

VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY  
UNIVERSITY OF TECHNOLOGY  
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



## MẬT MÃ VÀ AN NINH MẠNG (CO3069)

---

### BÀI TẬP LỚN

### *BẢO MẬT TRONG INTERNET OF THINGS*

---

**GVHD:** Nguyễn Cao Đạt (*CSE-HCMUT*)

**Sinh viên thực hiện:** Nguyễn Văn Huynh - 2211315 (*Group L01*)  
Nguyễn Duy Khiêm - 2211571 (*Group L01*)  
Nguyễn Đăng Tuấn Tài - 2014412 (*Group L01*)

THÀNH PHỐ HỒ CHÍ MINH 10/2025



## Contents

<b>List of Figures</b>	<b>3</b>
<b>List of Tables</b>	<b>3</b>
<b>Danh sách các thành viên và phân công nhiệm vụ</b>	<b>3</b>
<b>1 Giới thiệu</b>	<b>4</b>
<b>2 Cơ sở lý thuyết</b>	<b>5</b>
2.1 Về Internet of Things (IoT) và bảo mật trong IoT . . . . .	5
2.1.1 Tầm hiểu Internet of Things (IoT) . . . . .	5
2.1.2 Bảo mật Internet vạn vật (IoT Security) là gì? . . . . .	5
2.1.3 Tầm quan trọng của bảo mật IoT . . . . .	6
2.1.4 Khác biệt của bảo mật IoT so với bảo mật thông thường . . . . .	6
2.1.5 Giới thiệu về các tiêu chuẩn về an ninh hệ thống và an toàn thông tin trong IoT . . . . .	7
2.1.5.a Các tiêu chuẩn về an ninh hệ thống và an toàn thông tin trong IoT trên thế giới . . . . .	7
2.1.5.b Các tiêu chuẩn về an ninh hệ thống và an toàn thông tin trong IoT ở Việt Nam . . . . .	8
2.1.6 Nghiên cứu các nguy cơ, mối đe dọa đối với các thành phần trong hệ thống IoT . . . . .	9
2.1.6.a Tầng thiết bị đầu cuối . . . . .	9
2.1.6.b Tầng mạng . . . . .	9
2.1.6.c Tầng nền tảng . . . . .	9
2.1.6.d Tầng ứng dụng . . . . .	10
2.1.6.e Tầng Quản lý và Chính sách . . . . .	10
2.1.7 Nghiên cứu các giải pháp an ninh hệ thống và an toàn thông tin cho IoT . . . . .	10
2.1.7.a Tầng thiết bị . . . . .	10
2.1.7.b Tầng mạng . . . . .	10
2.1.7.c Tầng nền tảng và ứng dụng . . . . .	11
2.1.7.d Tầng quản lý và chính sách . . . . .	11
2.2 Về lý thuyết và công cụ phục vụ triển khai . . . . .	12
2.2.1 Cơ sở lý thuyết . . . . .	12
2.2.1.a Secure Boot . . . . .	12
2.2.1.b Flash Encryption . . . . .	12
2.2.1.c ECDSA (Elliptic Curve Digital Signature Algorithm) thuật toán sinh chữ ký số dựa trên đường cong Elliptic. . . . .	12
2.2.2 Công cụ phục vụ triển khai . . . . .	15
2.2.2.a Phần mềm . . . . .	15
2.2.2.b Phần cứng . . . . .	16
<b>3 Phân tích và thiết kế</b>	<b>17</b>
3.1 Mục tiêu thiết kế bảo mật . . . . .	17
3.2 Phân tích SoC ESP32 và các module phần cứng liên quan . . . . .	17
3.3 Kiến trúc bảo mật tổng thể . . . . .	18
<b>4 Hiện thực và đánh giá</b>	<b>21</b>
4.1 Hiện thực secure boot version 1 . . . . .	21
4.2 Hiện thực mã hóa bộ nhớ flash . . . . .	33
<b>5 Kết luận</b>	<b>40</b>
5.1 Kết luận chung . . . . .	40
5.2 Hướng phát triển . . . . .	41
<b>6 Tài liệu tham khảo</b>	<b>42</b>
<b>7 Phụ lục</b>	<b>43</b>



## List of Figures

1	Ví dụ về kiến trúc hệ thống IOT . . . . .	9
2	Dường cong Elliptic. <a href="#">Nguồn</a> . . . . .	13
3	Phép cộng trên Elliptic. <a href="#">Nguồn</a> . . . . .	14
4	Chuỗi tin cậy khởi động . . . . .	17
5	Sơ đồ kiến trúc bảo mật tổng thể . . . . .	19
6	Kết quả Build Bootloader (1) . . . . .	23
7	Kết quả Build Bootloader (2) . . . . .	23
8	Build Firmware . . . . .	24
9	Bảng tóm tắt cấu hình secure boot v1 . . . . .	25
10	Quan sát dữ liệu trong bộ nhớ flash tại địa chỉ 0x0 . . . . .	26
11	Kết quả chạy firmware đã kí . . . . .	27
12	Kết quả chạy firmware không được kí . . . . .	29
13	Kết quả chạy bootloader "lạ" . . . . .	30
14	Dánh giá hiện trạng bảo mật khi chưa mã hóa . . . . .	32
15	Tự động mã hóa Bootloader, Partition Table . . . . .	34
16	Tự động mã hóa firmware . . . . .	34
17	Tự động cập nhật eFuse - Block0 . . . . .	35
18	Tự động reset sau khi build lần đầu tiên . . . . .	35
19	Khởi động lại kết hợp với secure boot . . . . .	35
20	Chương trình được mã hóa tự động sau lần boot đầu tiên đã thực thi . . . . .	35
21	Đảo key để phục vụ mã hóa thủ công . . . . .	36
22	So sánh kết quả mã hóa thủ công và chưa mã hóa . . . . .	37
23	Kết quả sau khi nạp firmware đã mã hóa thủ công . . . . .	38
24	So sánh khả năng bảo mật dữ liệu khi mã hóa và khi chưa mã hóa . . . . .	39

## List of Tables

1	So sánh bảo mật hệ thống truyền thông và bảo mật IoT . . . . .	6
2	So sánh giữa ECDSA và RSA . . . . .	15
3	Các công cụ và chức năng chính sử dụng trong bảo mật ESP32 . . . . .	16
5	Cấu hình secure boot version 1 trong sdkconfig . . . . .	21
6	Phân tích chi tiết Log báo lỗi Secure Boot Check Fail . . . . .	31



