使用此函数时，只有加密分区中的数据才会被解密。其他分区将被不加密读取。这样，软件可以以相同的方式访问加密和非加密的 flash。

通过其他 SPI 串口 API 读取的数据不会被解密：

- 通过 `esp_spi_flash_read` 读取的数据不被解密
- 通过 ROM 函数 `SPIRead()` 读取的数据不被解密（ESP-IDF app 不支持此函数）
- 使用非易失性存储 (NVS) API 存储的数据不被加密，只能通过 SPI 的方式读取明文数据。

4.3. 写加密 Flash

可能的话，建议使用分区写函数 `esp_partition_write`。使用此函数时，数据只有写入加密分区时才会被加密。数据写入其他分区不会被加密。这样，软件可以以相同的方式访问加密和非加密的 flash。

当 `write_encrypted` 参数设置为 `true` 时，`esp_spi_flash_write` 函数才会加密数据，否则，不会加密数据。

ROM 函数 `esp_rom_spiflash_write_encrypted` 把加密数据写入 flash，ROM 函数 `SPIWrite` 把不加密的数据写入 flash。（ESP-IDF app 不支持这些函数）。

未加密数据的最小写入大小为 4 字节（对齐方式为 4 字节）。由于数据以块的形式加密，所以加密数据的最小写入大小为 16 字节（对齐方式为 16 字节）。

5. 更新加密 Flash

5.1. OTA 升级

加密分区的 OTA 升级将自动加密，使用 `esp_partition_write` 函数即可。

5.2. 串口烧写

如果不使用 secure boot，`FLASH_CRYPT_CNT` eFuse 允许通过串口（或其他物理方法）重烧 flash，最多可以重烧 3次。

该过程涉及烧写明文数据，然后重复烧写 `FLASH_CRYPT_CNT` eFuse，导致引导加载程序重新加密此数据。

5.3. 更新 Flash 的限制

加上初始的加密 flash，总共可以通过串口进行 4 次烧写。

关闭第四次加密后，`FLASH_CRYPT_CNT` 的最大值为 0xFF，加密被永久关闭。

使用 OTA 升级或预生成的加密密钥可以不受次数的限制。

5.4. 串口烧写的注意事项

- 当通过串口重新烧写时，要重新烧写每一个最初写入明文数据的分区（包括引导加载程序）。可以跳过不是“当前选择的”OTA app 分区（除非在那里找到明文 app image，否则这些分区不会被重新加密。）但是，任何标有“encrypt”标志的分区将无条件地被重新加密，这意味着任何已加密的数据将被二次加密并被破坏。

  使用 `make flash` 一起烧写所有需要烧写的分区。

- 如果启用 secure boot，则无法通过串口重新烧写，除非启用 secure boot的 ***Reflashable*** 选项，预生成一个密钥并将其烧写到 ESP32（请参阅 *secure boot* 文档）。在这种情况下，您可以重新烧写明文 secure boot 摘要和 bootloader image 到偏移地址 0x0。在烧写其他明文数据之前，需要重新烧写该摘要。

5.5. 串口重新烧写程序

1. 正常构建 app。

2. 正常使用明文数据烧写设备（使用 `make flash` 或 `esptool.py` 命令。）烧写所有之前加密的分区，包括引导程序（见上一节）。
3. 此时，设备将无法启动（消息为 `flash read err, 1000`），因为它需要一个加密的引导加载程序，但引导加载程序是明文。
4. 通过运行命令 `espefuse.py burn_efuse FLASH_CRYPT_CNT` 来烧写 `FLASH_CRYPT_CNT`。`espefuse.py` 自动将比特的计数递增 1，以此来关闭加密。
5. 重置设备，设备会重新加密明文分区，然后再次烧写 `FLASH_CRYPT_CNT` 以重新启用加密。

5.6. 关闭串口烧写

启用 flash 加密（即第一次启动完成）后，使用 `espefuse.py` 写保护 `FLASH_CRYPT_CNT`，以此关闭串口烧写。

```
espefuse.py --port PORT write_protect_efuse FLASH_CRYPT_CNT
```

这样可以防止进一步修改 flash 加密的启用和关闭。

6. 预生成 Flash 加密密钥

可以在主机上预生成一个 flash 加密密钥，并将其烧写到 ESP32 的 eFuse 密钥块中。这实现了在主机上预先加密数据，并烧写到 ESP32，而不需要明文烧写更新 flash。

