UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

DANIEL FERREIRA DA SILVA

CIBERSEGURANÇA PARA IOT: Arquitetura Segura para Comunicação e Atualização de Firmware em Dispositivos ESP32

Recife
2025

DANIEL FERREIRA DA SILVA

CIBERSEGURANÇA PARA IOT: Arquitetura Segura para Comunicação e Atualização de Firmware em Dispositivos ESP32

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Engenharia de Controle e Automação.

Orientador(a): Prof. Dr. Márcio Evaristo da Cruz Brito

Recife
2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva, Daniel Ferreira da.
Cibersegurança para IoT: arquitetura segura para comunicação e atualização de firmware em dispositivos ESP32 / Daniel Ferreira da Silva. - Recife, 2025.
77p : il., tab.

Orientador(a): Márcio Evaristo da Cruz Brito
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Tecnologia e Geociências, Engenharia de Controle e Automação - Bacharelado, 2025.
Inclui referências.

1. IoT. 2. Cibersegurança. 3. Criptografia. 4. ESP32. 5. OTA. I. Brito, Márcio Evaristo da Cruz. (Orientação). II. Título.

620 CDD (22.ed.)

DANIEL FERREIRA DA SILVA

CIBERSEGURANÇA PARA IOT: Arquitetura Segura para Comunicação e Atualização de Firmware em Dispositivos ESP32

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Engenharia de Controle e Automação.

Aprovado em: 16/12/2025.

BANCA EXAMINADORA

_____

Prof. Dr. Márcio Evaristo da Cruz Brito (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Geraldo Leite Maia (Examinador Interno)
Universidade Federal de Pernambuco

_____

Eng. M.Sc. Néstor Iván Medina Giraldo (Examinador Interno)
Universidade Federal de Pernambuco

AGRADECIMENTOS

Agradeço a todos que contribuíram, direta ou indiretamente, para a realização deste trabalho e para a minha trajetória acadêmica. Aos professores, pela dedicação, orientação e conhecimentos compartilhados ao longo do curso. Aos amigos e colegas, pelo apoio, incentivo e pelas trocas de experiências que contribuíram para o meu crescimento pessoal e profissional. À minha família, pelo suporte, compreensão e encorajamento constantes, fundamentais para a conclusão desta etapa.

RESUMO

O presente trabalho tem como objetivo desenvolver e integrar múltiplas camadas de segurança aplicáveis a sistemas de Internet das Coisas (IoT), com foco na proteção de dados e na confiabilidade da comunicação entre dispositivos conectados. A proposta consiste na criação de um sistema capaz de realizar a troca segura de mensagens, assegurando que apenas usuários e dispositivos autenticados possam se comunicar e que o código-fonte permaneça protegido contra acessos indevidos e tentativas de modificação.

Para a validação prática da proposta, foi implementado um ambiente experimental composto por dois módulos ESP32, nos quais um atua como emissor e o outro como receptor de mensagens. A comunicação entre os dispositivos é realizada por meio do protocolo ESP-NOW, desenvolvido pela Espressif, o qual possibilita a troca de mensagens de forma eficiente e com baixo consumo de energia, utilizando mecanismos nativos de criptografia simétrica baseados no algoritmo AES para a proteção dos dados transmitidos. Além da camada de confidencialidade, o sistema emprega o algoritmo de assinatura digital ECDSA (Elliptic Curve Digital Signature Algorithm), garantindo a autenticidade e a integridade das mensagens transmitidas.

Complementarmente, foram integradas as funcionalidades de Secure Boot, criptografia da memória flash e atualização segura de firmware Over-The-Air (OTA), assegurando a proteção integral do dispositivo desde o processo de inicialização até sua manutenção remota.

Os resultados obtidos demonstram que a integração das camadas de segurança propostas reforça significativamente a resiliência do sistema IoT contra ataques de interceptação e adulteração de dados, oferecendo uma arquitetura viável, segura, escalável e compatível com aplicações industriais e domésticas que exigem elevado nível de segurança.

Palavras-chave: IoT; Cibersegurança; Criptografia; ESP32; OTA.

ABSTRACT

The present work aims to develop and integrate multiple security layers applicable to Internet of Things (IoT) systems, focusing on data protection and the reliability of communication between connected devices. The proposal consists of creating a system capable of performing secure message exchange, ensuring that only authenticated users and devices can communicate, and that the source code remains protected against unauthorized access and modification attempts.

For the practical validation of the proposed approach, an experimental environment composed of two ESP32 modules was implemented, in which one acts as a message transmitter and the other as a receiver. Communication between the devices is carried out using the ESP-NOW protocol, developed by Espressif, which enables efficient message exchange with low power consumption by employing native symmetric cryptographic mechanisms based on the AES algorithm to protect the transmitted data. In addition to the confidentiality layer, the system employs the ECDSA (*Elliptic Curve Digital Signature Algorithm*), ensuring the authenticity and integrity of the transmitted messages.

Furthermore, Secure Boot, flash memory encryption, and secure Over The Air (OTA) firmware update functionalities were integrated, ensuring complete device protection from the boot process to remote maintenance.

The obtained results demonstrate that the integration of the proposed security layers significantly enhances the resilience of the IoT system against data interception and tampering attacks, providing a viable, secure, scalable architecture compatible with industrial and domestic applications that require a high level of security.

Keywords: IoT; Cybersecurity; Cryptography; ESP32; OTA.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

4G	Fourth Generation
5G	Fifth Generation
ACK	Acknowledged
AES	Advanced Encryption Standard
BLE	Bluetooth Low Energy
CAN	Controller Area Network
CCMP	Counter Mode Cipher Block Chaining Message Authentication Code Protocol
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DDoS	Distributed Denial of Service
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
ESP-IDF	Espressif IoT Development Framework
FAT	File Allocation Table
HTTPS	HyperText Transfer Protocol Secure
I2C	Inter-Integrated Circuit
I2S	Inter-Integrated Circuit Sound
IEEE	Institute of Electrical and Electronics Engineers
IIoT	Industrial Internet of Things
IP	Internet Protocol
IRAM	Internal Random Access Memory
IoT	Internet of Things
LMK	Local Master Key
LoRa	Long Range
MAC	Media Access Control
NVS	Non-Volatile Storage
OSI	Open Systems Interconnection
OTA	Over the Air
PEM	Privacy Enhanced Mail
PMK	Primary Master Key

ROM	Read Only Memory
RSA	Rivest Shamir Adleman
RTC	Real Time Clock
SHA	Secure Hash Algorithm
SoC	System on Chip
SPI	Serial Peripheral Interface
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver Transmitter
URL	Uniform Resource Locator

SUMÁRIO

1 INTRODUÇÃO

A Internet das Coisas (IoT) tem se consolidado como uma das mais promissoras vertentes tecnológicas da era digital, promovendo maior conectividade, automação e praticidade no cotidiano. Trata-se de uma tecnologia em franca expansão, impulsionada pelos avanços em áreas como redes de computadores, microeletrônica, computação embarcada e sensoriamento (ATZORI, IERA e MORABITO, 2010). Estimativas apontam que o número global de dispositivos conectados via IoT atingiu 18,8 bilhões em 2024, representando um crescimento de 13 % em relação a 2023, com projeções que indicam a marca de 40 a 43 bilhões de conexões até 2030 (IOT ANALYTICS, 2024); (RFID JOURNAL, 2024). Em termos de mercado, o setor global de IoT movimentou US$ 714,48 bilhões em 2024, com expectativa de alcançar US$ 4 trilhões em 2032, a uma taxa média de crescimento anual (CAGR) de 24,3 % (BUSINESS INSIGHTS FORTUNE, 2024).

A presença da IoT é perceptível em diversos segmentos. Na agricultura, sensores inteligentes monitoram parâmetros como temperatura, umidade e acidez do solo, auxiliando no manejo sustentável e na produtividade (GARCIA, PARRA, *et al.*, 2020). No setor da saúde, o uso de wearables e dispositivos conectados para o monitoramento remoto de pacientes já movimenta cerca de US$ 1,6 bilhão no Brasil, favorecendo diagnósticos preventivos e a gestão hospitalar (KEN RESEARCH, 2025). Já nas cidades inteligentes, soluções de IoT têm se expandido rapidamente, com o mercado brasileiro estimado em US$ 17,2 milhões em 2024, devendo atingir US$ 61 milhões até 2030, com uma CAGR de 25,3% (GRANDVIEW RESEARCH, 2024). Essa tendência reflete um contexto urbano em que 85 % da população brasileira vive em áreas urbanas, impulsionando a digitalização de serviços públicos e a modernização da infraestrutura (BRASIL, 2021).

O Brasil também apresenta crescimento expressivo na adoção da IoT. O mercado nacional foi avaliado em US$ 18,41 bilhões em 2024, com previsão de atingir US$ 99,34 bilhões até 2033, e crescimento médio anual de 17,8% (IMARC GROUP, 2024). No mesmo período, o número de conexões IoT passou de 28 milhões em 2020 para 46,2 milhões em 2024, consolidando o país como um dos principais mercados emergentes da América Latina (MVNO INDEX, 2024).

Apesar dos benefícios, a expansão acelerada da IoT traz consigo desafios significativos em segurança, privacidade e confiabilidade (SICARI, RIZZARDI, *et al.*, 2015). Muitos dispositivos conectados são desenvolvidos com foco na funcionalidade, negligenciando mecanismos de proteção robustos. Esse cenário torna os sistemas vulneráveis a ataques cibernéticos, como demonstrado pelo botnet Mirai, responsável em 2016 por um ataque de negação de serviço distribuído (DDoS) em larga escala, explorando falhas de segurança em câmeras IP, roteadores e outros dispositivos IoT (ANTONAKAKIS, APRIL, *et al.*, 2017).

Diante desse panorama, torna-se imprescindível o desenvolvimento de mecanismos de segurança integrados que assegurem a confidencialidade, integridade e autenticidade das comunicações entre dispositivos. Este trabalho propõe a criação de um sistema de camadas de segurança aplicáveis à IoT, com o objetivo de fortalecer a proteção na troca de dados, impedir o acesso não autorizado e mitigar o risco de interceptação ou roubo de informações sensíveis. Assim, busca-se contribuir para o avanço de uma infraestrutura digital mais segura e confiável, alinhada às crescentes demandas da sociedade conectada.

1.1 Objetivos

1.1.1 Geral

Desenvolver um conjunto de ferramentas que promovam a segurança de dispositivos IoT, com foco na proteção dos usuários, de forma a garantir a Confidencialidade, Autenticidade e Integridade (CIA) dos sistemas a serem concebidos.

1.1.2 Específicos

1.1.2.1 Desenvolver mecanismos de troca de mensagens seguras utilizando o protocolo CCMP.

1.1.2.2 Implementar criptografia de memória flash para evitar o acesso não autorizado ao código fonte.

1.1.2.3 Estabelecer métodos de autenticação confiáveis para garantir acesso ao meio apenas aos usuários autorizados.

1.1.2.4 Aplicar e validar as ferramentas propostas fazendo uso do SoC ESP32.

1.1.2.5 Implementar uma forma segura de atualização de firmware via atualização remota sem fio (Over The Air – OTA).

1.1.2.6 Documentar os métodos utilizados, bem como o seu papel na construção de sistemas IoT seguros.

1.2 Organização do Trabalho

Este trabalho está organizado em seis capítulos, conforme descrito a seguir:

- **Capítulo 2**: Reúne o embasamento teórico necessário para compreender a proposta desenvolvida. São abordados conceitos de Internet das Coisas, as principais características do microcontrolador ESP32, bem como os mecanismos e protocolos de segurança aplicáveis a estes dispositivos.

- **Capítulo 3**: Descreve a organização do processo de desenvolvimento do sistema. Este capítulo inclui a definição da arquitetura proposta, a seleção dos dispositivos e protocolos empregados, além dos métodos utilizados para validação e verificação do funcionamento do sistema.

- **Capítulo 4**: Detalha a aplicação prática da proposta, abrangendo a comunicação entre os dispositivos ESP32, a integração das camadas de segurança implementadas, a estrutura do código-fonte desenvolvido e as configurações necessárias para a execução do sistema.

- **Capítulo 5**: Apresenta os resultados obtidos durante os testes realizados, demonstrando o funcionamento da troca de mensagens, o processo de autenticação e assinatura digital, o procedimento de atualização segura, bem como as implementações de Secure Boot e da criptografia da memória flash (Flash Encryption).

- **Capítulo 6**: Expõe as conclusões do trabalho, discutindo a eficiência dos mecanismos propostos, os desafios enfrentados e as limitações observadas. Por fim, sugere possíveis aprimoramentos e direções para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os fundamentos teóricos essenciais para o desenvolvimento do sistema de comunicação segura entre dispositivos ESP32. São abordados os conceitos de Internet das Coisas e suas aplicações, os mecanismos de segurança em redes sem fio, o protocolo ESP-NOW e os mecanismos internos de criptografia simétrica baseados em AES utilizados para proteção das mensagens, bem como o uso do algoritmo de assinatura digital ECDSA (Elliptic Curve Digital Signature Algorithm) para autenticação e integridade das mensagens. Além disso, são discutidas as tecnologias de Secure Boot, criptografia da memória flash e atualização segura de firmware *Over The Air*.

2.1 Internet das Coisas

Nas últimas décadas, observou-se um avanço expressivo em diversos setores da tecnologia, especialmente no campo da comunicação. A evolução desse ecossistema tem impulsionado inovações que viabilizam a conectividade de dispositivos com recursos computacionais limitados, permitindo sua integração direta com a internet. (RAFIULLAH, SARMAD, *et al.*, 2012).

Graças a esta evolução, tem-se buscado agregar características de inteligência a esses dispositivos, permitindo que analisem informações, identifiquem padrões e tomem decisões de forma autônoma, com base nos dados coletados do ambiente. Essa capacidade de percepção e reação consolidou o conceito de *Internet das Coisas* (*Internet of Things* – IoT), termo introduzido por Kevin Ashton em 1999, ao propor um sistema capaz de conectar o mundo físico ao digital por meio de sensores e redes de comunicação (ANSHUMAN, PAWANI e MADHUSANKA, 2019).

Embora não exista uma definição universalmente aceita, uma das descrições mais abrangentes caracteriza a IoT como “Uma rede aberta e abrangente de objetos inteligentes capazes de se auto-organizar, compartilhar informações, dados e recursos, reagindo e agindo diante de mudanças no ambiente” (ANSHUMAN, PAWANI e MADHUSANKA, 2019). É importante destacar que, apesar de o termo

conter *Internet,* os dispositivos IoT não precisam estar necessariamente conectados a redes Wi-Fi ou móveis (4G/5G). Outras tecnologias de comunicação, como Bluetooth, ZigBee, LoRa e ESP-NOW, também podem ser utilizadas, dependendo da aplicação e das restrições de energia ou alcance (MADAKAM SOMAYYA, 2015).

Estima-se que até o final de 2025, existam cerca de 75 bilhões de dispositivos IoT em operação. Ainda não existe uma arquitetura totalmente padronizada e amplamente aceita para sistemas baseados IoT. Porém, as arquiteturas existentes adotam uma arquitetura flexível tomando como base o modelo OSI (Open Systems Interconnection). Dentre as propostas, destacam-se duas abordagens mais difundidas: uma composta por três camadas e outra por cinco camadas (ANSHUMAN, PAWANI e MADHUSANKA, 2019).

A arquitetura de três camadas é constituída pelas camadas de Percepção, Rede e Aplicação. A camada de Percepção corresponde à camada mais baixa da arquitetura e tem como função interagir com o ambiente físico por meio de sensores, realizando a aquisição, o pré-processamento e o envio das informações para as camadas superiores. A camada de Rede é responsável pelo roteamento e pela transmissão dos dados por meio de diferentes tecnologias de comunicação, enquanto a camada de Aplicação fornece os serviços destinados aos usuários finais, baseando-se nos dados processados pelas camadas inferiores. (WU, LU, *et al.*, 2010).

A arquitetura de cinco camadas mantém as camadas já mencionadas e acrescenta duas novas: Camada de Middleware e Camada de Negócios. A camada de Middleware situa-se entre as camadas de rede e aplicação, sendo responsável por receber as informações provenientes da camada de rede, organizá-las para processamento e as armazená-las em bancos de dados para posterior análise. Por sua vez, a camada de negócio é encarregada do gerenciamento de todo o sistema IoT, da geração de representações visuais dos dados processados e do tratamento de aspectos relacionados à privacidade do usuário (WU, LU, *et al.*, 2010).

A Internet das Coisas está presente nos mais diversos campos, sendo a agricultura um dos exemplos mais notáveis. O monitoramento e aquisição dos dados ambientais auxiliam na tomada de decisões de curto e longo prazo, como na previsão de produtividade e na detecção precoce de doenças que possam se disseminar nas

lavouras, possibilitando a adoção de medidas preventivas em tempo hábil (ANSHUMAN, PAWANI e MADHUSANKA, 2019).

Outro campo amplamente impactado pela IoT é o setor industrial, contexto em que o conceito é conhecido como Industrial Internet of Things (IIoT). Neste cenário, é possível interconectar os ativos industriais por meio diversos protocolos de comunicação, aliados a tecnologias como Inteligência Artificial e análise *Big Data,* a fim de facilitar a supervisão dos processos produtivos e otimizar a produtividade reduzindo custos operacionais e de manutenção (ANSHUMAN, PAWANI e MADHUSANKA, 2019).

O advento dos dispositivos baseados em Internet das Coisas trouxe uma série de benefícios, como o aumento dos níveis de automação, a possibilidade de controle remoto de equipamentos, a comunicação mais rápida e eficiente entre sistemas, além da melhoria na coleta e análise de dados em tempo real (MADAKAM SOMAYYA, 2015).

2.2 ESP32

O ESP32 consiste em um sistema em um único chip (*System on Chip* - SoC), ou seja, um circuito integrado que reúne em um único chip os componentes principais de um sistema computacional, tais como CPU, memória, interfaces de comunicação e periféricos. Desenvolvido pela *Espressif Systems*, o ESP32 permite a criação de sistemas embarcados compactos, eficientes e com baixo consumo de energia (ESPRESSIF, 2019).