## Danh sách các thành viên và phân công nhiệm vụ

No.	Họ và tên	MSSV	Nhiệm vụ	Mức độ hoàn thành
1	Nguyễn Văn Huynh	2211315	<ul style="list-style-type: none"><li>- Tìm hiểu công cụ phục vụ triển khai và cơ sở lý thuyết secure boot &amp; mã hóa flash</li><li>- Phân tích và thiết kế kiến trúc bảo mật tổng thể</li><li>- Hiện thực và đánh giá giải pháp secure boot &amp; mã hóa flash</li></ul>	100%
2	Nguyễn Duy Khiêm	2211571	<ul style="list-style-type: none"><li>- Tìm hiểu công cụ phục vụ triển khai và cơ sở lý thuyết secure boot &amp; mã hóa flash</li><li>- Phân tích và thiết kế kiến trúc bảo mật tổng thể</li><li>- Hiện thực và đánh giá giải pháp secure boot &amp; mã hóa flash</li></ul>	100%
3	Nguyễn Đăng Tuấn Tài	2014412	<ul style="list-style-type: none"><li>- Tìm hiểu bảo mật trong IoT, các tiêu chuẩn về an ninh và an toàn thông tin trong IoT</li><li>- Nghiên cứu các nguy cơ đe dọa các thành phần trong IoT và giải pháp an ninh tương ứng</li><li>- Dánh giá ưu, nhược điểm giải pháp được hiện thực và hướng phát triển trong tương lai</li></ul>	100%



## 1 Giới thiệu

Trong bối cảnh cuộc cách mạng công nghiệp 4.0 diễn ra mạnh mẽ, Internet vạn vật (IoT – Internet of Things) đang nổi lên như một trong những nền tảng công nghệ trọng tâm, góp phần thúc đẩy quá trình chuyển đổi số trên toàn cầu. Với khả năng kết nối hàng tỷ thiết bị vào cùng một mạng lưới thông minh, IoT đang được ứng dụng sâu rộng trong nhiều lĩnh vực như nhà thông minh, y tế số, giao thông thông minh, sản xuất công nghiệp, nông nghiệp chính xác và nhiều lĩnh vực khác. Việc khai thác dữ liệu theo thời gian thực từ các thiết bị này giúp con người tối ưu hóa quy trình, giảm chi phí vận hành và nâng cao chất lượng cuộc sống.

Tuy nhiên, song hành với sự phát triển mạnh mẽ của IoT là những thách thức nghiêm trọng về bảo mật và quyền riêng tư. Số lượng thiết bị IoT tăng nhanh đồng nghĩa với việc bề mặt tấn công cũng mở rộng đáng kể. Nhiều thiết bị thu thập dữ liệu nhạy cảm như vị trí, hình ảnh, thông tin sức khỏe hoặc hành vi người dùng, nhưng lại thiếu các cơ chế bảo vệ an toàn. Thực tế đã ghi nhận nhiều vụ tấn công vào thiết bị IoT, từ việc chiếm quyền điều khiển camera, router, thiết bị gia đình cho đến sử dụng chúng làm công cụ tấn công DDoS quy mô lớn.

Nguyên nhân chủ yếu xuất phát từ việc thiếu tiêu chuẩn bảo mật chuyên biệt cho hệ sinh thái IoT, cũng như hạn chế trong khâu thiết kế, phát triển và quản lý vòng đời thiết bị. Nhiều nhà sản xuất tập trung vào chi phí và tốc độ ra thị trường hơn là tính an toàn. Hệ quả là nhiều thiết bị không được cập nhật phần mềm, vắng lối hoặc xác thực người dùng đúng cách, khiến chúng trở thành điểm yếu trong toàn bộ hệ thống mạng.

Trước tình hình đó, việc nghiên cứu, tìm hiểu và triển khai các biện pháp bảo mật chuyên sâu cho IoT là yêu cầu cấp thiết. Không chỉ nhằm ngăn chặn các rủi ro an ninh mạng, mà còn giúp đảm bảo tính toàn vẹn, bí mật và sẵn sàng của dữ liệu, cũng như xây dựng lòng tin của người dùng và doanh nghiệp khi ứng dụng IoT trong thực tế.



## 2 Cơ sở lý thuyết

### 2.1 Về Internet of Things (IoT) và bảo mật trong IoT

#### 2.1.1 Tìm hiểu Internet of Things (IoT)

Internet of Things (IoT) hay Internet vạn vật là khái niệm chỉ mạng lưới các thiết bị vật lý (cảm biến, bộ điều khiển, máy móc, thiết bị đeo, camera, xe cộ...) có khả năng kết nối Internet để thu thập, trao đổi và xử lý dữ liệu.

#### Các thành phần cơ bản của IoT

Môi trường IoT cũng có kiến trúc phân tầng tương tự mô hình tham khảo OSI hay TCP/IP. Một hệ thống IoT hoàn chỉnh có thể được mô tả theo kiến trúc 5 tầng, trong đó mỗi tầng đảm nhận một vai trò riêng biệt trong quá trình thu thập, truyền tải, xử lý và ứng dụng dữ liệu:

- **Tầng thiết bị (Device Layer):** Bao gồm các cảm biến, bộ truyền động (actuator), thiết bị đeo, máy móc công nghiệp, vi điều khiển... Chức năng chính: thu thập dữ liệu từ môi trường thực tế, định danh thiết bị và cung cấp dữ liệu đầu vào cho hệ thống.
- **Tầng biên/công (Edge Layer):** Bao gồm các gateway, edge server hoặc phần cứng bảo mật chuyên dụng. Dảm nhiệm các tác vụ như: giao thức an toàn, tính toán biên, xử lý sơ bộ dữ liệu trước khi gửi đi, mã hóa, lọc dữ liệu.
- **Tầng mạng (Network Layer):** Là cầu nối giúp truyền dữ liệu giữa thiết bị/edge đến nền tảng xử lý. Sử dụng các công nghệ mạng như VLANs, VPN, 5G, LPWAN (LoRaWAN, NB-IoT...), Wi-Fi, Ethernet,... Dảm bảo dữ liệu được truyền nhanh chóng, toàn vẹn và có tính bảo mật.
- **Tầng đám mây/nền tảng (Cloud/Platform Layer):** Chịu trách nhiệm quản lý thiết bị, thu thập, xử lý và lưu trữ dữ liệu. Các nền tảng phổ biến: AWS IoT, Azure IoT Hub, Google Cloud IoT Core, EMQX, ThingsBoard,... Có thể áp dụng cơ chế xử lý luồng dữ liệu (stream processing), phân tích dữ liệu lớn và trí tuệ nhân tạo (AI/ML).
- **Tầng ứng dụng (Application Layer):** Là nơi dữ liệu được hiển thị và khai thác thông qua dashboard, APIs, dịch vụ ứng dụng. Cung cấp chức năng giám sát, điều khiển thiết bị, phân tích dữ liệu và hỗ trợ ra quyết định.