这种方式有利于开发，因为它消除了 4 次重新烧写的限制。它还允许在启用 secure boot 的情况下重新烧写，因为引导加载程序不需要每次都重新烧写。

> ⚠ **注意：**
>
> 此方法仅用于协助开发，不适用于生产设备。如果是用于生产设备，请确保密钥是从高质量的随机数字源生成的，并且不要在多个设备上共享 *flash* 加密密钥。

6.1. 预生成 Flash 加密密钥

Flash 加密密钥是 32 个字节的随机数据。您可以使用 `espsecure.py` 生成随机密钥：

```
espsecure.py generate_flash_encryption_key my_flash_encryption_key.bin
```

（该数据的随机性取决于操作系统，也是 Python 安装的随机数据源。）

或者，如果您使用了 secure boot，并且有 secure boot 签名密钥，那么可以生成一个安全引导私有签名密钥的确定性 SHA-256 摘要，并将其作为 flash 加密密钥：

```
espsecure.py digest_private-key --keyfile secure_boot_signing_key.pem
my_flash_encryption_key.bin
```

（如果启用 secure boot 的 ***reflashable*** 模式，则相同的 32 个字节会作为 secure boot 的摘要密钥。）

从 secure boot 的数字签名 (signing) 密钥生成 flash 加密密钥意味着您只需要存储一个密钥文件。但是，这种方法不适用于生产设备。

6.2. 烧写 Flash 加密密钥

一旦生成了 flash 加密密钥，您需要将其烧写到 ESP32 的 eFuse 密钥块。（这必须在第一次加密启动之前完成，否则 ESP32 将生成一个软件无法访问或修改的随机密钥。）

烧写密钥到设备（只需要烧录一次）：

```
espefuse.py --port PORT burn_key flash_encryption my_flash_encryption_key.bin
```

6.3. 使用预生成的密钥进行第一次烧写

烧写密钥后，按照与章节 ***Flash* 加密初始化**相同的步骤在第一次启动时烧写明文 image。引导程序将使用预烧录的密钥启用 flash 加密，并加密所有分区。

6.4. 使用预生成的密钥重烧 Flash

Flash 加密在第一次安全启动开启后，重新烧写加密 image 需要一步额外的手动操作。我们需要预先加密想要在 flash 中更新的数据。

假设这是用于烧写明文数据的命令：

```
esptool.py --port / dev / ttyUSB0 --baud 115200 write_flash 0x10000 build / my-app.bin
```

二进制 app image ***build / my-app.bin*** 被写入 `0x10000` 。需要使用此文件名和偏移地址来加密数据，如下所示：

```
esfsecure.py encrypt_flash_data --keyfile my_flash_encryption_key.bin --address 0x10000 -o
build / my-app-encrypted.bin build / my-app.bin
```

此示例命令将使用提供的密钥加密 ***my-app.bin***，并生成一个加密的文件 ***my-app-encrypted.bin***。确保地址参数与您打算烧录 BIN 文件的地址匹配。然后，使用 `esptool.py` 烧写加密的 BIN 文件：

```
esptool.py --port / dev / ttyUSB0 --baud 115200 write_flash 0x10000 build / my-app-
encrypted.bin
```

不需要其他操作或 eFuse 操作，因为数据在烧写时已经加密了。

7. 关闭 Flash 加密

如果您不小心启用了 flash 加密，下一次烧写明文数据时将会软启动 ESP32（设备将不断重启，并打印错误 `flash read err, 1000`）。您可以通过烧写 `FLASH_CRYPT_CNT` eFuse 来关闭 flash 加密。

1. 首先运行 `make menuconfig`，取消选中 ***Security Features*** 下的 ***Enable flash encryption boot***。
2. 退出 ***menuconfig*** 并保存新配置。
3. 再次运行 `make menuconfig`，并仔细检查你是否真的取消了这个选项！(如果启用此选项，则引导加载程序将在启动时立即重新启用加密。)
4. 运行 `make flash` 构建并烧写一个新的引导加载程序和 app，而不启用 flash 加密。
5. 运行 `espefuse.py` (`components/esptool_py/esptool`) 来禁用 `FLASH_CRYPT_CNT` eFuse：
6. 重启 ESP32，flash 加密应该已经被关闭，引导加载程序将正常启动。