O primeiro modelo do ESP32 foi lançado em 2016 e destacou-se rapidamente pela combinação de recursos avançados, baixo custo e alta versatilidade. Sua arquitetura é baseada em processadores Xtensa LX6, Xtensa 32-bit e, em versões mais recentes, RISC-V 32-bit (MAIER, SHARP e VAGAPOV, 2017).

O ESP32 integra conectividade Wi-Fi 802.11 b/g/n e Bluetooth, incluindo Bluetooth Low Energy (BLE), o que o torna altamente adequado para aplicações em Internet das Coisas. Considerado o sucessor do ESP8266, apresenta melhorias

significativas, como processador dual-core e frequência de clock de até 240 MHz, variando conforme a versão (BABIUCH, FOLTýNEK e SMUTNý, 2019).

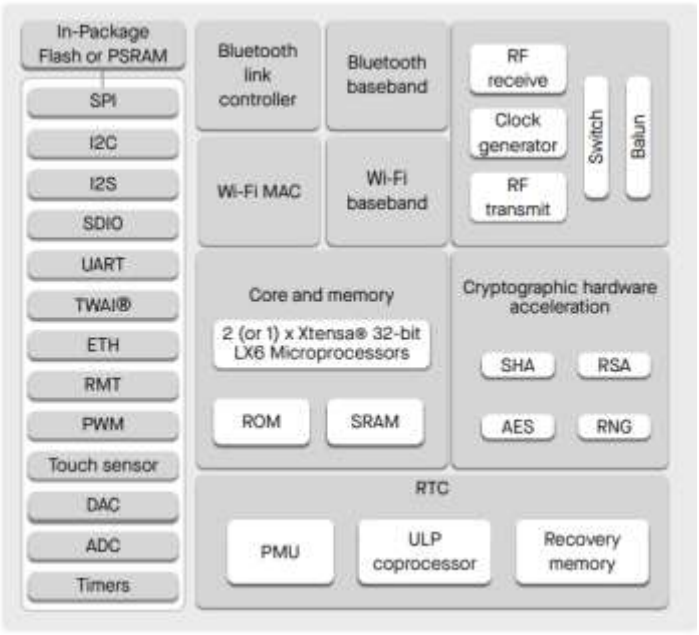
A estrutura interna do ESP32 é composta por dois núcleos de CPU, denominados PRO_CPU e APP_CPU, que podem ser controlados de forma independente. Todos os blocos de memória e periféricos estão conectados ao barramento de dados e instruções. Além disso, o ESP32 dispõe de 520 KB de SRAM, 448 KB de ROM e duas memórias RTC de 8 KB, utilizadas para operação em modos de baixo consumo energético (MAIER, SHARP e VAGAPOV, 2017).

Ademais, o chip oferece suporte a diversos protocolos e interfaces de comunicação, como SPI, I2S, I2C, CAN, UART e Ethernet MAC, dependendo da placa utilizada. Entre os periféricos integrados, destacam-se o sensor de efeito Hall, o sensor de temperatura e os sensores touch (ESPRESSIF, 2019).

O ESP32 também conta com aceleradores de hardware para criptografia, oferecendo suporte a diversos algoritmos, incluindo gerador de números aleatórios (RNG), SHA-2, RSA, *Elliptic Curve Cryptography* (ECC) e AES (BABIUCH, FOLTýNEK e SMUTNý, 2019).

A Figura 1 apresenta a estrutura interna do ESP32, evidenciando seus principais blocos funcionais, como os núcleos de processamento, módulos de comunicação sem fio e aceleradores criptográficos.

Figura 1 - Estrutura interna do ESP32



Fonte: retirado de (ESPRESSIF, 2019)

As placas baseadas no ESP32 podem ser aplicadas em uma ampla variedade de áreas, incluindo automação residencial e industrial, *wearables*, casas inteligentes, e sistemas IoT conectados à nuvem, demonstrando sua flexibilidade e relevância no contexto dos sistemas embarcados modernos (MAIER, SHARP e VAGAPOV, 2017).

2.3 ESP-NOW

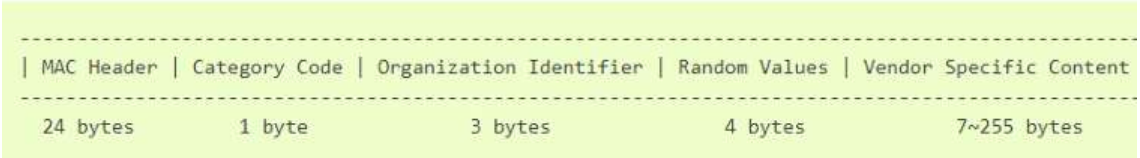
O ESP-NOW consiste em um protocolo de comunicação sem fio desenvolvido pela *Espressif Systems*, projetado para operar com Wi-Fi em modo *peer-to-peer*, ou seja, sem a necessidade de um ponto de acesso. Esse protocolo possibilita comunicações com baixo consumo de energia, baixa latência e alto rendimento entre dispositivos ESP32 (ESPRESSIF, 2025).

Diferentemente do Wi-Fi tradicional, o ESP-NOW dispensa o processo de associação e autenticação com roteadores, o que reduz significativamente o tempo de inicialização da comunicação (PASIC, KUZMANOV e ATANASOVSKI, 2021). As mensagens são transmitidas por meio de quadros de ação, um tipo específico de

quadro definido na camada MAC do protocolo Wi-Fi. Nesses quadros, os dados da aplicação são encapsulados e transmitidos diretamente de um dispositivo para outro, sem a necessidade de estabelecer uma conexão formal. Exemplos de aplicação incluem dispositivos de controle remoto, sensoriamento e uso luzes inteligentes (ESPRESSIF, 2025).

O quadro utilizado pelo ESP-NOW segue o formato de um *Vendor-Specific Action Frame*, um tipo de quadro reservado a fabricantes para a implementação de extensões proprietárias (ESPRESSIF, 2025). A Figura 2 ilustra a estrutura geral desse quadro.

Figura 2 - Quadro Vendor-Specific Action



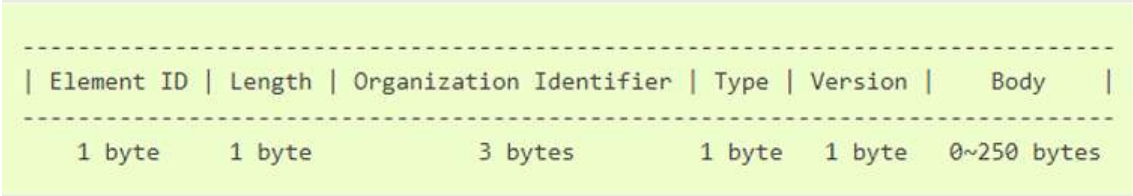
Fonte: retirado de (ESPRESSIF, 2025)

A estrutura do Vendor-Specific Action Frame utilizado pelo ESP-NOW é composta pelos seguintes campos:

- Category Code: definido com o valor 127, indicando que se trata de uma categoria específica do fabricante.
- Organization Identifier: contém um identificador exclusivo da Espressif, representado pelos três primeiros bytes do endereço MAC.
- Random Values: utilizados para prevenir ataques de repetição durante a transmissão.
- Vendor Specific Content: campo que contém um ou mais elementos específicos do fornecedor, contendo as informações do protocolo ESP-NOW propriamente dito.

A estrutura detalhada do campo Vendor Specific Content pode ser vista na Figura 3.

Figura 3 - Estrutura do Vendor Specific Content



Fonte: retirado de (ESPRESSIF, 2025)

A estrutura do campo Vendor Specific Content é composta pelos seguintes elementos:

- Element ID: definido como 221, valor reservado para elementos específicos de fornecedor.
- Length: indica o comprimento total do campo, incluindo o Organization Identifier, Type, Version e Body; o valor máximo permitido é de 255 bytes.
- Organization Identifier: contém um identificador exclusivo da Espressif, representado pelos três primeiros bytes do endereço MAC.
- Type: definido como 4, indicando que o conteúdo pertence ao protocolo ESP-NOW.
- Version: especifica a versão do protocolo ESP-NOW utilizada.
- Body: contém os dados da aplicação que serão transmitidos entre os dispositivos.

Por padrão, o bit rate de transmissão dos quadros ESP-NOW é de 1 Mbps, garantindo a comunicação estável em ambientes de curta distância e com baixo consumo energético (PASIC, KUZMANOV e ATANASOVSKI, 2021).

O ESP-NOW emprega mecanismos internos de criptografia simétrica baseados no algoritmo AES-128, fornecidos pela plataforma ESP32, com o objetivo de garantir a confidencialidade e a integridade das mensagens trocadas entre os dispositivos. Embora esses mecanismos sejam conceitualmente semelhantes aos utilizados no CCMP, o ESP-NOW não implementa integralmente o modelo de segurança definido no padrão IEEE 802.11i. (PASIC, KUZMANOV e ATANASOVSKI, 2021).

Cada ESP32 possui uma Primary Master Key (PMK) e uma ou mais Local Master Keys (LMK), que permitem estabelecer conexões seguras com os dispositivos alvo.

Os comprimentos da PMK e da LMK são de 16 bytes. A PMK é usada para criptografar a LMK por meio do algoritmo AES-128, enquanto a LMK é empregada na criptografia dos quadros de ação transmitidos entre os dispositivos (ESPRESSIF, 2025)

Além disso, o protocolo oferece suporte à criptografia e descriptografia automáticas, ao envio de cargas de dados (*payloads*) de até 250 bytes, bem como à utilização de funções de retorno (*callbacks*), que são executadas automaticamente pelo sistema para informar ao desenvolvedor o sucesso ou a falha de cada transmissão (ESPRESSIF, 2025).

2.4 Flash Encryption

Nos dispositivos ESP32, o firmware enviado é geralmente armazenado na memória flash externa do dispositivo. Essa arquitetura apresenta um risco de segurança, uma vez que um atacante poderia remover a memória e realizar a leitura de seu conteúdo com auxílio de um dispositivo externo. Mesmo em versões com memória integrada, como a ESP32-PICO, ainda é possível exportar o conteúdo da flash usando ferramentas adequadas (EMBARCADOS, 2020).

Para mitigar este risco, o ESP32 dispõe de um recurso denominado criptografia da flash (*flash encryption*). Quando habilitado, esse recurso faz com que o firmware em texto claro (*plaintext)* seja criptografado durante a primeira inicialização. Dessa forma, embora leituras físicas possam ser realizadas, o conteúdo obtido estará cifrado, sendo impossível interpretá-lo sem a chave correspondente (SABBATINI, 2024).

Quando a criptografia da memória flash é ativada, alguns blocos são criptografados por padrão, sendo eles: o segundo estágio do *bootloader*, a tabela de partição, a partição da chave NVS, os dados de atualização OTA e todas as partições de aplicação. Opcionalmente, outros blocos também podem ser criptografados, como as partições marcadas com a flag *encrypted* e o resumo criptográfico (*digest*) do *bootloader* utilizado no processo de inicialização segura (*Secure Boot*), mecanismo responsável por verificar a autenticidade e a integridade do *firmware* antes de sua execução (ESPRESSIF, 2024).

A criptografia da memória flash pode ser habilitada em dois modos distintos: modo de desenvolvimento (*Development Mode*) e modo de produção (*Release Mode*). O

modo de desenvolvimento (*Development Mode*) é recomendado para as fases de implementação e testes do software. Nesse modo, ainda é possível enviar firmware em texto claro (*plaintext*) para o dispositivo, sendo o *bootloader* responsável por criptografá-lo automaticamente, utilizando uma chave armazenada em um dos eFuses do dispositivo. Dessa forma, novos firmwares em texto claro podem ser regravados e criptografados pelo hardware durante o primeiro processo de inicialização (*boot*) (ESPRESSIF, 2024).

O *Release Mode*, por sua vez, é indicado para ambientes de produção e manufatura, após a conclusão de todos os testes. Nesse modo, o envio de firmware em *plaintext* para o dispositivo sem conhecimento prévio da chave já não é mais permitido (ESPRESSIF, 2024).

Durante o processo de criptografia da flash, são utilizados uma série de eFuses que controlam o comportamento do sistema. A documentação da Espressif descreve os principais eFuses relacionados ao processo, conforme apresentado na Tabela 1.

Tabela 1 – Principais eFuses relacionados à criptografia da flash

eFuse	Descrição	Bit Depth
CODING_SCHEME	Controla o número de bits do bloco 1 usados para gerar a chave AES de 256 bits. Valores possíveis: 0 para 256 bits, 1 para 192 bits, 2 para 128 bits. A chave AES é derivada baseada no valor de FLASH_CRYPT_CONFIG	2
flash_encryption (block1)	Armazena a chave AES	256
FLASH_CRYPT_CONFIG	Controla o processo de criptografia AES	4
DISABLE_DL_ENCRYPT	Caso habilitado, desabilita a operação de criptografia da flash enquanto o	1

eFuse	Descrição	Bit Depth
	firmware está no modo Download	
DISABLE_DL_DECRYPT	Caso habilitado, desabilita a descriptografia enquanto o firmware está no modo Download via UART.	1
FLASH_CRYPT_CNT	Um número que indica se o conteúdo da flash está criptografado. - Se um número impar de bits está habilitado (Ex: 0b0000001 ou 0b0000111), isto indica que o conteúdo da flash está criptografado. O conteúdo precisará ser descriptografado de forma transparente quando lido. - Se um número par de bits está habilitado (Ex: 0b0000000 ou 0b0000011), isto indica que o conteúdo da flash não está criptografado.	7

Fonte: retirado de (ESPRESSIF, 2024).

Durante o primeiro reset, os dados armazenados na memória flash ainda não se encontram criptografados. O processo inicia-se quando o primeiro estágio do bootloader (ROM) carrega o segundo estágio. Este, por sua vez, realiza a leitura do

eFuse FLASH_CRYPT_CNT, cujo valor inicial é 0b0000000. Esse eFuse, conforme indicado na Tabela 1, é responsável por indicar se a criptografia da flash está habilitada. Além disso, o segundo estágio também configura o eFuse FLASH_CRYPT_CONFIG com o valor 0xF (ESPRESSIF, 2024).

Se a criptografia estiver habilitada, o sistema verifica o eFuse flash_encryption para determinar se há uma chave válida gravada. Caso não exista, o módulo RNG gera uma nova chave AES de 256 bits, gravando-a nesse eFuse (ESPRESSIF, 2024).

Alternativamente o próprio usuário gerar e gravar sua chave manualmente, utilizando para isto ferramentas como o *OpenSSL*. A chave é protegida contra leitura e escrita, não podendo ser acessada por meio de software. Dessa forma, toda a operação de criptografia é executada diretamente pelo hardware, utilizando a chave armazenada internamente (EMBARCADOS, 2020)

Após a verificação da chave, o bloco de criptografia da flash é responsável por cifrar o conteúdo do segundo estágio do *bootloader*, das aplicações e das partições marcadas com a flag *encrypted*. Em seguida, o primeiro bit do eFuse FLASH_CRYPT_CNT é ativado, indicando que a criptografia foi realizada com sucesso (ESPRESSIF, 2024).

No *Development Mode*, os eFuses DISABLE_DL_DECRTPT e DISABLE_DL_CACHE são ativados, porém sem proteção contra escrita. Já no modo Realease, os bits dos eFuses DISABLE_DL_ENCRYTPT, DISABLE_DL_DECRTPT e DISABLE_DL_CACHE são ativados, prevenindo o *bootloader* UART de descriptografar conteúdo da flash. Além disso, a proteção contra leitura e escrita é habilitada para os bits do eFuse FLASH_CRYPT_CNT (ESPRESSIF, 2024).

Por fim, o dispositivo é reiniciado e passa a executar imagem criptografada. Durante o processo de inicialização, o segundo estágio do bootloader aciona o bloco de descriptografia da flash, que realiza a decodificação do conteúdo em tempo real, carregando-o na IRAM. Esse mecanismo garante a execução segura do firmware, preservando sua confidencialidade e integridade (ESPRESSIF, 2024).

2.5 Atualizações Over The Air.

O advento das atualizações *Over The Air* trouxe a possibilidade de um dispositivo solicitar a um servidor remoto novas imagens de firmware, baixá-las e atualizar automaticamente o sistema para uma nova versão. Esse conceito surgiu no final do século XX, inicialmente voltado à atualização de firmwares de computadores pessoais, processo que até então exigia intervenção manual e conhecimento técnico especializado (MEDIUM, 2025).

Com a popularização dos dispositivos móveis no início dos anos 2000, motivada especialmente pelo lançamento do primeiro iPhone e do sistema operacional Android, tornou-se evidente a necessidade de mecanismos que permitissem a atualização remota, segura e automatizada de software. Nesse contexto, atualizações OTA passaram a ganhar destaque, consolidando-se como um novo padrão no processo de manutenção e aprimoramento de sistemas, ao proporcionarem um aumento de segurança e eficiência das atualizações de sistemas (MEDIUM, 2025).

A implementação de atualizações Over The Air requer uma infraestrutura em nuvem que contemple um conjunto de características essenciais. Entre elas, destacam-se a capacidade de gerenciar pacotes de forma a garantir sua integridade e segurança, coordenação da sequência de atualizações, registro de atualizações, além da coleta de feedbacks dos usuários (MEDIUM, 2025).

Em geral, desenvolvedores criam seus firmwares e os criptografam antes de armazená-los em nuvem, isto é feito para oferecer uma maior camada de segurança contra acessos não autorizados e adulterações. Quando armazenado, o firmware passa a ser acessado por meio de redes sem fio, usualmente WiFi ou 4G/5G. Após o download da nova imagem, o dispositivo realiza a descriptografia e validação do conteúdo, verificando aspectos como integridade, autenticidade e versão do firmware.