Ví dụ thực tế: nhà thông minh (smart home), xe tự lái, hệ thống SCADA công nghiệp, thiết bị y tế thông minh, nông trại thông minh...

#### 2.1.2 Bảo mật Internet vạn vật (IoT Security) là gì?

Bảo mật IoT là tập hợp các biện pháp, công nghệ và quy trình được áp dụng để:

- Bảo vệ thiết bị IoT khỏi việc bị truy cập hoặc kiểm soát trái phép.
- Dảm bảo dữ liệu truyền giữa thiết bị và hệ thống backend không bị nghe lén, sửa đổi hay giả mạo.
- Duy trì tính toàn vẹn và tin cậy của toàn bộ hệ sinh thái IoT trong suốt vòng đời thiết bị (từ sản xuất → triển khai → cập nhật → hủy bỏ).

Nói cách khác, bảo mật IoT không chỉ dừng ở phần mềm như bảo mật máy tính truyền thống, mà bao gồm cả:

- **Phần cứng:** root of trust, secure element, secure boot...
- **Giao thức kết nối:** TLS, mTLS, DTLS, OSCORE...
- **Quy trình vận hành:** OTA update, quản lý chứng chỉ, provisioning an toàn...

### 2.1.3 Tầm quan trọng của bảo mật IoT

Bảo mật IoT (Internet of Things) đóng vai trò quan trọng vì các thiết bị IoT ngày càng phổ biến, kết nối mọi thứ từ thiết bị gia dụng đến hệ thống công nghiệp. Dưới đây là những lý do chính giải thích tầm quan trọng của bảo mật IoT:

- Bảo vệ dữ liệu nhạy cảm: Các thiết bị IoT thu thập và truyền tải lượng lớn dữ liệu, bao gồm thông tin cá nhân, tài chính hoặc y tế. Nếu không được bảo mật, dữ liệu này có thể bị đánh cắp, dẫn đến vi phạm quyền riêng tư hoặc thiệt hại tài chính.
- Ngăn chặn truy cập trái phép: Thiết bị IoT thường có giao diện kết nối mạng, dễ trở thành mục tiêu cho tin tặc. Nếu bị xâm nhập, chúng có thể bị điều khiển để thực hiện các hành vi độc hại, như tấn công DDoS hoặc gián điệp.
- Đảm bảo an toàn vật lý: Nhiều thiết bị IoT điều khiển các hệ thống quan trọng (xe tự hành, thiết bị y tế, hệ thống điện). Lỗ hổng bảo mật có thể dẫn đến hậu quả nghiêm trọng, như tai nạn hoặc đe dọa tính mạng.
- Bảo vệ cơ sở hạ tầng: Các thiết bị IoT thường là một phần của mạng lưới lớn (nhà thông minh, thành phố thông minh, nhà máy). Một thiết bị bị xâm phạm có thể là cửa ngõ để tấn công toàn bộ hệ thống.
- Tuân thủ quy định pháp luật: Các quy định như GDPR, CCPA yêu cầu bảo vệ dữ liệu người dùng. Việc không đảm bảo bảo mật IoT có thể dẫn đến vi phạm pháp luật và bị phạt nặng.
- Duy trì niềm tin người dùng: Bảo mật yếu kém làm giảm niềm tin vào sản phẩm IoT, ảnh hưởng đến danh tiếng nhà sản xuất và sự phát triển của ngành.

### 2.1.4 Khác biệt của bảo mật IoT so với bảo mật thông thường

Sau khi tìm hiểu về bảo mật IoT, ta có thể thấy được sự khác biệt cơ bản so với bảo mật an ninh mạng thông thường, điều đó được thể hiện thông qua bảng sau:

Tiêu chí	Bảo mật hệ thống truyền thống	Bảo mật IoT
Dặc điểm thiết bị	Máy tính, laptop, server có tài nguyên mạnh, dễ triển khai bảo mật phức tạp.	Thiết bị đa dạng (cảm biến, thiết bị đeo, máy công nghiệp) với hạn chế về xử lý, bộ nhớ, pin.
Kết nối	Kết nối trong mạng an toàn (VPN, giao thức bảo mật chuẩn).	Thường kết nối trực tiếp Internet và dịch vụ đám mây, mở rộng bề mặt tấn công.
Quản lý dữ liệu	Lưu trữ tập trung, dễ mã hóa, phân quyền, giám sát.	Phân tán, xử lý thời gian thực, khó đảm bảo toàn vẹn và quyền riêng tư xuyên suốt.
Vòng đời và cập nhật	Có thể vá lỗi, cập nhật phần mềm thường xuyên.	Nhiều thiết bị ngoài thực địa, khó hoặc không thể cập nhật, dễ tồn tại lỗ hổng lâu dài.
Bối cảnh đe dọa	Malware, phishing, truy cập trái phép, có công cụ đối phó sẵn.	Nguy cơ chiếm quyền thiết bị, khai thác giao thức yếu, đánh cắp dữ liệu, gây hậu quả vật lý.
Quy định pháp lý	Tuân thủ chuẩn hiện hành (GDPR, HIPAA).	Tiêu chuẩn IoT đang phát triển, tập trung vào quyền riêng tư và an toàn thiết bị.

Table 1: So sánh bảo mật hệ thống truyền thống và bảo mật IoT

Nhìn từ tổng thể, an ninh IoT đòi hỏi một chiến lược mang tính chuyên sâu nhằm đối phó với các lỗ hổng đặc thù của thiết bị. Những giải pháp này phải bao trùm từ khâu quản lý thiết bị, duy trì kết nối an toàn cho tới bảo vệ dữ liệu, nhằm đảm bảo tính bền vững và an toàn của các ứng dụng IoT trong thực tế.