8. Flash 加密的局限

Flash 加密可防止明文被读出，从而防止未经授权的固件读写。了解 flash 加密系统的局限性非常重要：

- Flash 加密性能的强弱取决于密钥。因此，我们建议首次启动时在设备上生成密钥（默认行为）。如果在设备外生成密钥，请确保遵循正确的过程。
- 并非所有数据都被加密存储。如果将数据存储在 flash，请检查您使用的方法（库，API 等）是否支持 flash 加密。
- Flash 加密不会阻止攻击者了解闪存的高级布局。这是因为相邻的两个 16 字节 AES 块使用同一个 AES 密钥。如果这些相邻的 16 字节块包含相同的内容（比如空白或填充区域）时，这些块被加密时会产生匹配的加密块对。这可能会让攻击者在加密设备之间进行高级别比较（即，判断两台设备是否运行相同的固件版本）。
- 出于同样的原因，攻击者可以随时知道一对相邻的 16 字节块（32 字节对齐）是否包含相同的内容。如果要存储的是敏感数据，请牢记这一点。可以设计一下 flash 阻止攻击者（每 16 个字节使用计数器字节或其他不相同的值即可）。
- Flash 加密不足以阻止攻击者修改设备的固件。要防止未经授权的固件在设备上运行，请配合使用 secure boot。

9. Flash 加密高级功能

9.1. 加密分区标志

某些分区默认加密。您也可以将任意分区标记为需要加密：

在分区表描述的 CSV 文件中有标志字段。

标志字段通常留空，如果在字段中写 `encrypted`，则分区将标记为加密，并将此处写入的数据视为加密（就像 app 分区一样）：

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x9000,  0x6000
phy_init, data, phy,     0xf000,  0x1000
factory,  app,  factory, 0x10000, 1M
secret_data, 0x40, 0x01, 0x20000, 256K, encrypted
```

- 默认分区表都不包括任何加密的数据分区。
- 没有必要将 app 分区标记为加密，它们总是被视为加密的。
- 如果未启用 flash 加密，则 `encrypted` 标志不起作用。
- 如果您希望保护 `phy_init` 数据防止物理读取或修改，可以将 phy 分区标记为加密。
- nvs 分区无法被标记为加密。

9.2. 启用 UART 引导加载程序加解密

默认情况下，首次启动时，flash 加密过程会烧写 `DISABLE_DL_ENCRYPT`，`DISABLE_DL_DECRYPT` 和 `DISABLE_DL_CACHE`：

- `DISABLE_DL_ENCRYPT` 在 UART 引导加载程序启动模式下禁用 flash 加密操作。
- `DISABLE_DL_DECRYPT` 在 UART 引导加载程序模式下关闭透明解密，即使使用 `FLASH_CRYPT_CNT` eFuse 将其使能。
- `DISABLE_DL_CACHE` 在 UART 引导加载程序模式下禁用整个 MMU flash 缓存。

在第一次启动之前，可以仅烧写某些 eFuse 比特，并写保护其余的 eFuse 比特（使用未设置的值 0），以便保留它们。例如：