Caso a verificação seja bem sucedida, a nova versão é instalada no dispositivo e o sistema é reiniciado para que a atualizações entre em vigor (ANDRÁS, 2020).
Embora microcontroladores possuam recursos limitados, estes também conseguem fazer atualizações via OTA de forma muito similar aos dispositivos convencionais (ANDRÁS, 2020). Um dos métodos para estabelecer comunicação segura com o servidor é por meio do Transport Layer Security (TLS), um protocolo baseado em criptografia assimétrica que oferece como benefícios criptografia, autenticação e integridade durante a comunicação de suas aplicações. Durante o processo de atualização, após o estabelecimento da conexão segura com o servidor, o dispositivo armazena o firmware na memória como um arquivo binário, sendo verificada sua validade e integridade antes da atualização. Durante o handshake TLS, o servidor apresenta seu certificado digital, e o dispositivo realiza a validação do domínio ou do endereço IP contido no campo *Common Name* do certificado. Como o TLS reside em um esquema de assinatura digital e criptografia assimétrica, um par de chaves deve ser gerado para que a atualização seja possível (KRAWCZYK, PATERSON e WEE, 2013).

A *Espressif* permite que seus dispositivos realizem atualizações OTA com suporte a dois modos: Modo de atualização segura e Modo de atualização Inseguro. O modo de atualização seguro foi projetado para operar de forma resiliente, uma vez que caso ocorra uma queda de energia durante uma atualização o chip ainda permanecerá operacional, sendo capaz de inicializar a aplicação atual. Para assegurar essa resiliência, a tabela de partições deve conter, no mínimo, duas partições OTA, ota_0 e ota_1, além de uma partição de dados OTA, que armazena informações sobre qual partição será inicializada. Assim, caso a nova imagem não seja validada, o sistema pode reverter automaticamente à versão previamente funcional do firmware (ESPRESSIF, 2025).

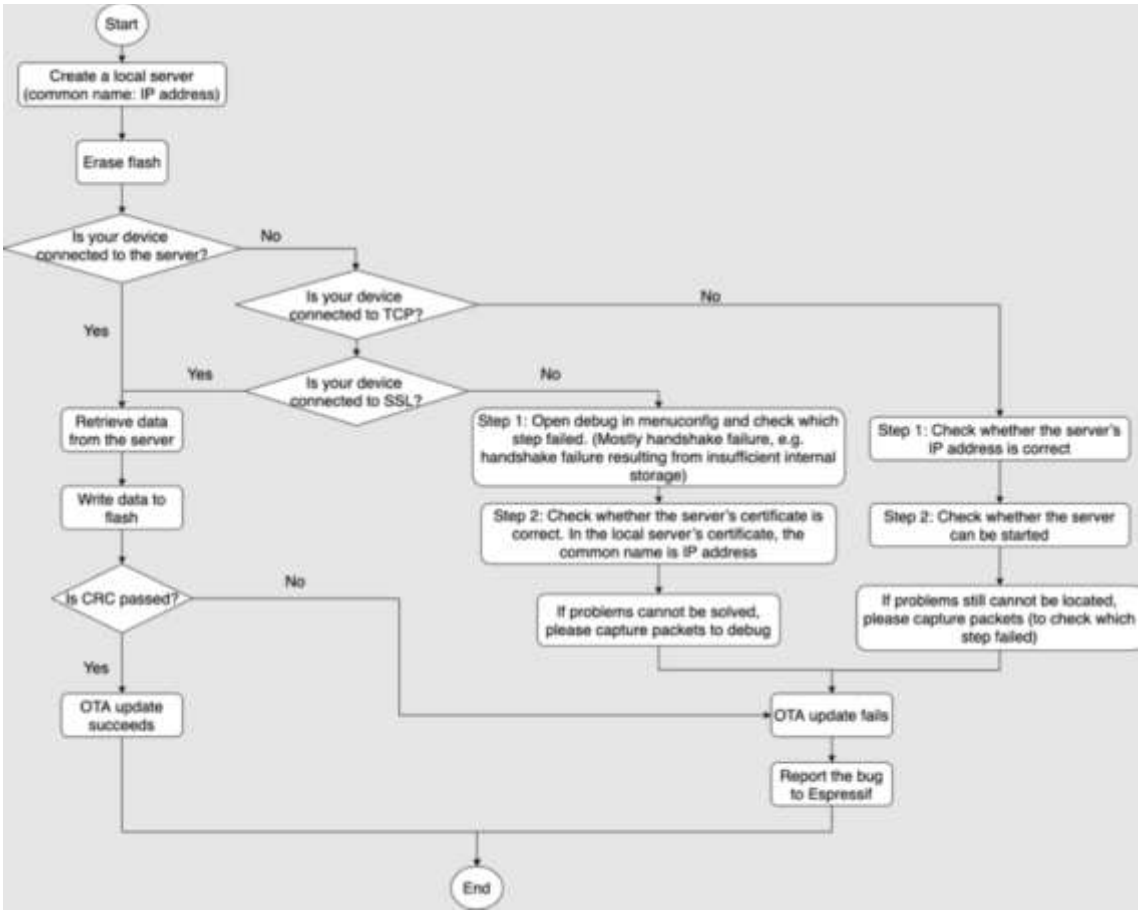
O firmware é gravado na partição inativa, por exemplo, ota_1 se ota_0 estiver em execução. Após a gravação e validação bem-sucedida, a partição ativa é atualizada no registro de dados OTA, e o sistema é reiniciado na nova versão.
A partição de dados OTA utiliza dois setores de 8192 bytes cada, que são escritos de forma independente para evitar corrupção em caso de falha de energia. Para isso, utiliza-se um contador para determinar qual setor foi gravado mais recentemente,

garantindo integridade dos dados durante o processo de atualização (ESPRESSIF, 2025).

No modo de atualização inseguro, por sua vez, a nova imagem é baixada para uma partição temporária e, após o download, é copiada para a partição final. Caso ocorra uma interrupção nesse processo, como falha no fornecimento de energia, o firmware pode ser corrompido, resultando em falha irreversível na inicialização do chip. As partições envolvidas incluem a de bootloader, tabela de partição e partições de dados como NVS ou FAT (ESPRESSIF, 2025).

O processo completo de atualização OTA da *Espressif* pode ser visualizado no fluxograma apresentado na Figura 4, que ilustra o caminho lógico desde a criação do servidor local, verificação final de integridade (CRC) e etapas de depuração recomendadas pela *Espressif*.

Figura 4 - Fluxograma de depuração do OTA



Fonte: retirado de (ESPRESSIF, 2025)

2.6 Secure Boot

O *Secure Boot* é um mecanismo de segurança que assegura que apenas código autorizado seja executado no microcontrolador. Durante cada inicialização, os dados carregados da memória flash são verificados antes da execução (ANDRÁS, 2020).

Para habilitar o Secure Boot, é necessário implementar o conceito de cadeia de confiança (*chain of trust*). Nessa cadeia, cada estágio de inicialização valida a integridade do estágio seguinte por meio de assinaturas digitais. O primeiro elemento confiável dessa cadeia é denominado Root of Trust, que é implementado por meio dos componentes de hardware, como memórias somente de leitura (ROM) ou memórias de programação única (eFuse). Esses elementos armazenam permanentemente chaves criptográficas, que não podem ser modificadas ou acessadas por software (ANDRÁS, 2020).

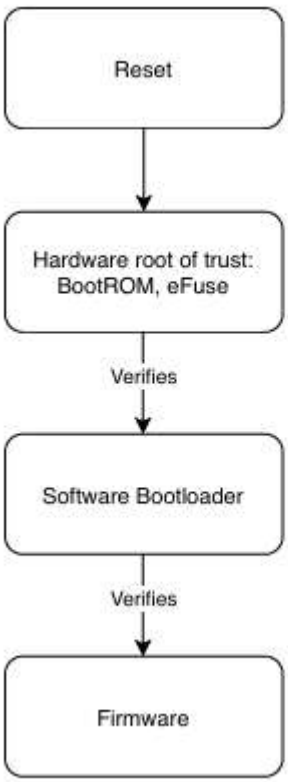
A chave armazenada nos eFuses é utilizada para validar o estágio seguinte, cuja imagem é assinada digitalmente com essa chave. Quando necessário, a cadeia de confiança pode ser estendida, permitindo que cada componente verifique a integridade do estágio subsequente. Esse fluxo modular assegura que a execução do sistema ocorra apenas se todos os estágios forem validados com sucesso (ESPRESSIF, 2024).

A Figura 5 apresenta o fluxograma do processo de verificação da cadeia de confiança do Secure Boot.

Os dados críticos relacionados ao Secure Boot são armazenados em eFuses internos, inacessíveis ao software.
O bloco 2 dos eFuses é destinado ao armazenamento da chave AES de 256 bits utilizada pelo Secure Boot. Um bit de controle, denominado ABS_DONE_0, é queimado como indicação de que o Secure Boot foi ativado. Uma vez configurado, esse bit não pode ser revertido (ESPRESSIF, 2024).

O Secure Boot, por padrão, assina imagens e dados contidos na tabela de partição durante o build. As chaves utilizadas para assinatura e verificação consistem em um par ECDSA no formato PEM. A chave privada é informada no arquivo de configuração (*menuconfig*) e utilizada para assinar as imagens, enquanto a chave pública é incorporada ao bootloader para realizar a verificação (ESPRESSIF, 2024).

Figura 5 - Chain of Trust do Secure Boot



Fonte: retirado de (ANDRÁS, 2020)

Durante o processo de compilação, a imagem do bootloader de segundo estágio é gerada e armazenada no offset 0x1000 da memória flash (ESPRESSIF, 2024). Na primeira inicialização, o bootloader segue o seguinte fluxo para ativar o Secure Boot:

- O suporte de hardware do Secure Boot gera uma chave AES de 256 bits e um digest associado.
- A chave é gerada com auxílio do gerador de números aleatórios (RNG) do hardware e armazenado no eFuse, com proteção contra leitura e escrita habilitada.
- O digest é derivado a partir da chave, de um vetor de inicialização (IV) e do conteúdo da imagem do bootloader, sendo armazenado no offset 0x0 da flash.

Após essa etapa, o bootloader configura o bit ABS_DONE_0, ativando o Secure Boot de forma permanente. Dessa maneira, o dispositivo só poderá inicializar imagens cujo digest corresponda ao valor armazenado (ESPRESSIF, 2024).

Nas inicializações subsequentes, o bootloader de primeiro estágio detecta que o bit ABS_DONE_0 foi queimado e passa a comparar o digest armazenado no

offset 0x0 com o digest calculado no momento do boot. Essa comparação é realizada integralmente por hardware (ESPRESSIF, 2024).

Enquanto o Secure Boot estiver ativo, o bootloader de segundo estágio usará a chave pública ECDSA incorporada para verificar a assinatura digital das imagens de aplicação e da tabela de partições antes de carregá-las na memória (ANDRÁS, 2020).

2.7 Assinaturas Digitais

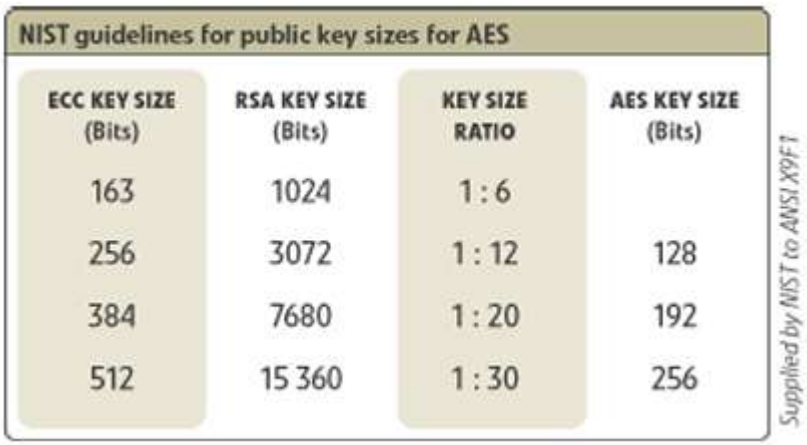
Uma assinatura, em contexto jurídico, representa o ato de concordância de uma parte em relação às condições impostas por outra, servindo como uma comprovação da aceitação dos termos propostos (BARROS, 2015). A assinatura digital é uma aplicação do campo da criptografia, desenvolvida para atuar como equivalente digital de assinaturas manuscritas, podendo oferecer propriedades adicionais de segurança. Esse mecanismo consiste em um valor numérico cuja geração depende de dois componentes principais: uma chave privada, as vezes referida como segredo do signatário, e o conteúdo de mensagem a ser assinada. Qualquer alteração em um desses elementos resulta em uma assinatura distinta, assegurando características fundamentais da segurança da informação, como integridade, autenticidade e não repúdio (JOHNSON, MENEZES e VANSTONE, 2013).

É requerida que assinaturas digitais sejam verificáveis. Dessa forma, caso ocorra algum impasse quanto à participação de determinada entidade na assinatura de um documento, deve existir um terceiro que, de maneira imparcial, possa analisar a autenticidade da assinatura sem a necessidade de acesso à chave privada do signatário. Um dos esquemas mais comuns de assinaturas digitais é o esquema de criptografia assimétrica. Neste método, cada participante gera um par de chaves composto por uma chave privada e uma chave pública. A chave privada deve ser mantida em sigilo e é utilizada para gerar a assinatura digital, enquanto a chave pública é distribuída para que outras entidades possam verificar a validade das assinaturas geradas. (JOHNSON, MENEZES e VANSTONE, 2013).

Tradicionalmente, algoritmos de criptografia assimétrica e de assinatura digital, como o RSA e o DAS, utilizam chaves com um grande número de bits. Essa característica resulta em maior consumo de recursos computacionais, redução de desempenho e aumento na complexidade de implementação e compatibilidade. Para contornar essas limitações, Victor Miller e Neal Koblitz propuseram, em 1985, um novo algoritmo de criptografia baseado em curvas elípticas, conhecido como Eliptic Curve Cryptography (ECC). Esse tipo de algoritmo utiliza chaves menores, porém oferecem níveis de segurança equivalentes aos métodos tradicionais, proporcionando maior agilidade no processamento e menor consumo de recursos. Por essa razão, a ECC é recomendada para dispositivos com recursos limitados, como é o caso de microcontroladores (MOREIRA, 2006).

A Figura 6 apresenta um comparativo entre o tamanho das chaves utilizadas em um algoritmo tradicional (RSA e AES) e no ECC.

Figura 6 - Comparativo entre o tamanho das chaves usados no RSA, AES e ECC



Fonte: retirado de (MOREIRA, 2006)

No algoritmo ECC são usados pontos definidos sobre uma curva elíptica em um campo finito, os quais são empregados na geração dos componentes da assinatura digital (MOREIRA, 2006).

3 METODOLOGIA

3.1 Arquitetura do Sistema

O sistema de segurança proposto foi planejado para ser implementado utilizando dois microcontroladores ESP32, configurados de forma que cada um desempenhe uma função específica dentro da arquitetura desenvolvida. A escolha do ESP32 se justifica pelo dispositivo integrar módulos de Wi-Fi e Bluetooth, além de aceleradores de criptografia em hardware. Tais recursos facilitam a implementação de criptografia com maior eficiência e menor consumo de energia. Além disso, o ESP32 também oferece suporte a outros recursos de segurança como Secure Boot e Flash Encryption.

O primeiro dispositivo do sistema atua como emissor sendo encarregado de gerar, assinar digitalmente e criptografar as mensagens antes da transmissão. O segundo dispositivo, por sua vez, opera como receptor, sendo responsável pela recepção, verificação da assinatura e descriptografia das mensagens recebidas.

A comunicação entre os dispositivos é realizada por meio do protocolo ESP-NOW, o qual foi escolhido por oferecer baixa latência, eficiência energética e mecanismos nativos de criptografia simétrica baseados no algoritmo AES, fornecidos pela plataforma ESP32 e adequados para aplicações embarcadas e de Internet das Coisas.

Além da criptografia, a arquitetura também faz uso de assinaturas digitais ECDSA para assegurar a autenticidade e a integridade das mensagens trocadas. Antes que uma mensagem seja aceita pelo receptor, sua assinatura é verificada utilizando a chave pública correspondente, garantindo que os dados realmente foram enviados por um dispositivo legítimo e que não sofreram alterações durante o processo de transmissão.

3.2 Mecanismos de Segurança do Firmware

Foram aplicadas medidas de proteção diretamente relacionadas ao firmware e à memória não volátil do ESP32. Essas medidas tiveram como objetivo garantir que

apenas códigos legítimos fossem executados e que os dados armazenados permanecessem confidenciais, mesmo em caso de acesso físico ao dispositivo.

O primeiro procedimento consistiu na ativação da criptografia da flash, funcionalidade nativa do ESP32 responsável por criptografar automaticamente o conteúdo gravado na memória flash. A configuração foi feita de forma manual usando o utilitário espsecure.py. A chave utilizada no processo foi gerada utilizando a ferramenta openssl, posteriormente a chave foi armazenada de forma segura no eFuse.

Em seguida, foi habilitado o *Secure Boot*, realizado por meio das ferramentas de configuração do ESP-IDF. O processo envolveu a geração de uma chave privada utilizada para assinar digitalmente o firmware e a posterior gravação da chave pública correspondente na partição de boot do microcontrolador

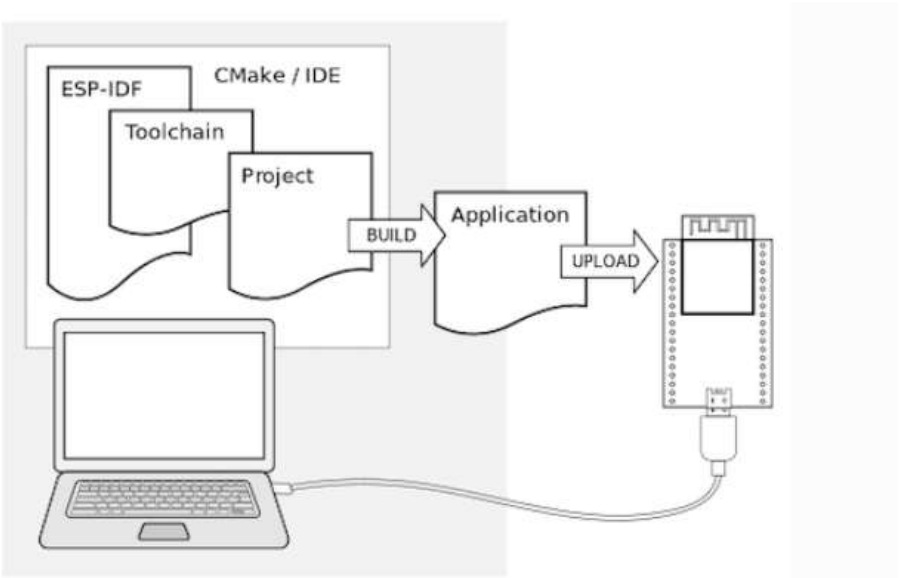
Por fim, foi implementado o mecanismo de atualização remota via OTA. O firmware foi configurado para aceitar apenas imagens assinadas digitalmente, de modo que o dispositivo realizasse a verificação da assinatura antes da substituição da partição ativa.