### 2.1.5 Giới thiệu về các tiêu chuẩn về an ninh hệ thống và an toàn thông tin trong IoT

#### 2.1.5.a Các tiêu chuẩn về an ninh hệ thống và an toàn thông tin trong IoT trên thế giới

##### ITU-T Y.4806:2017 - Khung năng lực bảo mật cho IoT

- Tổ chức ban hành: ITU-T (Liên minh Viễn thông Quốc tế)
- Phạm vi áp dụng: Các hệ thống IoT ở nhiều lĩnh vực (đô thị thông minh, công nghiệp, nông nghiệp, giao thông, y tế, năng lượng, nhà thông minh...); các thiết bị IoT (sensors, actuators, gateway, edge devices); các dịch vụ và ứng dụng IoT xử lý, truyền tải, hoặc lưu trữ dữ liệu.
- Nội dung chính: Xác định các năng lực mà hệ thống IoT cần có để hỗ trợ an toàn trong việc triển khai, vận hành.
- Dánh giá: Khung chuẩn toàn diện, linh hoạt, được quốc tế công nhận và nhấn mạnh bảo vệ an toàn con người; tuy nhiên còn thiếu chi tiết kỹ thuật, khó áp dụng trực tiếp cho thiết bị IoT giá rẻ và đòi hỏi hệ sinh thái đồng bộ cùng các chuẩn khác để triển khai hiệu quả.

##### ETSI EN 303 645 – Tiêu chuẩn IoT dành cho người tiêu dùng

- Tổ chức ban hành: ETSI (Viện Tiêu chuẩn Viễn thông Châu Âu)
- Phạm vi áp dụng: Thiết bị IoT tiêu dùng như camera, loa thông minh, TV, thiết bị đeo, router gia đình...
- Nội dung chính: Danh sách 13 khuyến nghị bảo mật tối thiểu (ví dụ: không dùng mật khẩu mặc định, cập nhật phần mềm an toàn, mã hóa truyền dữ liệu, bảo vệ quyền riêng tư...). Hướng đến việc thiết kế bảo mật ngay từ khi thiết kế
- Dánh giá: Thực tế, dễ áp dụng và được nhiều nước nội địa hóa, tuy nhiên chỉ áp dụng cho thiết bị tiêu dùng mà chưa bao phủ các lĩnh vực khác như IOT công nghiệp hay y tế.

##### NIST SP 800-213 và NIST IR 8259 Series – Khung bảo mật IoT (Hoa Kỳ)

- Tổ chức ban hành: NIST (Viện Tiêu chuẩn và Công nghệ Quốc gia Hoa Kỳ)
- Phạm vi áp dụng: Thiết bị IoT trong cả khu vực công và tư nhân.
- Nội dung chính: Xác định 6 năng lực bảo mật cốt lõi cho thiết bị IoT (ví dụ: xác thực, cập nhật phần mềm, bảo vệ cấu hình...). Hướng dẫn cho cả nhà sản xuất thiết bị lẫn tổ chức triển khai.
- Dánh giá: Rất đầy đủ, logic, mang tính kỹ thuật sâu nên vì vậy cũng khó tiếp cận thực tiễn tại các nước đang phát triển.

##### IETF SUIT & OTA – Chuẩn cập nhật phần mềm an toàn cho IoT

- Tổ chức ban hành: IETF (Lực lượng Đặc trách kỹ thuật Internet)
- Phạm vi áp dụng: Thiết bị IoT và hệ thống có nhu cầu cập nhật firmware/phần mềm từ xa (OTA) trong các lĩnh vực công nghiệp, y tế, giao thông, nông nghiệp, nhà thông minh...
- Nội dung chính: Định nghĩa SUIT Manifest – tệp metadata chuẩn chứa thông tin về bản cập nhật (phiên bản, chữ ký, hash, cách cài đặt). Đồng thời cung cấp thông tin để hỗ trợ cập nhật OTA an toàn: xác thực nguồn gốc, toàn vẹn dữ liệu, cơ chế rollback khi lỗi, tối ưu cho thiết bị tài nguyên hạn chế.
- Dánh giá: Khung chuẩn quốc tế giúp OTA diễn ra an toàn, giảm rủi ro firmware giả mạo và nâng cao khả năng quản lý vòng đời thiết bị IoT; tuy nhiên, chỉ đưa ra định dạng và cơ chế chung, cần được kết hợp với hạ tầng OTA cụ thể, và có thể gặp hạn chế với thiết bị IoT giá rẻ.



### 2.1.5.b Các tiêu chuẩn về an ninh hệ thống và an toàn thông tin trong IoT ở Việt Nam **Quyết định số 736/QĐ-BTTTT**

- Tổ chức ban hành: Bộ Thông tin và Truyền thông
- Phạm vi áp dụng: Thiết bị IoT tiêu dùng (thiết bị gia dụng, wearable, loa thông minh, camera, thiết bị sức khỏe...).
- Nội dung chính: 13 yêu cầu cơ bản về bảo mật dựa trên tiêu chuẩn ETSI EN 303 645, như: mật khẩu duy nhất, cập nhật phần mềm, bảo vệ dữ liệu, xác thực và lưu trữ an toàn.
- Dánh giá: Bám sát chuẩn quốc tế, dễ hiểu và thực tiễn, giúp nâng cao nhận thức bảo mật cho thị trường IoT tiêu dùng. Tuy nhiên, vì chỉ mang tính khuyến nghị nên hiệu lực hạn chế; thiết bị giá rẻ khó đáp ứng đầy đủ và chưa bao phủ các lĩnh vực IoT công nghiệp, y tế có yêu cầu cao hơn.

### **Quy chuẩn kỹ thuật quốc gia QCVN 135:2024/BTTTT**

- Tổ chức ban hành: Cục An toàn thông tin (Bộ TT&TT) biên soạn; Bộ KH&CN thẩm định; Bộ TT&TT ban hành.
- Phạm vi áp dụng: Camera giám sát sử dụng giao thức Internet (IP Camera).
- Nội dung chính: Quy định 11 nhóm yêu cầu kỹ thuật an toàn thông tin (mật khẩu, cập nhật, quản lý lỗ hổng, bảo vệ dữ liệu, xác thực đầu vào, kênh truyền an toàn, tự khôi phục... ).
- Dánh giá: Quy chuẩn kịp thời, cụ thể và khả thi, góp phần bảo vệ quyền riêng tư và thúc đẩy sản xuất “Made in Vietnam”. Tuy nhiên, triển khai diện rộng gặp khó do lượng thiết bị cũ lớn, chi phí tăng cho nhà sản xuất nhỏ và cần năng lực kiểm định, giám sát mạnh hơn.