```
espefuse.py --port PORT burn_efuse DISABLE_DL_DECRYPT
espefuse.py --port PORT write_protect_efuse DISABLE_DL_ENCRYPT
```

（请注意，这 3 个 eFuse 比特都通过同一个写保护位关闭，因此写保护一个比特将一起作用于这 3 个比特。因此，配置比特必须在写保护之前进行。）

⚠ **注意：**

- 写保护这些 *eFuse* 比特以保持他们不被设置作用不大，因为 *esptool.py* 不支持写入或读取加密 *flash*。
- *DISABLE_DL_DECRYPT* 未被设置（即值为 *0*），这会使 *flash* 加密无效，因为具有物理访问权限的攻击者可以使用 *UART* 引导加载程序模式（使用自定义存根代码）读出 *flash* 中的内容。

9.3. 设置 FLASH_CRYPT_CONFIG

`FLASH_CRYPT_CONFIG` eFuse 决定 flash 加密密钥中进行块偏移“调整”的比特数。详细信息请参阅章节 ***Flash* 加密算法**。

引导加载程序的首次引导始终将此值设置为最大值 0xF。

可以手动烧写这些 eFuse，并在首次启动之前进行写保护，以便选择不同的调整值。但是我们不推荐这样做。

当 `FLASH_CRYPT_CONFIG` 值为零时，我们强烈建议不要写保护。如果此 eFuse 设置为零，flash 加密密钥中的比特不会被调整，flash 加密算法等同于 AES ECB 模式。

10. 技术参考

10.1. FLASH_CRYPT_CNT eFuse

FLASH_CRYPT_CNT 是一个 8-bit eFuse 字段，用于控制 flash 加密。Flash 加密的启用或关闭取决于此 eFuse 中设置为“1”的位数，具体如下：

- 当偶数个 (0, 2, 4, 6, 8) 比特被置位时：关闭 Flash 加密，任何加密数据都不能被解密。

如果引导加载程序的 ***Enable flash encryption on boot*** 被启用，则引导加载程序会获得这个信息，然后当它发现未被加密的数据时会重新加密 flash。加密完成后，引导加载程序会将 eFuse 中的另一比特置为'1'，表示有奇数个比特被置为 1。

1. 在第一个明文启动时，比特的计数为全新值 0，引导加载程序在加密后将计数更改为 1（值为 0x01）。
2. 重烧 flash 后，比特数手动更新为 2（值为 0x03）。重新加密后，引导程序将计数更改为 3（值为 0x07）。
3. 重烧 flash 后，比特数手动更新为 4（值为 0x0F）。重新加密后，引导程序将计数更改为 5（值为 0x1F）。
4. 重烧 flash 后，比特数手动更新为 6（值为 0x3F）。重新加密后，引导程序将计数更改为 7（值为 0x7F）。

- 当奇数个 (1, 3, 5, 7) 比特被置位时：使能透明读取加密 flash。
- 当 8 个比特被被置位后（eFuse 值为 0xFF）：关闭透明读取加密 flash，任何加密数据永久不可访问。引导加载程序通常会检测到这种情况并停止。为避免这种状态被利用加载未经授权的代码，必须使用 secure boot 或者写保护 FLASH_CRYPT_CNT eFuse。

10.2. FLASH 加密算法

- AES-256 在 16 字节的数据块上运行。Flash 加密引擎对 32 字节块（即两个连续的 AES 块）中的数据进行加解密。
- AES 算法在 flash 加密中被反转使用，即 flash 加密的“加密”操作是 AES 解密，“解密”操作是 AES 加密。这种机制是出于性能的考虑，并不会影响算法的有效性。
- Flash 加密的主密钥存储在 eFuse (BLOCK1) 中，默认情况下软件不能读写。
- 每个 32 字节块（两个相邻的 16 字节 AES 块）共用一个唯一的密钥进行加密。这个密钥源自 eFuse 中的主密钥，与 flash 中该块的偏移进行异或（“密钥调整”）。

- 具体的调整取决于 FLASH_CRYPT_CONFIG eFuse 的配置。FLASH_CRYPT_CONFIG 是一个 4-bit eFuse，其中每个比特使能特定范围的密钥比特的异或：
  - Bit1，密钥的 0-66 比特被异或。
  - Bit2，密钥的 67-131 比特被异或。
  - Bit3，密钥的 132-194 比特被异或。
  - Bit4，密钥的 195-256 比特被异或。

  建议将 FLASH_CRYPT_CONFIG 始终设置为默认值 0xF，以便所有密钥比特都能与块偏移进行异或。有关详细信息，请参阅章节**设置 *FLASH_CRYPT_CONFIG***。
- 块偏移（比特 5-23）的高 19 位与主加密密钥进行异或。选择此范围有两个原因：flash 最大为 16 MB（24 位），每个块为 32 字节，所以最低有效 5 位始终为零。
- 19 个块偏移比特与 flash 加密密钥的 256 个比特之间存在特定映射，这种映射决定哪个比特与哪个块偏移进行异或。有关映射的完整信息，请参阅 espsecure.py 源代码中的_FLASH_ENCRYPTION_TWEAK_PATTERN 变量。
- 关于 Python 中完整的 flash 加密算法，请参阅 espsecure.py 源代码中的 _flash_encryption_operation() 函数。

乐鑫 IoT 团队
www.espressif.com