3.3 Ambiente de Desenvolvimento

A implementação do sistema foi realizada utilizando o *Espressif IoT Development Framework*, traduzido como *framework* de desenvolvimento para Internet das Coisas, abreviado como ESP-IDF, que consiste no ambiente oficial fornecido pela *Espressif Systems* para a programação de microcontroladores da família ESP32. O ESP-IDF oferece suporte nativo a diversas bibliotecas, multitarefa por meio do FreeRTOS e múltiplos recursos de segurança, o que o torna adequado para aplicações embarcadas que exigem confiabilidade e eficiência.

A Figura 7 apresenta o fluxo geral do processo de desenvolvimento e gravação da aplicação no microcontrolador ESP32. Esse processo inicia-se no ambiente de desenvolvimento ESP-IDF, que inclui um conjunto de ferramentas, o sistema de build baseado em CMake e o gerenciamento do projeto. Após a etapa de build, o código é convertido em um binário que é então feito seu upload para o dispositivo.

Figura 7 - Fluxo de gravação do firmware



Fonte: retirado de (ESPRESSIF, 2025)

Para as rotinas de segurança, o projeto faz uso da biblioteca *mbedTLS*, incorporada ao ESP-IDF. Essa biblioteca fornece uma ampla gama de algoritmos criptográficos padronizados, como AES, SHA-256 e ECDSA, que foram utilizados para garantir confidencialidade, integridade e autenticação durante a comunicação entre os dispositivos. O *mbedTLS* também oferece suporte a aceleração de hardware, aproveitando os recursos internos do ESP32 para otimizar o desempenho das operações criptográficas.

O sistema foi desenvolvido e testado em ambiente Windows 11, utilizando o terminal ESP-IDF CMD para execução de comandos e monitoramento em tempo real. Além disso, foram empregados recursos de depuração e monitoramento serial para análise do comportamento do sistema durante as fases de teste e validação.

3.4 Procedimento de Implementação

A implementação do sistema seguiu uma sequência de etapas sequenciais, abrangendo a configuração da comunicação, a geração de chaves criptográficas, o processo de assinatura digital e a criptografia das mensagens.

Inicialmente, o ambiente de comunicação foi configurado utilizando o protocolo ESP-NOW, onde foram definidos os endereços MAC dos dispositivos participantes e

o canal de comunicação, assegurando que apenas o emissor e o receptor autorizados pudessem trocar dados.

Em seguida, foi implementada a camada de segurança da comunicação. O emissor é responsável por gerar uma mensagem inicial, aplicar uma assinatura digital ECDSA com sua chave privada e, posteriormente, criptografar o conteúdo utilizando mecanismos de criptografia simétrica baseados no algoritmo AES, conceitualmente semelhantes aos empregados no CCMP. O receptor, por sua vez, realiza a verificação da assinatura digital por meio da chave pública do emissor e procede com a descriptografia do conteúdo, validando a integridade e a autenticidade da mensagem recebida. Caso a assinatura não seja validada, o receptor encerra a comunicação com o emissor, deixando de aceitar novas mensagens provenientes desse dispositivo.

Para o cálculo da assinatura digital, foram utilizadas funções baseadas na curva elíptica secp256r1, recomendada pelo NIST. A criptografia e autenticação dos blocos de dados foram realizadas por meio das rotinas nativas do ESP-NOW.

Foram realizados testes de envio e recepção entre os dispositivos, analisando o desempenho da criptografia e a validade das assinaturas, de modo a garantir o funcionamento correto de todo o processo de autenticação e transmissão segura.

Paralelamente à proteção da comunicação, medidas de segurança do firmware foram implementadas para assegurar a execução apenas de códigos legítimos e a confidencialidade dos dados armazenados. O Secure Boot foi ativado, garantindo que o microcontrolador só execute firmware assinado digitalmente. A Flash Encryption foi configurada, de modo que todo o conteúdo gravado na memória não volátil seja automaticamente criptografado, protegendo os dados mesmo em caso de acesso físico ao dispositivo. O mecanismo de atualização remota via OTA também foi habilitado, permitindo que novas versões de firmware sejam aplicadas somente se assinadas digitalmente. Durante o processo de atualização, o dispositivo realiza a verificação da assinatura antes de substituir a partição ativa, garantindo a integridade e autenticidade da nova versão do firmware.

4 DESENVOLVIMENTO DO TRABALHO

Inicialmente, para a implementação da arquitetura proposta, foi necessário realizar a instalação do conjunto de ferramentas disponibilizado pela *Espressif*. Ao acessar o site oficial, é possível baixar um pacote contendo três ferramentas principais: *Espressif IDE*, *ESP-IDF CMD* e *ESP-IDF PowerShell*. As características gerais dessas ferramentas são descritas abaixo:

- *Espressif IDE* é o ambiente de desenvolvimento integrado baseado no Eclipse, que oferece recursos de edição, compilação e depuração de projetos voltados para os microcontroladores ESP32.
- *ESP-IDF CMD* é um prompt de comando configurado com todas as variáveis de ambiente necessárias para compilar, gravar e monitorar projetos utilizando o framework ESP-IDF.
- *ESP-IDF PowerShell* fornece funcionalidades semelhantes ao CMD, mas em um ambiente PowerShell, oferecendo suporte a scripts e comandos mais avançados.

Para o desenvolvimento das aplicações deste trabalho, foi utilizado o ESP-IDF CMD, por fornecer maior versatilidade durante o desenvolvimento e testes dos códigos.

Os algoritmos foram desenvolvidos de forma gradual. Inicialmente, implementou-se a comunicação entre emissor e receptor via protocolo ESP-NOW, com a criptografia habilitada. Em seguida, foram integrados os módulos de assinatura digital ECDSA, atualização segura OTA, e, por fim, as camadas de proteção da memória flash e o *Secure Boot*.

4.1 Comunicação ESP-NOW com mecanismos de criptografia simétrica

O ESP-NOW é um protocolo que oferece suporte tanto à comunicação unidirecional quanto à bidirecional, permitindo grande flexibilidade na topologia da rede.

Em comunicação unidirecional, pode-se configurar um mestre e um escravo, vários mestres comunicando-se com um único escravo, ou vários escravos enviando dados a um mestre central. A escolha da topologia depende diretamente da aplicação. Por exemplo, um mestre com múltiplos escravos é ideal em sistemas de automação ou monitoramento distribuído, nos quais um único ESP32 central coleta informações de diversos nós sensores espalhados por diferentes locais.

Já na comunicação bidirecional, cada ESP32 pode atuar simultaneamente como emissor e receptor, possibilitando o desenvolvimento de sistemas de troca de mensagens seguras e dinâmicas entre os dispositivos.

Um dos requisitos fundamentais para o estabelecimento da comunicação via ESP-NOW é o conhecimento prévio do endereço MAC dos dispositivos participantes. Cada ESP32 possui um identificador físico único, utilizado para distinguir os nós na rede, garantindo assim que os pacotes sejam enviados ao destino correto. O trecho de código apresentado na Figura 8 mostra o procedimento utilizado para obtenção desse identificador. Nessa figura, é possível observar a chamada da função responsável por recuperar o MAC e o armazenamento do valor no buffer correspondente. Embora o exemplo tenha sido retirado do código do emissor, o mesmo procedimento pode ser aplicado a qualquer dispositivo.

Figura 8 - Função para obtenção de endereço MAC

```
uint8_t mac[6];
esp_err_t err = esp_wifi_get_mac(ESP_IF_WIFI_STA, mac);
if (err == ESP_OK) {
    ESP_LOGI(TAG, "Endereço MAC do emissor: %02X:%02X:%02X:%02X:%02X:%02X",
             mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
} else {
    ESP_LOGE(TAG, "Falha ao obter MAC: %s", esp_err_to_name(err));
}
```

Fonte: Autor, 2025

A função “esp_wifi_get_mac” é disponibilizada pela própria ESP-IDF e permite obter o endereço MAC da interface Wi-Fi configurada. O valor retornado é armazenado em um vetor de seis bytes denominado “mac”. Em seguida, esse endereço é formatado em representação hexadecimal e exibido no log por meio da

função “ESP_LOGI”. Além disso, foi implementado um tratamento de erro para o caso de falha na leitura do endereço, utilizando “ESP_LOGE” para registrar no log uma mensagem informando o tipo de erro ocorrido.

Após a obtenção do endereço MAC dos dispositivos, a implementação do ESP-NOW segue um determinado fluxo. Inicialmente, o protocolo deve ser inicializado com suas configurações básicas, de modo a preparar o módulo para o envio e o recebimento de dados.

Em seguida, são desenvolvidos blocos de função distintos para o emissor e para o receptor. Esses blocos são responsáveis pela transmissão e recepção de mensagens, bem como fornecer um retorno indicando se uma mensagem foi enviada ou recebida com sucesso.

No caso do emissor, é necessário adicionar um par, correspondente ao dispositivo receptor, informando o endereço MAC obtido previamente. As funções desenvolvidas para o envio e recepção das mensagens devem ser registradas como funções de *callback*, sendo automaticamente chamadas pelo sistema sempre que uma mensagem for transmitida ou recebida.

A Tabela 2 apresenta uma relação das principais funções utilizadas na implementação do fluxo descrito.

Tabela 2 - Principais funções OTA

Função	Descrição
esp_now_init()	Inicializar o ESP-NOW. O Wi-Fi deve ser inicializado antes da chamada desta função
esp_now_add_peer()	Emparelhar dispositivos. Deve ser passado como argumento o endereço MAC do dispositivo alvo
esp_now_send()	Enviar dados
esp_now_register_send_cb()	Registrar função de *callback* que deve ser chamada ao enviar dados

Função	Descrição
esp_now_register_recv_cb()	Registrar função de *callback* que deve ser chamada ao receber dados.

Fonte: retirado de (ESPRESSIF, 2024).

A configuração do método de criptografia utilizado pelo ESP-NOW requer a definição prévia das chaves PMK e LMK. Caso a PMK não seja configurada manualmente, o ESP-NOW utiliza uma chave padrão.

A LMK deve ser definida para permitir a criptografia dos *action frames* trocados entre os dispositivos. O protocolo suporta até seis LMKs distintas, sendo cada uma associada a um par específico de comunicação.

No fluxo de envio e recepção de mensagens via ESP-NOW, descrito anteriormente, segue-se com a definição da chave PMK do dispositivo emissor, seguido da associação da chave LMK ao dispositivo receptor. É importante destacar que a PMK deve ser idêntica em ambos os dispositivos, enquanto a LMK é exclusiva para cada par emissor-receptor.

As chaves PMK e LMK possuem comprimento de 16 bytes e foram geradas previamente pelo módulo de geração de chaves criptográficas. Esse módulo utiliza o gerador de números aleatórios CTR-DRBG (Counter-mode Deterministic Random Bit Generator), alimentado por uma fonte de entropia interna, para criar a PMK de forma segura. A LMK, por sua vez, é derivada a partir da PMK e do endereço MAC do par de comunicação.

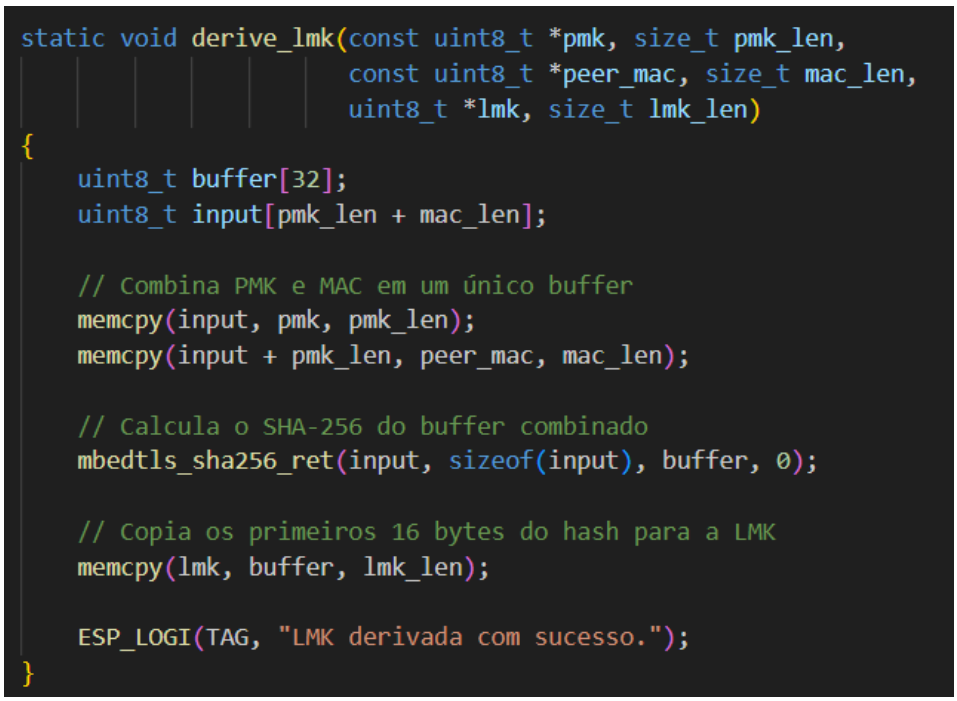
O processo de geração e derivação das chaves é realizado por meio de duas funções principais, apresentadas nos trechos de código das Figura 9 e Figura 10. Na Figura 9, observa-se a função “generate_keys”, que utiliza o gerador de números aleatórios da biblioteca *mbedTLS* para criar a PMK de 16 bytes. Já na Figura 10, é possível visualizar a função responsável pela derivação da LMK a partir da PMK. A PMK é impressa no console apenas para fins de verificação durante os testes, não devendo ser exibida em um ambiente real para evitar exposição indevida da chave.

Figura 9 - Função para geração das PKM e LMK

```
static void generate_keys(void)
{
    if (keys_generated)
        return;

    int ret = mbedtls_ctr_drbg_random(&ctr_drbg, pmk_key, sizeof(pmk_key));
    if (ret != 0)
    {
        ESP_LOGE(TAG, "Falha ao gerar PMK: -0x%04x", -ret);
        return;
    }

    ESP_LOGI(TAG, "PMK gerada com sucesso:");
    for (int i = 0; i < 16; i++)
        printf("%02x", pmk_key[i]);
    printf("\n");

    uint8_t peer_mac[6] = {0x44, 0x1d, 0x64, 0xbd, 0x2d, 0x3c};

    derive_lmk(pmk_key, sizeof(pmk_key), peer_mac, sizeof(peer_mac), lmk_key, sizeof(lmk_key));

    ESP_LOGI(TAG, "LMK derivada com sucesso:");
    for (int i = 0; i < 16; i++)
        printf("%02x", lmk_key[i]);
    printf("\n");

    keys_generated = true;
}
```

Fonte: Autor, 2025

Figura 10 - Função para derivação da LMK



```
static void derive_lmk(const uint8_t *pmk, size_t pmk_len,
                       const uint8_t *peer_mac, size_t mac_len,
                       uint8_t *lmk, size_t lmk_len)
{
    uint8_t buffer[32];
    uint8_t input[pmk_len + mac_len];

    // Combina PMK e MAC em um único buffer
    memcpy(input, pmk, pmk_len);
    memcpy(input + pmk_len, peer_mac, mac_len);

    // Calcula o SHA-256 do buffer combinado
    mbedtls_sha256_ret(input, sizeof(input), buffer, 0);

    // Copia os primeiros 16 bytes do hash para a LMK
    memcpy(lmk, buffer, lmk_len);

    ESP_LOGI(TAG, "LMK derivada com sucesso.");
}
```

Fonte: Autor, 2025

A função “derive_lmk” é responsável por derivar a LMK a partir da PMK e do endereço MAC do dispositivo pareado. O processo combina ambos os valores em um único buffer e aplica a função de hash SHA-256, extraindo os 16 primeiros bytes do resultado como a LMK. Esse método garante que cada par de dispositivos tenha uma chave única e vinculada ao seu endereço MAC.

A chave PMK do dispositivo deve ser definida antes de qualquer comunicação criptografada, sendo realizada através da função “esp_now_set_pmk”, que recebe como argumento um ponteiro para o array de bytes contendo a PMK previamente gerada.

A LMK deve ser configurada no momento em que o par é registrado utilizando “esp_now_add_peer”. Para isso, o array de 16 bytes contendo a LMK derivada deve ser copiado para o campo “lmk” da estrutura “esp_now_peer_info_t”. Além disso, é necessário habilitar a criptografia para esse par definindo o campo “encrypt” como *true*.

4.2 Implementação da assinatura ECDSA

A autenticação do emissor é realizada por meio da verificação da assinatura digital ECDSA utilizando a chave pública correspondente. A variável denominada ACK (*Acknowledgment*) é utilizada exclusivamente como um mecanismo de sinalização, responsável por indicar ao emissor o resultado do processo de verificação da assinatura. Os valores atribuídos à variável ACK representam estados de controle da comunicação, utilizados para sinalizar o resultado da verificação da assinatura digital, conforme descrito a seguir:

- ACK = 0: indicação de que a verificação da assinatura falhou;
- ACK = 1: indicação de que a assinatura foi verificada com sucesso;
- ACK = 2: estado inicial, utilizado enquanto o receptor aguarda a conclusão do processo de verificação da assinatura.