### **Tiêu chuẩn Quốc gia TCVN 13749:2023**

- Tổ chức ban hành: Học viện Công nghệ Bưu chính Viễn thông biên soạn; Bộ TT&TT đề nghị; Bộ KH&CN ban hành.
- Phạm vi áp dụng: Hệ thống IoT có sự cố bảo mật có thể ảnh hưởng đến an toàn con người, xã hội, môi trường.
- Nội dung chính: Phân tích mối đe dọa và xác định năng lực bảo mật cần thiết, kết hợp giữa bảo mật và an toàn (safety).
- Dánh giá: Phù hợp với bối cảnh phát triển IoT tại Việt Nam, là bản nội địa hóa của ITU-T Y.4806:2017, giúp đồng bộ chuẩn quốc tế và hỗ trợ cơ quan, doanh nghiệp trong thiết kế, kiểm định hệ thống. Tuy nhiên, thiếu hướng dẫn kỹ thuật cụ thể, khó áp dụng cho thiết bị giá rẻ và chưa có cơ chế chứng nhận, giám sát đi kèm.

### 2.1.6 Nghiên cứu các nguy cơ, mối đe dọa đối với các thành phần trong hệ thống IoT

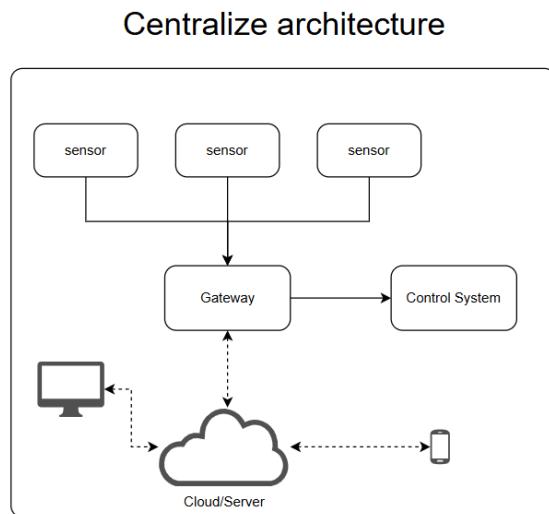


Figure 1: Ví dụ về kiến trúc hệ thống IOT

Nguy cơ và các mối đe dọa đến hệ thống IOT đến từ rất nhiều thành phần trong một hệ thống cơ bản: từ các thiết bị biên, cloud cho đến các ứng dụng IOT trên điện thoại. Dưới đây là các nhóm nguy cơ, mối đe dọa đặc thù trong các hệ thống IoT tại một số tầng liên quan.

#### 2.1.6.a Tầng thiết bị đầu cuối

- Phần cứng dễ bị truy cập vật lý: hacker có thể tháo, sao chép EEPROM, flash hoặc chèn mã độc vào firmware.
- Giới hạn tài nguyên: thiết bị nhỏ (MCU, sensor) khó triển khai các thuật toán mã hóa mạnh (AES, RSA...).
- Thiết bị giả mạo (Counterfeit device): tấn công chuỗi cung ứng.
- Thiếu cơ chế xác thực: nhiều cảm biến chỉ gửi dữ liệu "thô" mà không có chữ ký số hay hash kiểm tra toàn vẹn.
- Cập nhật firmware không an toàn: nguy cơ bị tấn công trong quá trình OTA update.

#### 2.1.6.b Tầng mạng

- Tấn công nghe lén (Eavesdropping / Sniffing): dữ liệu bị thu thập nếu không mã hóa (ví dụ: MQTT không TLS).
- Giả mạo và chiếm quyền (Spoofing, MITM): kẻ tấn công đóng giả thiết bị hoặc gateway hợp lệ.
- Tấn công từ chối dịch vụ (DoS/DDoS): gửi lưu lượng ảo làm nghẽn mạng.
- Routing attack (đối với mesh IoT): tấn công định tuyến như “sinkhole”, “wormhole”, “blackhole”.
- Replay attack: phát lại gói tin hợp lệ để lừa hệ thống (vd. mở cửa, kích hoạt thiết bị...).

#### 2.1.6.c Tầng nền tảng

- API của cloud service (AWS IoT, Azure IoT Hub...) có thể bị cấu hình sai hoặc lộ key
- Tấn công API: thiếu xác thực, lộ access token
- Tấn công quyền truy cập (Privilege escalation): khai thác lỗ trong hệ thống phân quyền người dùng.
- Rò rỉ dữ liệu người dùng: dữ liệu sao lưu trên cloud không được mã hóa đúng cách, dễ bị mất mát hoặc rò rỉ.



#### 2.1.6.d Tầng ứng dụng

- Giao diện người dùng thiếu bảo mật: Lộ thông tin nhạy cảm trên giao diện (ví dụ: hiển thị mật khẩu, tọa độ, ảnh từ camera...)
- Không minh bạch về quyền riêng tư: thu thập dữ liệu cá nhân vượt quá mức cần thiết, không có chính sách quyền riêng tư rõ ràng, ...
- Tấn công trên ứng dụng web/mobile: Ứng dụng không kiểm tra đầu vào, dễ bị tấn công SQL injection, XSS.
- Không có cơ chế kiểm soát và xóa dữ liệu cá nhân: người dùng không thể truy cập, sửa hoặc yêu cầu xóa dữ liệu

#### 2.1.6.e Tầng Quản lý và Chính sách

Dây là tầng đảm bảo an ninh tổng thể, không xử lý dữ liệu kỹ thuật trực tiếp như các tầng khác, mà tập trung vào: Thiết lập và giám sát chính sách an ninh; Quản lý vòng đời thiết bị; Dảm bảo tuân thủ tiêu chuẩn, audit, logging, backup; Quản lý khóa, chứng chỉ, tài khoản người dùng và chính sách cập nhật bảo mật. Các nguy cơ có thể kể ra như:

- Không cập nhật phần mềm định kỳ, không có OTA (Over-the-Air) an toàn.
- Không có cơ chế gỡ thiết bị khỏi hệ thống khi không còn sử dụng.
- Dữ liệu không bị xóa khi thiết bị bị loại bỏ.
- Không có hoặc không kiểm tra log hoạt động, nhật ký truy cập, cảnh báo bảo mật.

#### 2.1.7 Nghiên cứu các giải pháp an ninh hệ thống và an toàn thông tin cho IoT

Từ các nguy cơ, rủi ro đã được phân tích ở từng tầng như trên, các biện pháp an ninh cũng nên được nhóm theo từng tầng tương ứng để bao quát được cho toàn bộ hệ thống.