Inicialmente, o emissor envia uma mensagem de desafio ao receptor. Ao receber essa mensagem, o receptor realiza a verificação da assinatura digital ECDSA,

processo pelo qual é avaliada a autenticidade do emissor e a integridade da mensagem. O resultado dessa verificação é então sinalizado ao emissor por meio de uma variável de controle denominada ACK, utilizada exclusivamente para indicar o estado do processo. Caso a assinatura seja validada com sucesso, o receptor envia um ACK indicando sucesso, caso contrário, sinaliza falha e encerra a comunicação.

Como discutido anteriormente, o algoritmo ECDSA baseia-se em um esquema de chaves assimétricas, composto por uma chave privada (utilizada para gerar a assinatura) e uma chave pública (utilizada para verificar sua autenticidade). As chaves foram geradas externamente por meio do OpenSSL, sendo a chave privada armazenada no emissor e a pública distribuída entre os dispositivos receptores.

Para armazenar os componentes da assinatura, foi criada uma struct denominada curva_t, contendo os parâmetros r e s resultantes da assinatura, bem como a mensagem a ser assinada e enviada.

A ESP32 também utiliza a biblioteca *mbedTLS* para implementar a assinatura ECDSA. O processo de assinatura inicia com a definição dos contextos necessários:

- ecdsa: contexto principal do algoritmo de assinatura;
- entropy: responsável pela coleta de entropia para o gerador aleatório;
- ctr_drbg: gerador determinístico de números aleatórios.

Esses contextos são fundamentais, uma vez que armazenam as configurações e o estado do algoritmo durante a execução, além de permitir que a assinatura seja gerada de forma segura e imprevisível.

A função denominada “init_ecdsa**”** é responsável por inicializar os contextos necessários para o uso do ECDSA, além de configurar a variável *pers*, uma string utilizada como parâmetro na geração de números aleatórios. Essa função também carrega a curva elíptica SECP256R1 e importa a chave privada previamente gerada, armazenando-a no contexto ECDSA. A Figura 11 exibe a implementação desta função, onde destacam-se as etapas de inicialização dos contextos, a definição da string *pers* e o carregamento da chave privada.

Figura 11 - Função de inicialização do ECDSA

```c
static void init_ecdsa(void)
{
    mbedtls_ecdsa_init(&ecdsa);
    mbedtls_entropy_init(&entropy);
    mbedtls_ctr_drbg_init(&ctr_drbg);

    const char *pers = "ecdsa";
    esp_err_t ret = mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy,
                                          (const unsigned char *)pers, strlen(pers));
    if (ret != 0) {
        ESP_LOGE(TAG, "mbedtls_ctr_drbg_seed failed: -0x%04x", -ret);
    }

    mbedtls_ecp_group_load(&ecdsa.MBEDTLS_PRIVATE(grp), MBEDTLS_ECP_DP_SECP256R1);

    /* carrega a chave privada a partir do array binário */
    mbedtls_mpi_read_binary(&ecdsa.MBEDTLS_PRIVATE(d),
                            device_private_key, sizeof(device_private_key));
}
```

Fonte: Autor, 2025

Outro bloco relevante é a função “sign_and_send_challenge”, responsável por assinar e transmitir a mensagem de autenticação ao receptor. Como parâmetro, essa função recebe o endereço MAC do dispositivo de destino e a struct que contém os campos r, s e message.

Antes da assinatura, é gerado um hash SHA-256 da mensagem, visto que o ECDSA não assina o conteúdo diretamente, mas sim o seu resumo criptográfico. Esse procedimento garante que a assinatura tenha tamanho fixo e que pequenas alterações na mensagem resultem em hashes completamente distintos.

Após a geração do hash, os parâmetros r e s são inicializados e preenchidos com o resultado da operação de assinatura. Em seguida, esses valores são convertidos para formato binário e enviados ao receptor por meio do ESP-NOW. No receptor, a chave pública correspondente é utilizada para validar a assinatura e garantir a autenticidade da mensagem recebida. Na Figura 12, observam-se as etapas de conversão, transmissão e validação desses parâmetros ao longo do processo.

Figura 12 - Função de assinatura da mensagem com ECDSA

```c
static esp_err_t sign_and_send_challenge(const uint8_t *dest_mac)
{
    curva_t out;
    memset(&out, 0, sizeof(out));

    const char *challenge = "Challenge: Authenticate this device";
    strncpy((char*)out.message, challenge, sizeof(out.message)-1);
    size_t msg_len = strlen(challenge);

    unsigned char hash[32];
    mbedtls_sha256((const unsigned char*)out.message, msg_len, hash, 0);

    mbedtls_mpi r, s;
    mbedtls_mpi_init(&r); mbedtls_mpi_init(&s);

    int ret = mbedtls_ecdsa_sign(&ecdsa.MBEDTLS_PRIVATE(grp),
                                 &r, &s,
                                 &ecdsa.MBEDTLS_PRIVATE(d),
                                 hash, sizeof(hash),
                                 mbedtls_ctr_drbg_random, &ctr_drbg);
    if (ret != 0) {
        ESP_LOGE(TAG, "mbedtls_ecdsa_sign falhou: -0x%04x", -ret);
        mbedtls_mpi_free(&r); mbedtls_mpi_free(&s);
        return ESP_FAIL;
    }

    /* serializa r e s em binário 32 bytes (big-endian) */
    mbedtls_mpi_write_binary(&r, out.r_bin, 32);
    mbedtls_mpi_write_binary(&s, out.s_bin, 32);
    mbedtls_mpi_free(&r); mbedtls_mpi_free(&s);

    esp_err_t send_ret = esp_now_send(dest_mac, (uint8_t*)&out, sizeof(out));
    if (send_ret != ESP_OK) {
        ESP_LOGE(TAG, "Falha ao enviar CURVA: 0x%x", send_ret);
        return send_ret;
    }
    ESP_LOGI(TAG, "Challenge enviado.");
    return ESP_OK;
}
```

Fonte: Autor, 2025

O receptor foi implementado para verificar a autenticidade das mensagens recebidas via ESP-NOW antes de aceitar qualquer outro dado proveniente do emissor. Para isso, o código foi estruturado de modo a reconhecer automaticamente o tipo de pacote recebido. Se o payload possuir o tamanho correspondente à estrutura curva_t,

o receptor interpreta o conteúdo como uma mensagem acompanhada de assinatura digital ECDSA, caso contrário, se o tamanho corresponder à estrutura struct_message, o conteúdo é tratado como dados de aplicação.

Para validação da assinatura, o receptor realiza o cálculo do hash SHA-256 da mensagem original e utiliza a chave pública previamente cadastrada do emissor para verificar a validade da assinatura ECDSA. Se a assinatura digital for validada com sucesso, a variável interna ACK é atualizada para o valor 1, indicando o resultado positivo da verificação da assinatura e um pacote de confirmação é enviado de volta ao emissor, caso contrário, o ACK assume o valor 0 e o pacote é rejeitado.

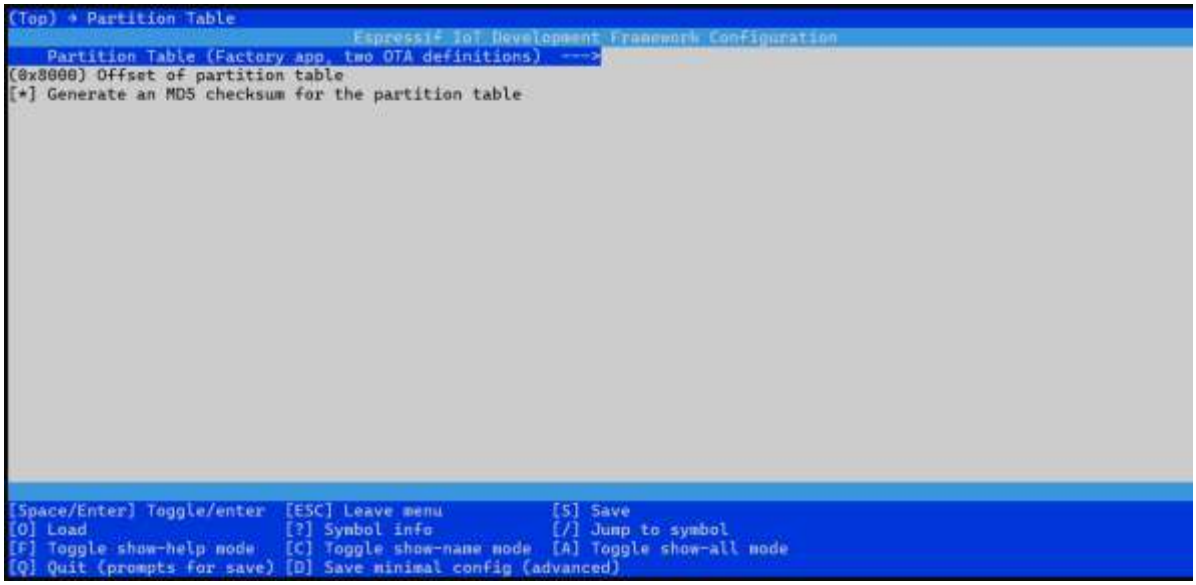
O código também realiza a verificação do endereço MAC de origem, como mais uma medida para assegurar que apenas dispositivos previamente registrados possam enviar mensagens de autenticação. Após a autenticação, o receptor passa a aceitar pacotes de dados e exibe as informações recebidas.

4.3 Implementação de atualizações OTA

A atualização OTA foi implementada utilizando um servidor HTTPS para armazenar o novo firmware. O dispositivo recebe uma URL do servidor a ser acessado e é configurado para realizar verificações periódicas em busca de novas versões do firmware, conforme o tempo definido em sua configuração.

Foi necessário configurar a tabela de partições para incluir duas partições OTA, requisito fundamental para a implementação do mecanismo de atualização. Para isso, utilizou-se o comando idf.py menuconfig a fim de acessar o arquivo de configuração do projeto. Na seção destinada à tabela de partições, selecionou-se a opção que habilita o suporte a duas partições OTA. Na Figura 13, nota-se a tela do *menuconfig* exibindo essa seleção, evidenciando a configuração aplicada.

Figura 13 - *Menuconfig* OTA



Fonte: Autor, 2025

Por se tratar de uma conexão HTTPS, é necessário o uso de um certificado digital, o qual foi gerado no formato PEM utilizando a ferramenta *OpenSSL*. Esse certificado é incorporado ao firmware e armazenado como uma string bruta, garantindo a autenticação do servidor durante o processo de atualização.

Para assegurar a integridade do firmware, foi criada a função “validate_image_header”. O propósito principal dessa função é verificar a validade das informações da nova imagem e comparar a versão do firmware disponível no servidor com a versão atualmente instalada no dispositivo. Esse bloco sempre é executado antes da aplicação de uma nova atualização.

Durante a validação, o código tenta ler a descrição da partição em execução e registra nos logs a versão atual. Caso seja detectado que a nova versão é idêntica à instalada, o processo é abortado.

Outro bloco importante é a função “ota_event_handler”, responsável por atuar como *callback* de eventos gerados durante o processo de atualização. Esses eventos permitem monitorar cada etapa do ciclo OTA, desde a conexão com o servidor até a finalização da gravação, registrando mensagens no log, o que auxilia no diagnóstico de falhas. Os principais eventos e seus propósitos estão descritos na Tabela 3.

Tabela 3 - Eventos do OTA

Evento	Descrição	Propósito
ESP_HTTPS_OTA_START	Início do processo OTA	Informa o início da atualização
ESP_HTTPS_OTA_CONNECTED	Conexão com servidor	Confirma a comunicação HTTPS
ESP_HTTPS_OTA_GET_IMG_DESC	Leitura da descrição da imagem	Obtém metadados da nova versão
ESP_HTTPS_OTA_VERIFY_CHIP_ID	Verificação do chip	Evita incompatibilidade de hardware
ESP_HTTPS_OTA_DECRYPT_CB	Descriptografia do firmware	Protege o firmware durante o download
ESP_HTTPS_OTA_WRITE_FLASH	Escrita de blocos na flash	Mostra o progresso da gravação
ESP_HTTPS_OTA_UPDATE_BOOT_PARTITION	Atualização da partição de boot	Define a próxima partição de inicialização
ESP_HTTPS_OTA_FINISH	Finalização da OTA	Indica sucesso e prontidão para reinicialização.
ESP_HTTPS_OTA_ABORT	Abortar OTA	Informa falha ou interrupção no processo

Fonte: retirado de (ESPRESSIF, 2024).

A parte principal do processo de atualização está contida na função “do_https_ota”, responsável por coordenar todo o processo de atualização via HTTPS, incluindo a

validação do certificado, verificação da versão, download da nova imagem, gravação na partição OTA e reinicialização do dispositivo. O fluxo de execução segue as seguintes etapas:

1. Inicialização do cliente HTTPS, com parâmetros como a URL do firmware, certificado do servidor e tempo limite de resposta.
2. Criação da configuração OTA, que associa o cliente HTTPS e define *callbacks*.
3. Obtenção da descrição da nova imagem, contendo versão e informações de segurança.
4. Validação da nova versão, comparando-a com o firmware em execução.
5. Download e gravação em flash da nova imagem, com logs indicando o progresso.
6. Finalização e atualização da partição de boot, seguida de reinicialização automática.

Além disso, foi criada uma *task* responsável por verificar periodicamente a disponibilidade de atualizações. Essa tarefa aguarda a conexão Wi-Fi ser estabelecida e, em seguida, executa a função “do_https_ota”. Caso nenhuma nova versão seja detectada ou ocorra erro de validação, o processo é encerrado sem reinicialização, mantendo o firmware atual em execução.

O servidor HTTPS foi criado utilizando o framework Flask, escrito em Python. Esse servidor tem a função de hospedar o arquivo binário do firmware e disponibilizá-lo de forma segura para o ESP32 durante o processo de atualização. O servidor foi configurado para operar localmente, armazenando os arquivos enviados em um diretório específico definido na variável UPLOAD_FOLDER.

A aplicação Flask implementa duas rotas principais. A primeira é responsável por receber o upload de novos firmwares através de uma interface simples em HTML. Essa interface pode ser acessada diretamente por um navegador, permitindo selecionar o arquivo de firmware com extensão .bin e enviá-lo ao servidor. Após o envio, o arquivo é salvo na pasta configurada, sendo sempre renomeado, no caso do emissor, para “EmissorOTA.bin”, o que garante que o firmware mais recente substitua automaticamente o anterior, simplificando o controle de versões. Ao concluir o upload,

o servidor retorna uma mensagem de confirmação, informando que o arquivo foi recebido com sucesso.

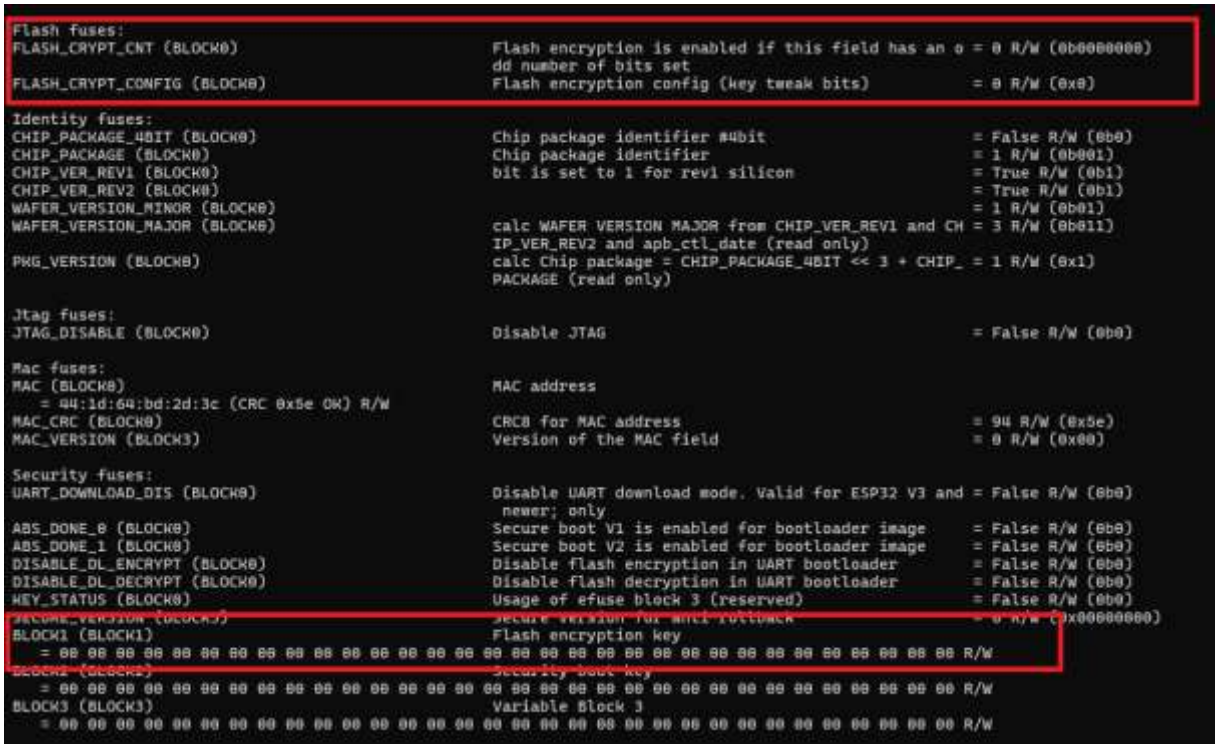
A segunda rota é a responsável por fornecer o firmware para o ESP32. Quando o microcontrolador realiza uma requisição HTTPS para essa rota, o servidor envia o arquivo binário armazenado, permitindo que o dispositivo realize o download do novo firmware e inicie o processo de atualização. Essa comunicação ocorre de forma autenticada e segura, pois o servidor Flask foi configurado para operar sobre o protocolo HTTPS. Para isso, foram gerados, por meio da ferramenta OpenSSL, um certificado digital e uma chave privada nos formatos cert.pem e key.pem, respectivamente. Esses arquivos são utilizados pelo parâmetro ssl_context na inicialização do servidor, garantindo que toda a comunicação seja criptografada e que o ESP32 possa validar a autenticidade do servidor antes de baixar o firmware.

O servidor é executado localmente por meio do terminal, utilizando o comando “python3 upload_server.py”, onde upload_server.py é o nome do servidor desenvolvido, a partir do diretório onde o arquivo do servidor e o firmware estão armazenados. Uma vez em execução, o ESP32 pode se conectar ao servidor pelo endereço configurado para verificar se há uma nova versão disponível.