##### 2.1.7.a Tầng thiết bị

- Secure Boot nhẹ: dùng hash SHA-256 và khóa công khai nhúng trong ROM để chỉ cho phép firmware hợp lệ chạy.
- Mã hóa firmware và dữ liệu lưu trữ bằng AES-128.
- Chống sao chép phần cứng: gắn mã định danh duy nhất (UID/Chip ID) và liên kết với khóa mã hóa.
- Cập nhật OTA an toàn: firmware ký số (Ed25519/ECDSA) và xác thực trước khi ghi flash.
- Giới hạn quyền truy cập vật lý: tắt cổng debug (SWD/JTAG), niêm phong thiết bị.

##### 2.1.7.b Tầng mạng

- Sử dụng giao thức nhẹ có mã hóa: MQTTs, CoAPs (DTLS), hoặc LoRaWAN AES-128.
- Xác thực hai chiều (mTLS) cho thiết bị quan trọng
- Ngăn nghe lén / giả mạo: mã hóa end-to-end, nonce/timestamp chống replay.
- Giới hạn lưu lượng & phát hiện DoS: thiết lập rate-limit, dùng gateway làm lớp bảo vệ đầu tiên.
- Phân tách mạng: tách kênh IoT khỏi mạng quản trị



#### 2.1.7.c Tầng nền tảng và ứng dụng

- Xác thực người dùng bằng OAuth2.0, JWT short-lived token
- Mã hóa dữ liệu người dùng và sensor trước khi lưu vào cloud (AES hoặc field-level encryption).
- Phân quyền người dùng/thiết bị: RBAC/ABAC để giới hạn truy cập dữ liệu theo vùng, thiết bị.
- Kiểm thử bảo mật ứng dụng IoT mobile/web: chống SQLi, XSS, CSRF.
- Cung cấp dashboard an toàn: ẩn khóa API, log truy cập, và thông báo khi phát hiện truy cập lạ.

#### 2.1.7.d Tầng quản lý và chính sách

- Quản lý danh tính và khóa thiết bị (Device Identity & Key Lifecycle): đăng ký, cấp, thu hồi khóa.
- Chính sách cập nhật bắt buộc: thiết bị cũ phải tự động yêu cầu bản vá khi kết nối.
- Theo dõi và phản ứng sự cố: thu thập log tập trung, gửi cảnh báo qua MQTT/HTTPS tới SIEM.
- Đào tạo và quy trình vận hành an toàn: hướng dẫn kỹ thuật viên về OTA, xử lý sự cố, phân quyền.
- Áp dụng tiêu chuẩn: TCVN 13749:2023, IEC 62443-4-2, NISTIR 8259A,...



## 2.2 Về lý thuyết và công cụ phục vụ triển khai

### 2.2.1 Cơ sở lý thuyết

#### 2.2.1.a Secure Boot

Secure Boot (Khởi động An toàn) là một tính năng bảo mật giúp bảo vệ máy tính/phần cứng khỏi phần mềm độc hại ngay từ giai đoạn khởi động, bằng cách đảm bảo chỉ những phần mềm được ký bởi nhà sản xuất đáng tin cậy mới được phép chạy. Tính năng này sử dụng các chữ ký số để xác thực trình điều khiển và hệ điều hành, ngăn chặn các mã độc, rootkit hoặc phần mềm trái phép can thiệp vào quá trình khởi động hệ thống. Cụ thể, khi bật tính năng này bộ nạp khởi động (bootloader) sẽ:

- Kiểm tra chữ ký số của firmware.
- Chỉ cho phép chạy firmware đã được ký bằng khóa riêng hợp lệ.
- Nếu firmware bị chỉnh sửa, thay thế hoặc chưa ký, phần cứng sẽ từ chối khởi động.

Điều này đảm bảo tính toàn vẹn và xác thực (Integrity & Authenticity) của firmware, ngăn chặn các hành vi cài mã độc, giả mạo firmware hoặc thay đổi chức năng của thiết bị.

#### 2.2.1.b Flash Encryption

Flash Encryption, hay Mã hóa ổ đĩa flash, là quá trình bảo vệ dữ liệu trên ổ đĩa flash bằng cách mã hóa chúng, biến chúng thành dữ liệu không thể đọc được nếu không có mật khẩu hoặc khóa giải mã chính xác. Điều này ngăn chặn việc truy cập trái phép, bảo vệ thông tin nhạy cảm trong trường hợp ổ đĩa bị mất, bị đánh cắp hoặc bị tấn công mạng. Khi kích hoạt, phần cứng sẽ :

- Mã hóa toàn bộ vùng flash chứa firmware và dữ liệu người dùng.
- Tự động giải mã khi CPU truy xuất, không ảnh hưởng đến hiệu năng.
- Ngăn cản việc đọc hoặc sao chép firmware trực tiếp từ bộ nhớ ngoài.

Cơ chế này đảm bảo tính bí mật (Confidentiality) của firmware và dữ liệu lưu trữ trong thiết bị, ngay cả khi kẻ tấn công có quyền truy cập vật lý vào flash.

Khi chúng ta kết hợp cả hai hướng giải pháp trên, Secure Boot sẽ bảo vệ toàn vẹn firmware (chỉ chạy mã tin cậy), bên cạnh đó Flash Encryption bảo vệ bí mật firmware và dữ liệu (ngăn đọc trộm hoặc sao chép). Sự kết hợp này hình thành chuỗi khởi động an toàn (Trusted Boot Chain), đảm bảo hệ thống Smart Home nói riêng hay các hệ thống IoT nói chung được bảo vệ ngay từ lớp thiết bị.

#### 2.2.1.c ECDSA (Elliptic Curve Digital Signature Algorithm) thuật toán sinh chữ ký số dựa trên đường cong Elliptic.

ECDSA là một thuật toán mã hóa bắt đối xứng, hoạt động dựa trên cơ chế sử dụng hai khóa khác nhau – một khóa riêng (private key) và một khóa công khai (public key) – để đảm bảo tính toàn vẹn và xác thực của dữ liệu. Khác với các thuật toán mã hóa đối xứng như AES, nơi cùng một khóa được dùng cho cả mã hóa và giải mã, ECDSA tách biệt hoàn toàn hai chức năng này:

- Khóa riêng (private key) được dùng để tạo chữ ký số cho dữ liệu, chứng minh rằng dữ liệu đến từ nguồn hợp lệ.
- Khóa công khai (public key) được dùng để xác minh chữ ký, đảm bảo rằng nội dung chưa bị thay đổi và đúng là được ký bởi khóa riêng tương ứng.