4.4 Flash Encryption

Antes de realizar a configuração necessária para habilitar a criptografia da flash, foi feita a verificação do estado dos eFuses relacionados a esse recurso. A Figura 14 apresenta o resumo dessas configurações. Nela, destaca-se que o campo FLASH_CRYPT_CNT, responsável por indicar se a criptografia está ativada, ainda não possui nenhum bit gravado. Além disso, o BLOCK1, área reservada para o armazenamento da chave de criptografia, aparece vazio, evidenciando que nenhuma chave havia sido definida. Assim, confirma-se que o dispositivo encontrava-se sem qualquer mecanismo de criptografia de flash habilitado antes das configurações realizadas.

Figura 14 - Estado dos eFuses antes da criptografia da flash



Fonte: Autor, 2025

Inicialmente, foi gerada uma chave AES-256, responsável por realizar a criptografia e descriptografia automática dos dados armazenados na memória Flash. A geração dessa chave foi realizada por meio do comando “espsecure.py generate_flash_encryption_key”. Como argumento adicional, foi especificado o caminho do arquivo da chave. Em seguida, essa chave foi gravada no bloco de eFuses reservado para o mecanismo de criptografia, utilizando o comando “espefuse.py burn_key flash_encryption”. Durante a execução, foram fornecidos como argumentos adicionais a porta de comunicação utilizada e o caminho do arquivo contendo a chave gerada.

Em seguida, foram verificados os endereços de offset das partições presentes na memória Flash, correspondentes às partições do bootloader, da tabela de partições e da aplicação principal. Essa informação é essencial para a etapa posterior, na qual os arquivos criptografados devem ser gravados na ESP32 exatamente nos endereços de memória destinados a cada partição.

Após essa verificação, procedeu-se com a configuração das regiões da memória que seriam submetidas ao processo de criptografia. O eFuse

FLASH_CRYPT_CONFIG foi ajustado com o valor 0x0F, o que determina que todas as regiões citadas anteriormente sejam criptografadas pelo dispositivo. Essa etapa foi executada por meio do comando “espefuse.py burn_efuse FLASH_CRYPT_CONFIG 0x0F”, sendo especificada como argumento adicional a porta de comunicação utilizada.

Com a chave e as regiões de memória configuradas, deu-se continuidade ao processo com a queima do eFuse FLASH_CRYPT_CNT. Ao gravar o valor 0x1, o número de bits ativos nesse registrador torna-se ímpar, o que habilita o mecanismo de criptografia do chip. Essa operação foi realizada utilizando o comando “espefuse.py burn_efuse FLASH_CRYPT_CNT 0x1”, no qual foi especificada como argumento adicional a porta de comunicação usada na gravação.

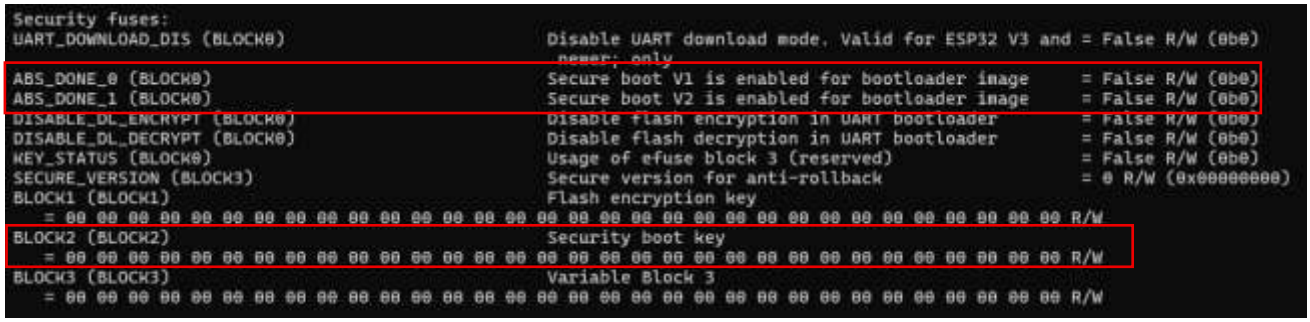
Posteriormente, cada binário referente às partições do firmware foi criptografado individualmente utilizando a ferramenta espsecure.py. As criptografias foram realizadas utilizando os endereços de cada partição na Flash, os caminhos dos arquivos binários das partições e da chave utilizada.

Cada arquivo resultante foi então gravado na memória do ESP32 nos respectivos endereços, completando o processo de criptografia do firmware. Após a primeira inicialização, o controlador passa a realizar automaticamente a descriptografia dos blocos de memória durante a execução.

4.5 Secure Boot

Assim como no processo de *Flash Encryption*, realizou-se inicialmente a verificação do estado dos eFuses do ESP32. O resumo da leitura desses registradores é apresentado na Figura 15. Na figura, percebe-se que o campo ABS_DONE_0, responsável por indicar a ativação da primeira versão do *Secure Boot*, ainda estava desabilitado. Além disso, o BLOCK2, área destinada ao armazenamento da chave pública utilizada na verificação da assinatura do firmware, aparece completamente zerado, revelando que nenhuma chave havia sido gravada. Dessa forma, confirma-se que o dispositivo se encontrava sem o recurso de Secure Boot habilitado antes da configuração realizada.

Figura 15 - Estado dos eFuses antes do *Secure Boot*



Fonte: Autor, 2025

Para ativar o *Secure Boot*, inicialmente foi gerada uma chave de assinatura no formato PEM, utilizando o comando “openssl ecparam -name prime256v1 -genkey -noout -out secure_boot_signing_key.pem”. Vale reforçar que, caso a chave não seja previamente criada, o próprio sistema gera automaticamente uma nova chave no momento da ativação do *Secure Boot*.

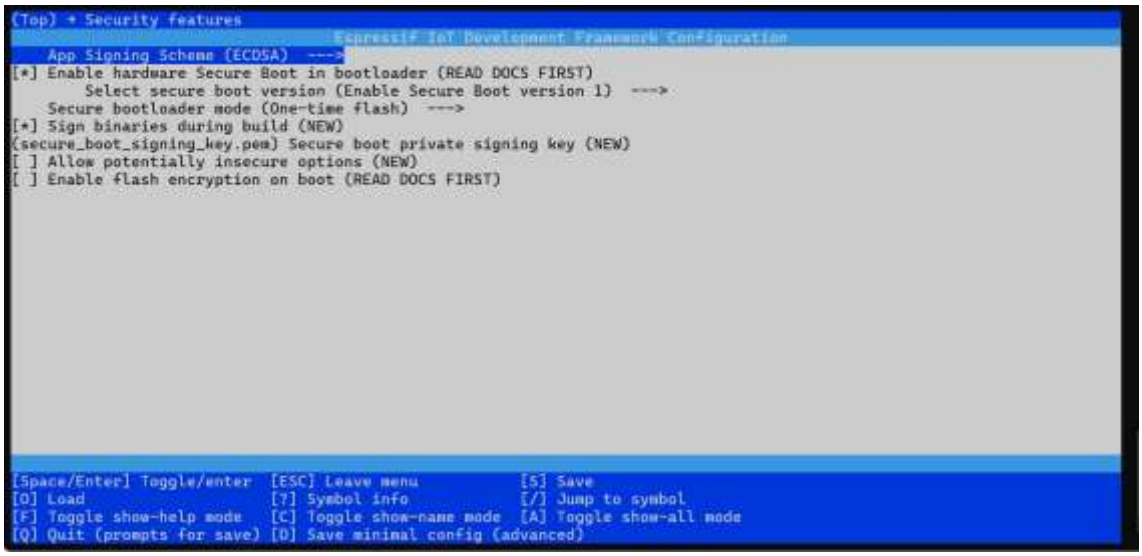
Em seguida, foram realizadas as configurações necessárias para a habilitação do recurso. Por meio do comando “idf.py menuconfig” foi acessado o menu de configuração do ESP32. Dentro da seção “Security Features”, a opção “Enable Secure Boot” foi marcada, configurando o recurso no modo One-Time Flash. Essa configuração garante que o *bootloader* seja permanentemente vinculado à chave utilizada na assinatura, impossibilitando futuras modificações após a queima dos *eFuses*. Ainda nessa etapa, foi especificado o caminho da chave de assinatura previamente gerada, permitindo que o sistema assine automaticamente os binários durante o processo de *build*.

A Figura 16 apresenta a tela de configuração do *menuconfig*, evidenciando as opções habilitadas para o *Secure Boot*. Nesse trecho, é possível identificar tanto a ativação do modo One-Time Flash quanto a definição do caminho da chave de assinatura utilizada no processo, conforme discutido anteriormente.

Posteriormente, foi realizada a construção do *bootloader* por meio do comando “idf.py bootloader”. Durante esse processo, o sistema gerou o arquivo bootloader.bin e exibiu, ao final da compilação, um comando sugerido para a gravação manual do *bootloader* na memória Flash. Esse comando deve ser executado antes do uso do “idf.py flash”, uma vez que o *bootloader* precisa ser gravado separadamente e de forma limpa antes do primeiro boot com o *Secure Boot* habilitado. Essa etapa garante

que o *bootloader* seja devidamente assinado e compatível com as configurações de segurança atuais do dispositivo, evitando falhas na inicialização.

Figura 16 - *Menuconfig Secure Boot*



Fonte: Autor, 2025

Após a gravação do *bootloader* e das demais partições realizadas pelo comando “idf.py flash”, foi necessário reinicializar a placa para que o segundo estágio do *bootloader* fosse executado. Essa reinicialização permite que o ESP32 valide as novas configurações de segurança e inicie o processo de autenticação e verificação de integridade do firmware. Após o carregamento do *bootloader* e das demais partições, o *Secure Boot* foi efetivamente ativado por meio da queima do *eFuse* correspondente.

5 RESULTADOS

Após a implementação completa da arquitetura proposta, foram realizados testes experimentais com o objetivo de validar o funcionamento correto de cada módulo do sistema e verificar a integração entre as camadas de segurança desenvolvidas. Os testes foram realizados utilizando dois microcontroladores ESP32, representando respectivamente o emissor e o receptor.

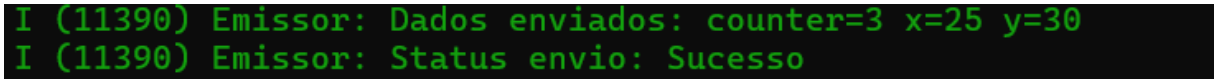
5.1 Comunicação entre dois dispositivos via ESP-NOW

Inicialmente, foi testada a comunicação ponto a ponto entre o emissor e o receptor utilizando o protocolo ESP-NOW com o modo de criptografia habilitado. Durante os testes, foi possível observar que os pacotes de dados eram transmitidos e recebidos corretamente, demonstrando o sucesso no emparelhamento dos dispositivos e na troca de mensagens criptografadas.

Durante os testes de transmissão, o dispositivo emissor enviou continuamente pacotes de dados contendo as variáveis simuladas *x* e *y*, além de um contador que identifica o número sequencial de cada pacote transmitido. A cada envio, o emissor registrou no log o status da transmissão, indicando “Sucesso” para todos os pacotes enviados dentro do intervalo observado. O receptor, ao receber os pacotes, enviou uma resposta ao emissor contendo apenas a indicação do resultado do processo, apresentando no log o tamanho do pacote recebido, o valor do contador correspondente e os dados transmitidos.

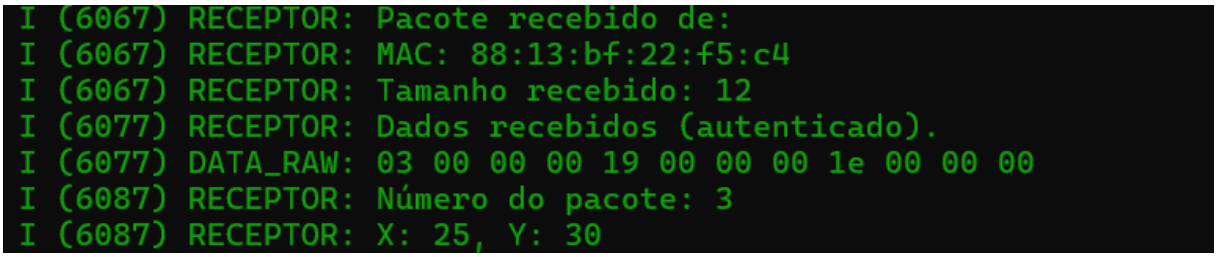
A Figura 17 apresenta o log de saída do emissor, no qual podem ser verificados os dados transmitidos e o respectivo status de envio. Já a Figura 18 mostra o log do receptor, permitindo observar o endereço MAC do dispositivo, o tamanho do pacote recebido e os dados recebidos, exibidos tanto em formato hexadecimal quanto em valores decodificados e legíveis. Dessa forma, confirma-se que o fluxo de transmissão e recepção de dados entre os dispositivos está ocorrendo de maneira correta e consistente.

Figura 17 - Log do emissor para transmissão de mensagem via ESP-NOW



Fonte: Autor, 2025

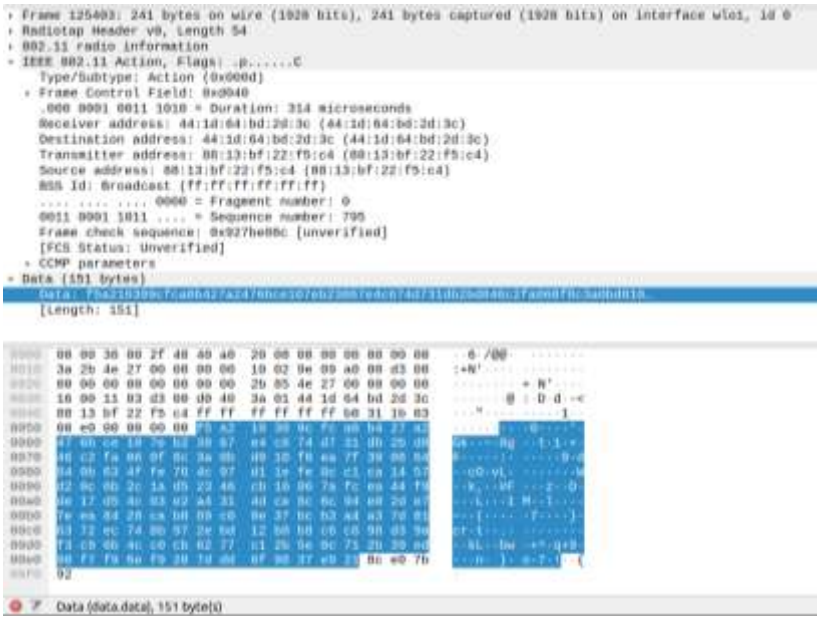
Figura 18 - Log do receptor para recepção de mensagens via ESP-NOW



Fonte: Autor, 2025

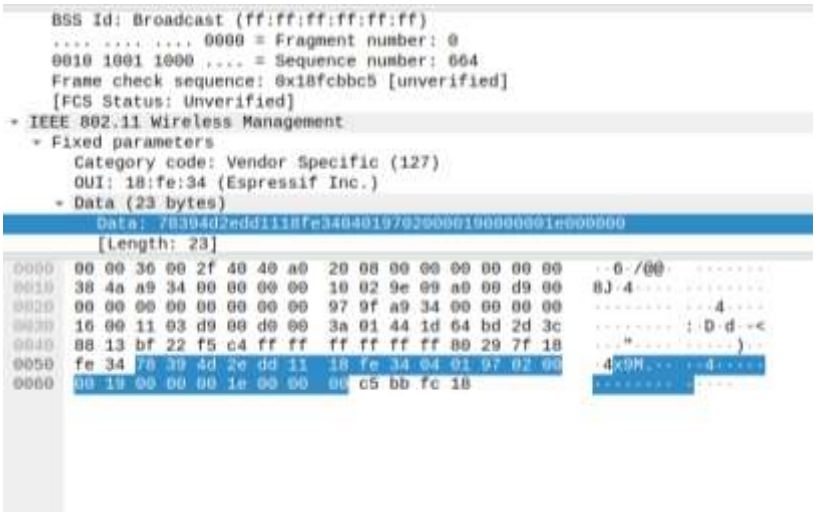
Durante os testes, foram realizadas capturas de pacotes ESP-NOW com a criptografia tanto habilitada quanto desabilitada. As coletas foram feitas no Wireshark operando em modo *monitor*, permitindo captar quadros transmitidos pelas redes sem fio. A Figura 19 apresenta a captura realizada com a criptografia ativada, enquanto a Figura 20 exibe a captura obtida após a desativação da criptografia. Com isso, torna-se possível comparar diretamente o impacto da criptografia sobre a legibilidade dos dados transmitidos.

Figura 19 - Captura de pacotes ESP-NOW via Wireshark com criptografia habilitada



Fonte: Autor, 2025

Figura 20 - Captura de pacotes ESP-NOW via Wireshark com a criptografia desabilitada



Fonte: Autor, 2025

Na primeira captura, correspondente à comunicação com a criptografia ativada, foram aplicados dois filtros no Wireshark para facilitar a identificação dos pacotes ESP-NOW. O primeiro filtro, “wlan.addr”, foi utilizado para exibir apenas os quadros que envolviam o endereço MAC de uma das placas envolvidas na comunicação, enquanto o segundo filtro, “wlan.fc.protected == 1”, foi empregado para exibir exclusivamente os pacotes protegidos por criptografia.

A análise da captura realizada no Wireshark indicou que os quadros transmitidos apresentavam o campo *Protected Frame* habilitado, bem como a presença do campo *CCMP parameters*, utilizado pelo padrão IEEE 802.11 para sinalizar quadros protegidos por mecanismos de criptografia na camada MAC. No contexto do ESP-NOW, a observação desses campos indica que os quadros foram transmitidos de forma protegida, sem que o Wireshark realize a decodificação ou validação criptográfica do conteúdo.

O campo de dados desses pacotes apresentou apenas valores aparentemente aleatórios em formato hexadecimal, impossibilitando a leitura de informações em texto claro. Dessa forma, a captura de tráfego evidencia que o conteúdo transmitido não é legível. Entretanto, a ferramenta não realiza a validação criptográfica dos quadros ESP-NOW, limitando-se à exibição do *payload* em formato bruto.

Na segunda captura, obtida após a desativação da criptografia, também foram aplicados filtros no Wireshark para identificação dos pacotes ESP-NOW. Novamente utilizou-se o filtro “wlan.addr” para exibir apenas os quadros que envolviam o endereço MAC de uma das ESP32, e o filtro “wlan.fc.protected == 0”, destinado a mostrar exclusivamente os pacotes não protegidos. O Wireshark mostrou que o campo “Protected Frame” estava desabilitado, e o cabeçalho CCMP não aparecia mais na estrutura do pacote. Nesse caso, o conteúdo do campo de dados continha bytes que correspondiam a informações legíveis, como identificadores e trechos reconhecidos pelo Wireshark, incluindo a identificação do fabricante *Espressif Inc*.

5.2 Autenticação via ECDSA

Durante os testes, o emissor transmitiu periodicamente uma mensagem de *challenge,* que consistia em uma mensagem de autenticação assinada com o algoritmo ECDSA. A mensagem foi usada para verificar a autenticidade do dispositivo emissor. No receptor, ao receber o *payload* contendo a assinatura e a mensagem, o sistema identificou corretamente o formato e iniciou o processo de verificação. Nos casos observados, a assinatura foi validada com sucesso, conforme indicado pelas mensagens no log do receptor. Após cada verificação bem-sucedida, o receptor

enviou uma resposta de confirmação ao emissor, que foi devidamente reconhecida no terminal do emissor.

A Figura 21 e Figura 22 exemplificam as saídas de log do emissor e do receptor durante o processo de validação da mensagem. No emissor, visto na Figura 21, é possível observar o envio bem-sucedido da mensagem de *challenge* e, em seguida, a recepção de uma mensagem de confirmação (ACK), acompanhada do endereço MAC do receptor, indicando ao emissor o resultado da verificação da assinatura digital, permitindo a continuidade da comunicação apenas após a autenticação bem-sucedida realizada pelo ECDSA.

No receptor, visto na Figura 22, observa-se que um pacote foi recebido, contendo o endereço MAC do emissor, o tamanho do pacote e a identificação do *payload*, correspondente à curva a ser assinada. Posteriormente, é possível verificar que a assinatura foi validada com sucesso, e um ACK de confirmação foi enviado ao emissor, concluindo o processo de autenticação. Assim, evidencia-se que o mecanismo de assinatura funciona corretamente, garantindo a autenticação mútua entre os dispositivos.

Figura 21 - Envio da mensagem a ser verificada

```
I (10990) Emissor: Challenge enviado.
I (10990) Emissor: Status envio: Sucesso
I (11290) Emissor: ACK recebido de 44:1d:64:bd:2d:3c => 1
```

Fonte: Autor, 2025

Figura 22 - Verificação da assinatura enviada pelo emissor

```
I (5667) RECEPTOR: MAC: 88:13:bf:22:f5:c4
I (5667) RECEPTOR: Tamanho recebido: 128
I (5677) RECEPTOR: Payload corresponde à CURVA (assinatura + msg). Tentando verificar...
I (5967) RECEPTOR: Assinatura válida!
I (5967) RECEPTOR: ACK enviado com sucesso (valor=1)
```

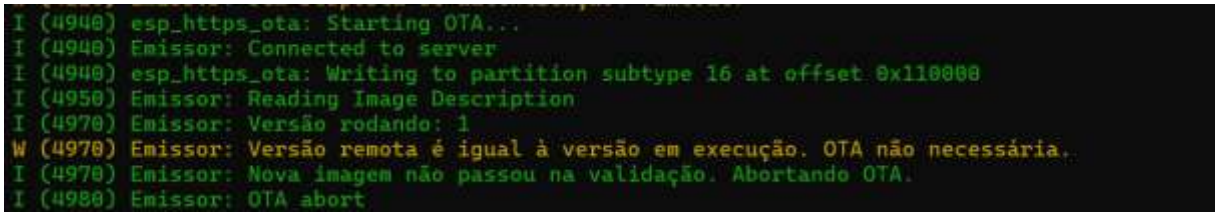
Fonte: Autor, 2025

5.3 Atualização OTA

Após a conclusão da implementação do mecanismo de atualização OTA, foram realizados testes para validar seu funcionamento e a integração com os demais módulos de segurança do sistema.

Durante os testes, o emissor estabeleceu a conexão com o ponto de acesso Wi-Fi configurado e, em seguida, iniciou o processo de verificação da versão disponível no servidor OTA. A Figura 23 apresenta a saída de log gerada durante uma tentativa de conexão com o servidor. Nessa figura, percebe-se que o sistema consulta a versão remota e identifica que o firmware em execução no dispositivo já corresponde à versão mais recente disponível. Como resultado, o procedimento de atualização é interrompido automaticamente. Dessa forma, demonstra-se que o mecanismo de verificação de versão está funcionando corretamente, evitando atualizações desnecessárias.

Figura 23 - Atualização OTA abortada



Fonte: Autor, 2025

Quando uma nova versão era detectada, o dispositivo realizava automaticamente o download do arquivo binário e o gravava na partição OTA designada. O progresso da operação pôde ser acompanhado por meio dos logs no terminal, que exibiam mensagens de status.

Os resultados podem ser observados na Figura 24 e Figura 25. Na Figura 24, nota-se que o firmware verifica a existência de atualizações disponíveis e identifica uma nova versão no servidor. Em seguida, o sistema compara o arquivo atual com o arquivo remoto e detecta que as versões são diferentes. Diante disso, o processo de

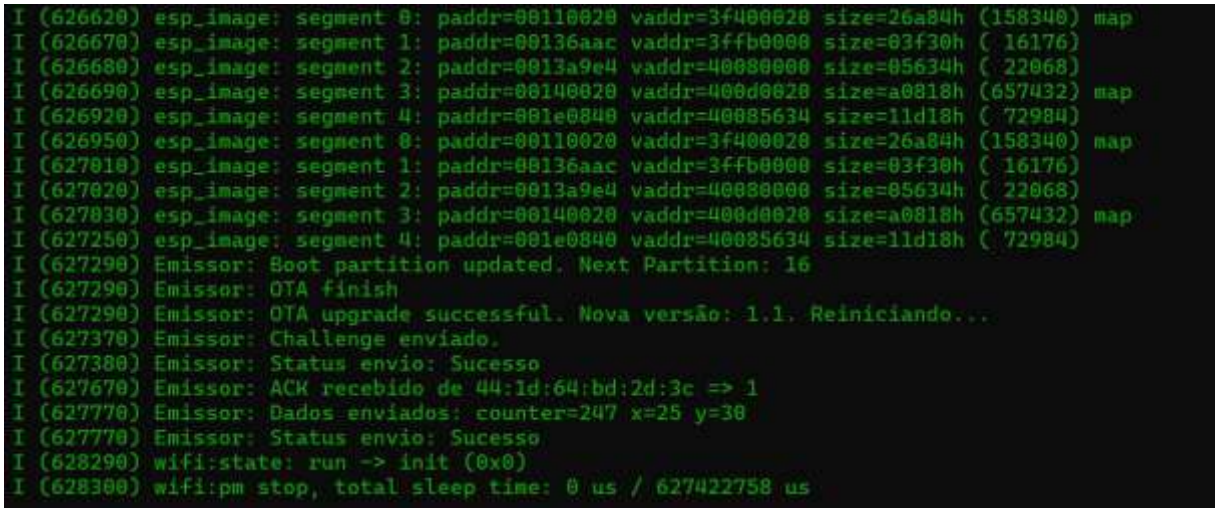
atualização OTA é iniciado, com a inicialização do serviço responsável pela transferência do novo firmware.

A Figura 25 ilustra o processo de atualização em andamento, mostrando a gravação do novo firmware na partição correspondente. Após a conclusão do download, o sistema informa o término da atualização e o dispositivo é reinicializado automaticamente, passando a executar a nova versão do firmware.

Figura 24 - Identificação de nova atualização disponível

Fonte: Autor, 2025

Figura 25 - Processo de atualização OTA



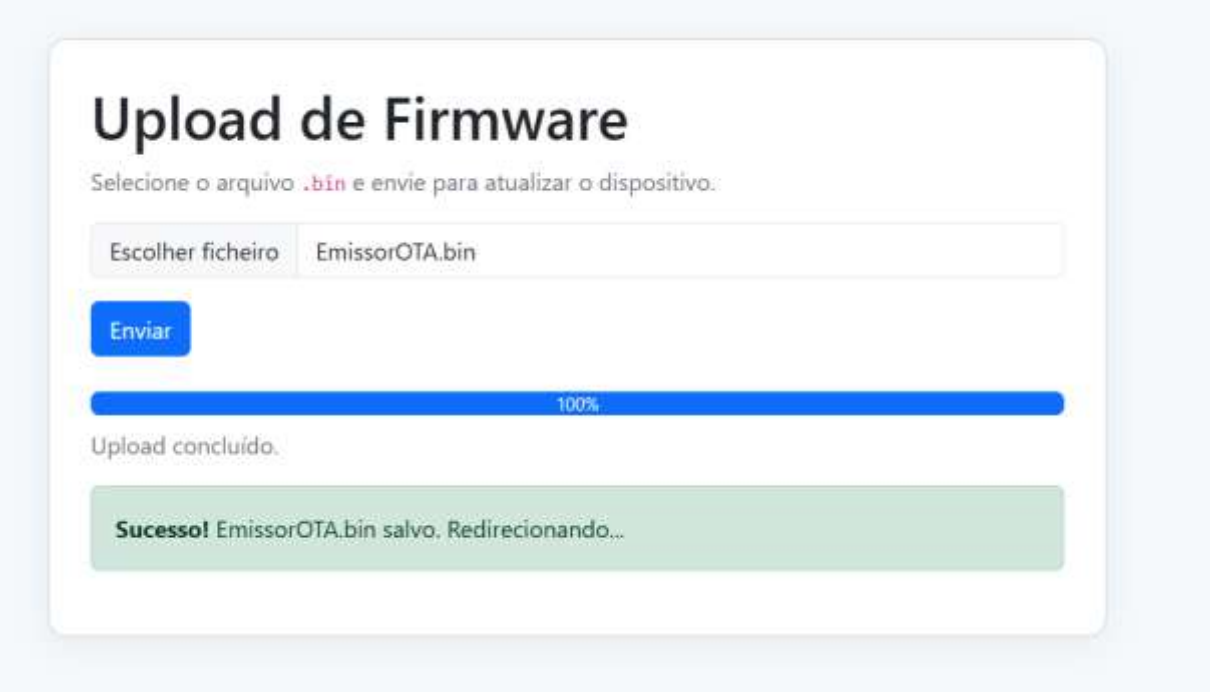
Fonte: Autor, 2025

Após a reinicialização, o emissor iniciou com a nova versão do firmware, evidenciada pela mensagem de identificação exibida no início da execução, o que confirmou que o processo de substituição da partição ativa foi concluído com sucesso.

Durante os testes, observou-se que o tempo médio de atualização variou entre 20 e 30 segundos, dependendo da qualidade do sinal Wi-Fi e do tamanho da imagem binária. Em todas as execuções, a comunicação TLS foi estabelecida corretamente utilizando o certificado digital configurado no firmware, garantindo a autenticação do servidor e evitando a aplicação de firmwares não autorizados.

O servidor OTA, por sua vez, registrou corretamente os uploads e downloads realizados. Esse recurso facilitou a validação do processo e demonstrou o bom funcionamento do sistema de gerenciamento das versões de firmware. A Figura 26 exibe o layout do servidor OTA desenvolvido.

Figura 26 - Layout do servidor desenvolvido



Fonte: Autor, 2025

Além disso, foi possível verificar que o processo de OTA não interferiu no funcionamento do protocolo ESP-NOW, permitindo que o dispositivo continuasse a se comunicar normalmente após a reinicialização.

5.4 Flash Encryption

A ativação da Flash Encryption exigiu alguns procedimentos de verificação para garantir sua correta implementação. A Figura 27 apresenta o resultado da operação de queima da chave de criptografia. Na figura é possível notar que no início da execução, o terminal informa que os dados sensíveis foram ocultados, garantindo que o conteúdo da chave não fosse exibido durante o procedimento. Em seguida, identifica-se o bloco de eFuse reservado para o armazenamento da chave e indica que suas permissões de leitura e escrita seriam desativadas de forma permanente, assegurando que a chave não possa ser lida nem modificada posteriormente. Por fim, o processo é concluído com sucesso, confirmando que a gravação foi realizada corretamente e que a chave criptográfica passou a estar protegida.

Figura 27 - Resultado da queima da chave de criptografia

Fonte: Autor, 2025

A Figura 28 apresenta o resultado do comando responsável por ativar a criptografia da memória Flash. Durante a execução, o utilitário indica que o valor inicial do campo FLASH_CRYPT_CNT era 0b00000000 e foi alterado para 0b00000001, habilitando assim o mecanismo de criptografia do chip.

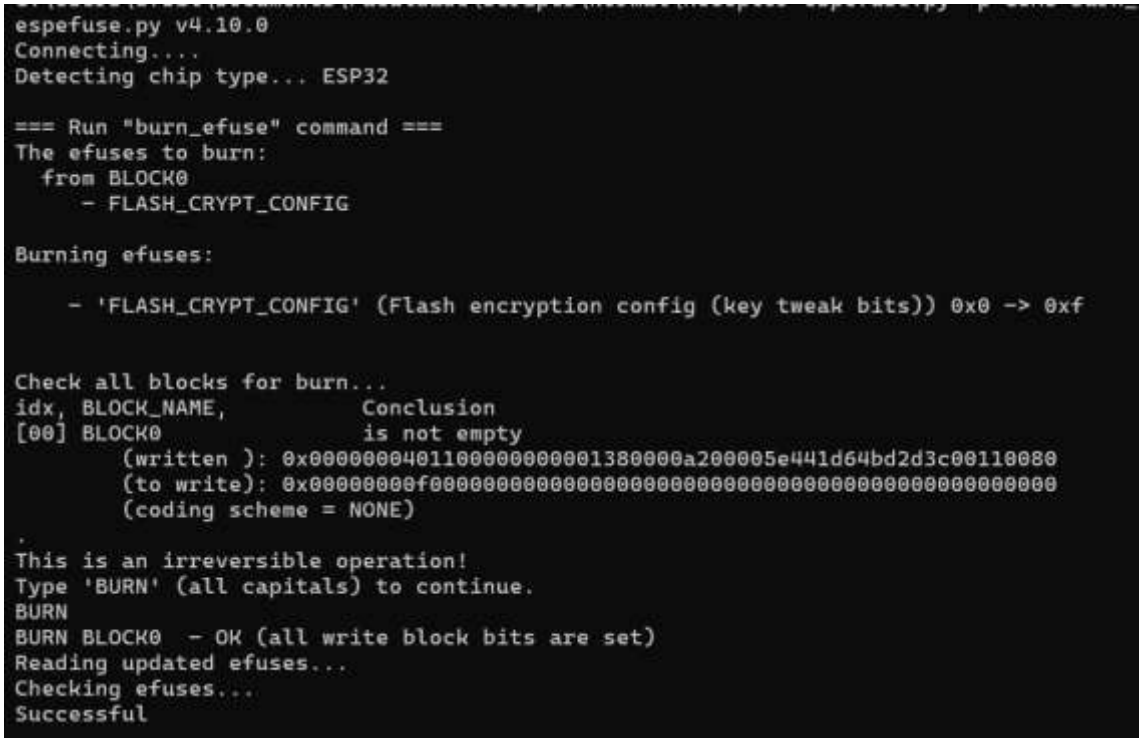
Figura 28 - Queima do eFuse FLASH_CRYPT_CNT

Fonte: Autor, 2025

A Figura 29 apresenta o resultado da configuração do eFuse FLASH_CRYPT_CONFIG, responsável por definir quais regiões da memória Flash serão protegidas pelo mecanismo de criptografia. Durante a execução, o utilitário informa que o valor anterior do campo era 0x0 e foi atualizado para 0x0F, configurando os bits de ajuste de chave utilizados internamente pelo hardware do ESP32 para determinar como os blocos de dados serão criptografados em cada região da memória.

Após a confirmação da operação, o log confirma que a queima foi concluída com êxito, validando a gravação no bloco de eFuses correspondente. Assim como nas demais operações de eFuse, essa modificação é irreversível, garantindo que a configuração de criptografia da memória Flash permaneça permanentemente habilitada.