Điểm quan trọng là: public key không thể giải mã hay khôi phục lại dữ liệu gốc. Nói cách khác, dù ai đó có nắm giữ khóa công khai, họ cũng chỉ có thể kiểm tra tính xác thực, chứ không thể giả mạo hay giải mã nội dung đã ký. Nhờ cơ chế này, ECDSA đảm bảo tính toàn vẹn, xác thực và chống giả mạo cho dữ liệu, đồng thời vẫn bảo vệ an toàn tuyệt đối cho khóa riêng – yếu tố nền tảng của mọi hệ thống chữ ký số và xác thực bảo mật hiện nay.

### Đường cong Elliptic:

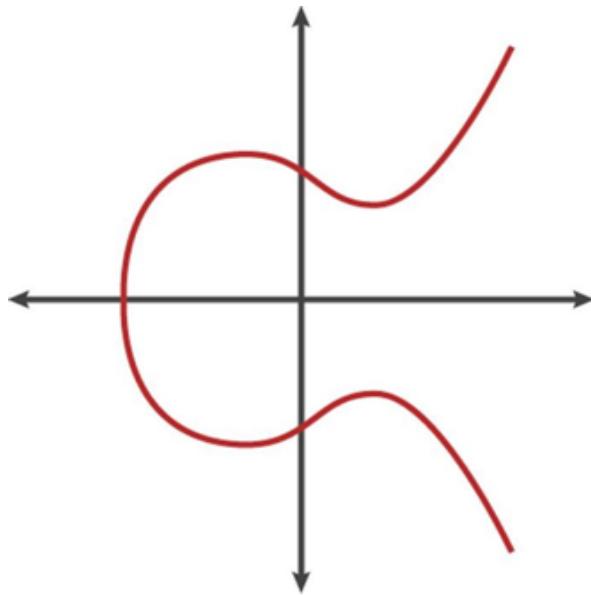


Figure 2: Đường cong Elliptic. [Nguồn](#).

Thuật toán ECDSA hoạt động dựa trên các phép toán đại số trên một đường cong Elliptic. Công thức tổng quát của đường cong Elliptic được biểu diễn như sau:

$$y^2 \pmod{p} = x^3 + ax + b \pmod{p}$$

Trong đó:

- $p$  là một số nguyên tố rất lớn, xác định không gian hữu hạn của các điểm trên đường cong.
- $a, b$  là các tham số định nghĩa hình dạng cụ thể của đường cong.

Các hệ thống như *Bitcoin* hay *Ethereum* sử dụng đường cong **secp256k1**, do Viện Tiêu Chuẩn và Kỹ Thuật Quốc Gia Hoa Kỳ (NIST) công bố. Đường cong này được định nghĩa bởi công thức:

$$y^2 \pmod{p} = x^3 + 7 \pmod{p}$$

với:

$$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$$

Đây là một đường cong đặc biệt an toàn, đảm bảo tính ngẫu nhiên cao và khả năng chống lại các tấn công giải mã *brute-force* nhờ không gian khóa khổng lồ (256 bit).

#### Các phép toán trên đường cong Elliptic:

Hai phép toán nền tảng trong ECDSA là **phép cộng** và **phép nhân điểm** trên đường cong.

##### a. Phép cộng hai điểm ( $P + Q$ )

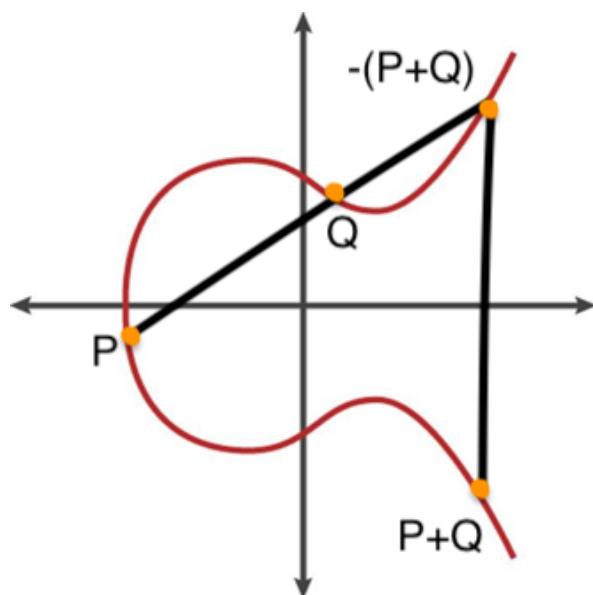


Figure 3: Phép cộng trên Elliptic. [Nguồn](#).

Nếu  $P$  và  $Q$  là hai điểm nằm trên đường cong Elliptic, thì đường thẳng đi qua  $P$  và  $Q$  sẽ cắt đường cong tại một điểm thứ ba  $-R$ . Lấy đối xứng điểm  $-R$  qua trục hoành, ta được điểm  $R = P + Q$ , và điểm này cũng nằm trên đường cong.

Tính chất đặc biệt này được biểu diễn hình học như sau:

- Vẽ đường thẳng nối hai điểm  $P$  và  $Q \rightarrow$  cắt đường cong tại điểm thứ ba.
- Lấy đối xứng điểm này qua trục  $x \rightarrow$  thu được kết quả  $P + Q$ .

Nếu ba điểm bất kỳ trên đường cong nằm trên cùng một đường thẳng, thì ta có:

$$P + Q + R = 0$$

### b. Phép nhân điểm ( $k \times P$ )

Phép nhân trên đường cong Elliptic thực chất là phép cộng lặp lại nhiều lần. Ví dụ, để tính  $3P$ :

- Đầu tiên tính  $2P = P + P$ .
- Sau đó,  $3P = 2P + P$ .

Việc thực hiện lặp phép cộng này nhiều lần sẽ tạo ra các điểm mới trên đường cong. Tuy nhiên, điều đáng chú ý là:

- Việc tính  $k \times P$  là dễ dàng.
- Nhưng việc tìm ngược lại  $k$  khi biết  $P$  và  $kP$  là cực kỳ khó.

Bài toán này được gọi là **Elliptic Curve Discrete Logarithm Problem (ECDLP)** — nền tảng đảm bảo tính an toàn của ECDSA.