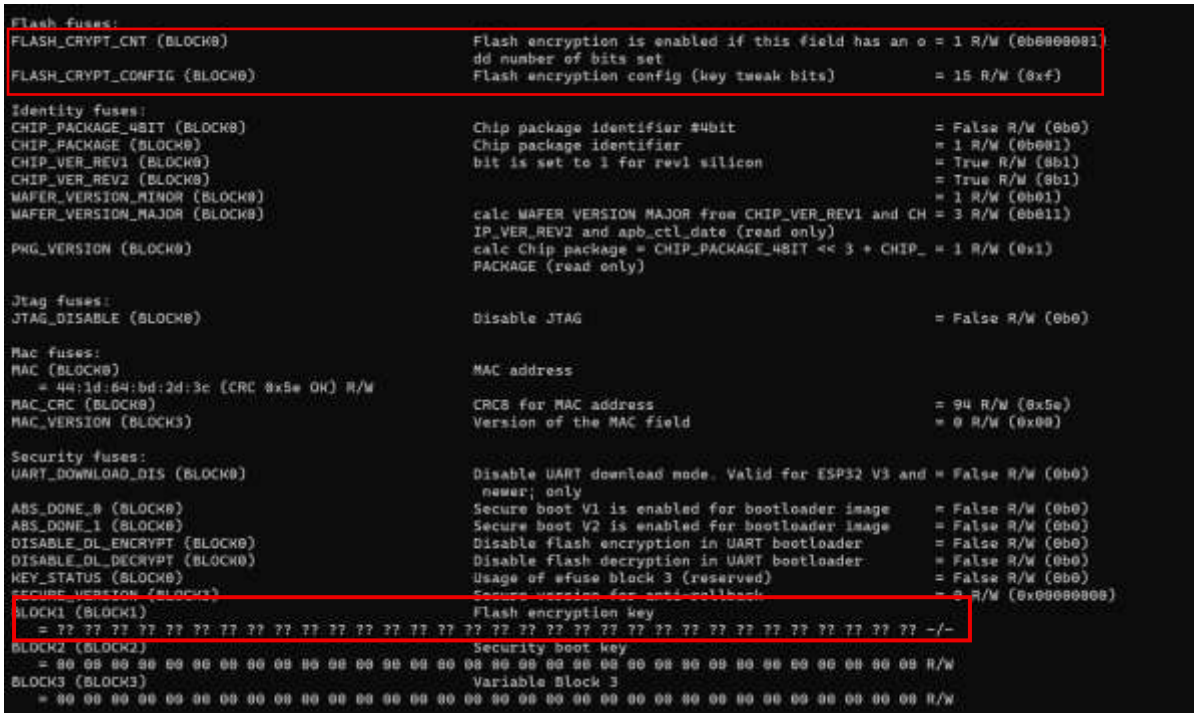
Figura 29 - Configuração eFuse FLASH_CRYPT_CONFIG



Fonte: Autor, 2025

Após a execução dos procedimentos de configuração e ativação da criptografia da memória Flash, foi possível validar as modificações por meio do comando “espefuse.py summary”. A Figura 30 apresenta o resumo do estado final dos eFuses do dispositivo após a conclusão do processo. Nela, podem ser verificados os campos modificados, incluindo os valores atualizados dos contadores e os blocos de chave devidamente gravados e protegidos

Figura 30 - Estado dos eFuses depois da criptografia da flash



Fonte: Autor, 2025

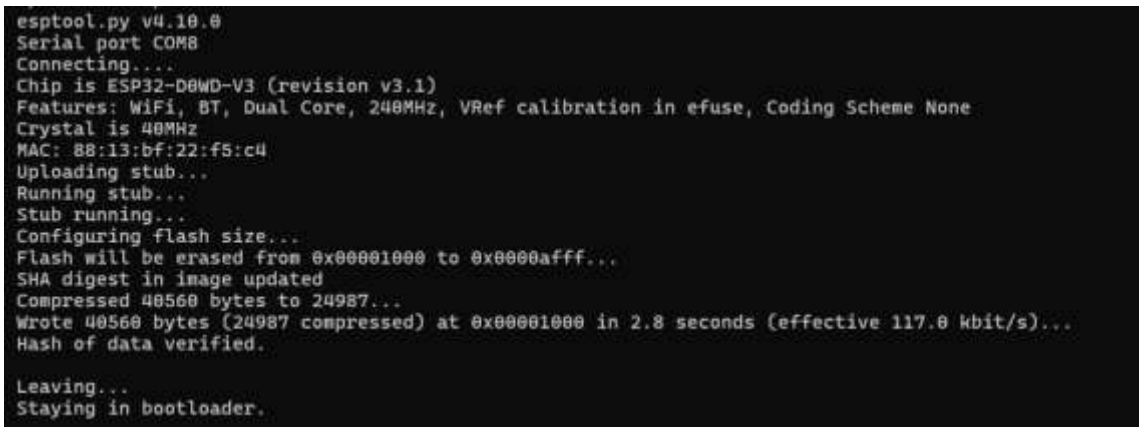
Observa-se que o campo FLASH_CRYPT_CNT passou a conter um bit ativo, o que confirma a habilitação do mecanismo de criptografia. Além disso, o BLOCK1, anteriormente vazio, encontra-se agora preenchido, indicando que a chave AES-256 foi devidamente gravada na área de eFuses reservada para esse propósito. Dessa forma, confirma-se que o dispositivo concluiu com sucesso o processo de habilitação da criptografia da Flash.

5.5 Secure Boot

A Figura 31 mostra o log do processo de gravação do bootloader na memória Flash do ESP32. No início, o esptool.py identifica corretamente o chip e exibe suas principais características, como frequência de 240 MHz, suporte a Wi-Fi e Bluetooth, e o esquema de codificação de eFuse configurado como *None*. Em seguida, ocorre o upload do stub, que faz a comunicação entre o computador e o microcontrolador durante a gravação. O log indica o sucesso dessa etapa, seguida do apagamento do setor localizado no endereço 0x1000, reservado para o bootloader.

Posteriormente, o sistema comprime e grava os dados, confirmando por meio do log que o arquivo bootloader.bin foi gravado com sucesso. Por fim, é feita a confirmação da integridade dos dados por meio de verificação de Hash.

Figura 31 - Resultado gravação do *bootloader*

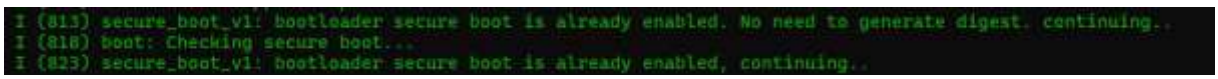


Fonte: Autor, 2025

Com a configuração concluída e o *bootloader* devidamente gravado, a placa foi reinicializada para dar início ao processo de verificação de inicialização segura. Durante o *boot*, foi possível observar no monitor serial que o sistema identificou corretamente a ativação do *Secure Boot*, exibindo mensagens que confirmam a validação da assinatura digital do *bootloader* e do aplicativo principal.

A Figura 32 apresenta o log de inicialização indicando a mensagem “Secure Boot is already enabled”, demonstrando que a assinatura digital do *bootloader* foi reconhecida e validada com sucesso.

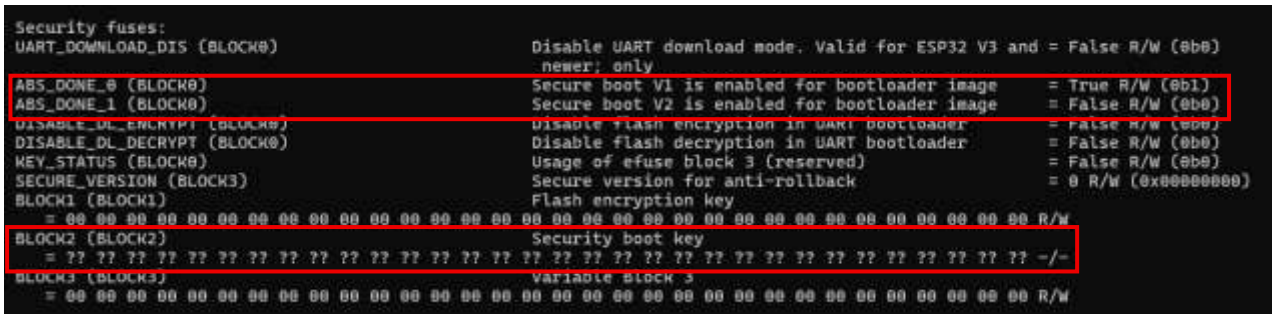
Figura 32 - Log indicando ativação do Secure Boot



Fonte: Autor, 2025

A Figura 33 exibe a leitura posterior dos eFuses, mostrando a queima do bit ABS_DONE_0, que indica a ativação permanente da primeira versão do *Secure Boot*. Além disso, o BLOCK2 apresentou-se devidamente preenchido, comprovando que o sistema havia gravado a chave utilizada na verificação das assinaturas digitais. Dessa forma, confirma-se que o dispositivo teve o Secure Boot habilitado com sucesso e que a chave de verificação foi armazenada corretamente nos eFuses.

Figura 33 - Estado dos eFuse depois do Secure Boot



Fonte: Autor, 2025

6 CONCLUSÃO E PROPOSTA DE TRABALHOS FUTUROS

A implementação e a validação experimental da arquitetura proposta permitiram avaliar a viabilidade do uso do SoC ESP32 como plataforma para a integração de mecanismos de segurança voltados para aplicações de Internet das Coisas. Os resultados obtidos indicaram que a proteção da comunicação via ESP-NOW por meio de mecanismos de criptografia simétrica baseados no algoritmo AES, a autenticação das mensagens por assinaturas digitais ECDSA, a atualização segura de firmware via OTA utilizando TLS, bem como a criptografia da memória Flash e o uso do *Secure Boot* apresentaram comportamento consistente durante os testes realizados, atendendo aos requisitos funcionais definidos para o sistema.

Os testes realizados mostraram que a comunicação entre os dispositivos por meio do protocolo ESP-NOW ocorreu de forma contínua, com as mensagens transmitidas apresentando conteúdo não legível quando analisadas externamente, o que é compatível com o uso de mecanismos de criptografia simétrica para proteção dos dados. A etapa de autenticação baseada no algoritmo ECDSA permitiu que o receptor verificasse a validade das assinaturas digitais associadas às mensagens recebidas, possibilitando a aceitação apenas de mensagens provenientes de dispositivos previamente autenticados no contexto da aplicação.

No que se refere ao mecanismo de atualização de firmware via OTA, os experimentos demonstraram que o dispositivo realizou o processo de verificação, download e gravação de novas imagens apenas quando um firmware válido estava disponível no servidor configurado. Durante esse processo, a comunicação estabelecida por meio do TLS ocorreu sem falhas, sendo observada a rejeição de conexões que não atendiam aos critérios de validação configurados, comportamento compatível com o uso de um canal seguro para a atualização do sistema.

A ativação do recurso de criptografia da memória Flash permitiu verificar que o conteúdo armazenado na memória não volátil não pôde ser interpretado quando acessado diretamente, indicando que o firmware e os dados sensíveis permaneceram cifrados. De forma complementar, os testes com o *Secure Boot* evidenciaram que apenas imagens previamente assinadas foram aceitas durante o processo de inicialização, impedindo a execução de firmwares não autorizados no dispositivo.

Durante o desenvolvimento e os testes, foram identificadas limitações práticas relacionadas ao tempo adicional de processamento introduzido pelas operações de assinatura e verificação ECDSA, bem como pela criptografia de firmware. Além disso, constatou-se a necessidade de atenção especial na configuração dos eFuses, uma vez que essas modificações são permanentes e impactam diretamente o ciclo de vida do dispositivo. Esses fatores não inviabilizaram a solução proposta, mas evidenciam a importância de um planejamento criterioso no projeto de sistemas embarcados seguros.

Por fim, embora os resultados experimentais indiquem o funcionamento adequado da arquitetura no contexto avaliado, não foram realizadas medições quantitativas de latência adicional, consumo energético ou impacto no desempenho decorrente da ativação dos mecanismos criptográficos.

Com base nos resultados alcançados, identificam-se pontos de aprimoramento e continuidade, sendo propostos os seguintes trabalhos futuros:

- Utilizar uma terceira ESP32 como unidade central responsável pela geração e distribuição segura das chaves criptográficas entre os dispositivos, atuando como um gerenciador de chaves.
- Implementar mecanismos de anti-rollback com o objetivo de impedir o upload de firmwares antigos, garantindo que apenas versões legítimas e mais recentes do software possam ser executadas no dispositivo.
- Implementar sistemas de atualização periódica das chaves LMK e PMK, de forma a reduzir o tempo de exposição de cada chave e aumentar a robustez do canal de comunicação.
- Implementar um sistema de monitoramento contínuo da rede para identificar padrões anômalos de tráfego, possíveis tentativas de acesso não autorizado e comportamentos suspeitos.
- Projetar uma rede mesh segura em que múltiplos dispositivos troquem dados criptografados, com estabelecimento e renovação dinâmica de chaves e gerenciamento de rotas autenticadas.

7 REFERÊNCIAS

ABNT - ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. Medição de resistência de aterramento e de potenciais na superfície do solo em sistemas de aterramento. **ABNT NBR 15749**, p. 49, set. 2009.

ANDRÁS, GEDEON. Secure boot and firmware update on a microcontroller-based embedded board, 10 Dezembro 2020.

ANSHUMAN, KALLA; PAWANI, PROMBAGE; MADHUSANKA, LIYANAGE. Introduction to IoT. In: MADHUSANKA, LIYANAGE, et al. **IoT security advantages in authentication**. [S.l.]: Wiley, 2019. Cap. 1, p. 03-26.

ANTONAKAKIS, Manos et al. Understanding the Mirai Botnet. **USENIX Security Symposium**, Vancouver, 2017. 1093-1110.

ATZORI, Luigi; IERA, Antonio; MORABITO, Giacomo. The Internet of Things: A survey. In: _____ **Computer Networks**. 15. ed. [S.l.]: [s.n.], v. 54, 2010. p. 2787-2805.

BABIUCH, Marek; FOLTýNEK, Petr; SMUTNý, Pavel. Using the ESP32 Microcontroller for Data Processing, 2019.

BARROS, Filipe. Estudo e Implementação do Protocolo ECDSA, 2015.

BRASIL. Carta Brasileira para Cidades Inteligentes, 2021. Disponível em: <https://www.gov.br/cidades/pt-br/acesso-a-informacao/acoes-e-programas/desenvolvimento-urbano-e-metropolitano/projeto-andus/carta-brasileira-para-cidades-inteligentes>. Acesso em: 10 Novembro 2025.

BUSINESS INSIGHTS FORTUNE. **Internet of Things (IoT) Market Size, Share & Industry Analysis**, 2024. Disponível em: <https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307>. Acesso em: 10 Novembro 2025.

DATTA, Soumya K. DRAFT- A Cybersecurity Framework for IoT Platforms, 2020.

DELGADO, Ismael et al. Exploring IoT Vulnerabilities in a Comprehensive Remote Cybersecurity Laboratory, 2023.

EMBARCADOS. EMBARCADOS. **ESP32 – Segurança e proteção da flash**, 2020. Disponível em: <https://embarcados.com.br/protecao-da-flash-no-esp32/>. Acesso em: 21 Outubro 2025.

ESPRESSIF. ESP32 Series. [S.l.]: [s.n.], 2019.

ESPRESSIF. Flash Encryption, 2024.

ESPRESSIF. Secure Boot, 2024.

ESPRESSIF. ESP-NOW, 2025.

ESPRESSIF. Over The Air Updates (OTA), 2025.

GARCIA, Laura et al. IoT-Based Smart Irrigation Systems: An Overview on the Recent Trends on Sensors and IoT Systems for Irrigation in Precision Agriculture. 4. ed. [S.l.]: [s.n.], v. 20, 2020. p. 1045-1058.

GRANDVIEW RESEARCH. **Brazil Smart Cities Market & Outlook 2024-2030**, 2024. Disponível em: <https://www.grandviewresearch.com/horizon/outlook/smart-cities-market/brazil>. Acesso em: 10 Novembro 2025.

IEEE. **IEEE Standard for an Architectural Framework for the Internet of Things (IOT)**.

IMARC GROUP. **Brazil Internet of Things (IoT) Market Overview**, 2024. Disponível em: <https://www.imarcgroup.com/brazil-internet-of-things-(iot)-market#:~:text=The%20Brazil%20Internet%20of%20Things,17.80%25%20during%202025%2D2033.>. Acesso em: 10 Novembro 2025.

IOT ANALYTICS. **Most recent analyses and market assessments**, 2024. Disponível em: <https://iot-analytics.com/>. Acesso em: 10 Novembro 2025.

JOHNSON, Don; MENEZES, Alfred; VANSTONE, Scott. The Elliptic Curve Digital Signature Algorithm (ECDSA), 2013.

KEN RESEARCH. **KSA Internet of Things in Healthcare Market**, 2025. Disponível em: <https://www.kenresearch.com/ksa-internet-of-things-in-healthcare-market>. Acesso em: 10 Novembro 2025.

KRAWCZYK, Hugo; PATERSON, Kenneth G.; WEE, Hoeteck. Onthe Security of the TLS Protocol: ASystematic Analysis, 2013.

MADAKAM SOMAYYA, R. R. T. S. Internet of Things (IoT): A Literature Review, Janeiro 2015.

MAIER, Alexander; SHARP, Andrew; VAGAPOV, Yuriy. Comparative Analysis and Practical Implementation of the ESP32 Microcontroller Module for the Internet of Things, 2017.

MEDIUM. **Revolutionizing Updates — The Power of OTA Technology in Modern Systems**, 2025. Disponível em: <https://medium.com/tech-x-humanity/revolutionizing-updates-the-power-of-ota-technology-in-modern-systems-03ed511ede9a>. Acesso em: 10 Novembro 2025.

MICROGENIOS, 2020. Disponível em: <https://www.youtube.com/watch?v=RGbF98XgIzs>. Acesso em: Setembro 30 2025.

MOREIRA, Márcio. ECDSA (Elliptic Curve Digital Signature Algorithm), Julho 2006.

MVNO INDEX. **Brazilian Market Update for IoT Cellular Connectivity**, 2024. Disponível em: <https://mvno-index.com/brazilian-market-update-for-iot-cellular-connectivity/>. Acesso em: 10 Novembro 2025.

PASIC, Roberto; KUZMANOV, Ivo; ATANASOVSKI, Kokan. ESP-NOW communication protocol with ESP32, 2021.

PODDER, Rakesh; BARAI, Ranjit K. Hybrid Encryption Algorithm for the Data Security of ESP32 based IoT-enable Robots, 2021.

RAFIULLAH, KHAN et al. Future Internet: The Internet of Things Architecture, Possible Applications and Key Challenges, 2012.

RFID JOURNAL. **Understanding Global IoT Growth**, 2024. Disponível em: <https://www.rfidjournal.com/news/understanding-global-iot-growth/222206/>. Acesso em: 10 Novembro 2025.

ROSA, Alan F.; TEIXEIRA, David V.; JÚNIOR, Nilton A. Comunicações seguras entre dispositivos IoT utilizando o ESP32, 20 Maio 2020.

SABBATINI, Michel. Hardening IoT Devices: An Analysis of the ESP32 Microcontroller, 1 Setembro 2024.

SICARI, Sabrina et al. Security, privacy and trust in Internet of Things: The road ahead, Janeiro 2015.

TIMKO, Alexander M. Cybersecurity of Internet of Things Devices: A Secure Shell Implementation , 5 Maio 2020.

WU, MIAO et al. Research on the architecture of Internet of things, 2010.