Dễ hiểu rõ hơn về ECDSA, nhóm chúng em đã lập bảng so sánh giữa ECDSA và RSA:

Tiêu chí	ECDSA (Elliptic Curve Digital Signature Algorithm)	RSA (Rivest–Shamir–Adleman)
Kích thước khóa phổ biến	256 – 384 bit	2048 – 4096 bit
Độ bảo mật tương đương	ECDSA 256 bit ≈ RSA 3072 bit ECDSA 224 bit ≈ RSA 2048 bit	RSA 2048 bit ≈ ECDSA 224 bit
Hiệu suất và tốc độ	Nhanh hơn trong quá trình tạo khóa, ký và xác minh chữ ký	Chậm hơn, đặc biệt trong quá trình tạo khóa và ký
Sử dụng tài nguyên	Tiêu tốn ít năng lượng, bộ nhớ và phép tính hơn → Phù hợp cho thiết bị IoT, nhúng	Tiêu tốn nhiều CPU, bộ nhớ và năng lượng → Phù hợp cho máy chủ, hệ thống mạnh
Mức độ an toàn	Bảo mật cao với khóa nhỏ (dựa trên bài toán logarit rời rạc trên đường cong elliptic)	Cần khóa lớn để đạt cùng mức bảo mật (dựa trên bài toán phân tích số nguyên lớn)
Khả năng tương thích	Hỗ trợ mạnh trong các hệ thống hiện đại (TLS, Blockchain, IoT, WebAuthn) Nhưng hạn chế hơn với hệ thống cũ	Phổ biến và tương thích rộng với hầu hết mọi hệ thống, kể cả nền tảng cũ
Ứng dụng điển hình	Blockchain (Bitcoin, Ethereum), TLS hiện đại, SSH, chữ ký số IoT	HTTPS, SSL/TLS cũ, chứng chỉ số, chữ ký điện tử truyền thống
Ưu điểm chính	Bảo mật cao với khóa nhỏ, hiệu suất cao, tiết kiệm tài nguyên	Dễ triển khai, phổ biến, tương thích rộng
Nhược điểm chính	Phức tạp về toán học, khó triển khai chính xác	Kích thước khóa và chữ ký lớn, hiệu suất thấp

Table 2: So sánh giữa ECDSA và RSA

Qua so sánh, khuyến khích chọn ECDSA khi ta làm việc với các thiết bị có tài nguyên tính toán hạn chế, chẳng hạn như điện thoại thông minh hoặc thiết bị IoT. Đó cũng là lý do bạn em sẽ sử dụng ECDSA trong nội dung đề tài này.

### 2.2.2 Công cụ phục vụ triển khai

#### 2.2.2.a Phần mềm

Dể thực hiện các thao tác can thiệp sâu vào cấu hình phần cứng (eFuse) và quy trình khởi động, nhóm sử dụng bộ công cụ phát triển chính thức (Native SDK) của Espressif. Chi tiết được trình bày trong Bảng ??.



Công cụ/Phần mềm	Chức năng chính
ESP-IDF	<ul style="list-style-type: none"><li><b>Espressif IoT Development Framework:</b> Khung phát triển phần mềm chính thức cho ESP32.</li><li><b>idf.py menuconfig:</b> Công cụ giao diện cấu hình dự án, dùng để bật các tính năng bảo mật (CONFIG_SECURE_BOOT, CONFIG_FLASH_ENCRYPTION) trong sdkconfig.</li><li><b>Build System:</b> Quản lý quy trình biên dịch toàn bộ hệ thống, tự động ký số firmware khi build nếu đã cấu hình.</li></ul>
esptool.py	<ul style="list-style-type: none"><li>Công cụ nạp (Flasher) giao tiếp trực tiếp với Bootloader trong ROM.</li><li>Được tích hợp trong ESP-IDF nhưng cũng được sử dụng độc lập để: nạp Bootloader Digest, nạp firmware thủ công tại các địa chỉ cụ thể.</li><li>Dùng để đọc (dump) dữ liệu Flash (read_flash) nhằm kiểm chứng dữ liệu đã được mã hóa hay chưa.</li></ul>
espefuse.py	<ul style="list-style-type: none"><li>Công cụ chuyên dụng để thao tác với vùng nhớ eFuse (OTP).</li><li>Chức năng trong đê tài:<ul style="list-style-type: none"><li>- burn_key: Nạp vĩnh viễn khóa Secure Boot (Block 2) và Flash Encryption (Block 1).</li><li>- burn_efuse: Kích hoạt cờ bảo mật ABS_DONE_0 (cho Secure Boot V1) và FLASH_CRYPT_CNT.</li><li>- summary: Kiểm tra trạng thái bảo mật của chip trước và sau khi thực hiện.</li></ul></li></ul>
espsecure.py	<ul style="list-style-type: none"><li>Công cụ mật mã hỗ trợ tạo khóa và ký số.</li><li>Chức năng trong đê tài:<ul style="list-style-type: none"><li>- generate_signing_key: Tạo khóa riêng tư ECDSA.</li><li>- digest_secure_bootloader: Tạo chuỗi tóm tắt và vector ngẫu nhiên (IV) cho Bootloader.</li><li>- encrypt_flash_data: Mã hóa thủ công firmware trên máy tính (Pre-encryption) để nạp cập nhật mà không làm tăng bộ đếm eFuse.</li></ul></li></ul>
Python & Serial Monitor	<ul style="list-style-type: none"><li>Môi trường thực thi các script của ESP-IDF.</li><li>Sử dụng Serial Monitor (trong VS Code hoặc Terminal) để theo dõi log khởi động (Boot log), phân tích các mã lỗi như secure boot check fail hoặc trạng thái flash encryption is enabled.</li></ul>

Table 3: Các công cụ và chức năng chính sử dụng trong bảo mật ESP32

### 2.2.2.b Phản ứng

Hệ thống sử dụng thiết bị trung tâm là bộ kit giáo dục Yolo:Bit tích hợp sẵn vi điều khiển ESP32 phục vụ cho việc minh họa các cơ chế bảo mật.

#### • Bộ kit Yolo:Bit

- **Vi điều khiển trung tâm:** ESP32-WROVER-B (Dual-core, 240MHz).
- **Bộ nhớ:** 4MB Flash (lưu trữ Firmware và thực hiện Flash Encryption).
- **Kết nối mạng:** Wi-Fi 802.11 b/g/n và Bluetooth Classic/LE.
- **Vai trò trong đê tài:** Thiết bị mục tiêu (Target Device) để triển khai Secure Boot V1 và Flash Encryption; thực hiện xử lý mã hóa và xác thực phản ứng.

### 3 Phân tích và thiết kế

#### 3.1 Mục tiêu thiết kế bảo mật

Dể đảm bảo Smart Home vận hành an toàn, nhóm tập trung vào bảo vệ firmware và dữ liệu bên trong thiết bị IoT (ESP32) — vì đây là nơi dễ bị tấn công vật lý nhất. Các yêu cầu chính đặt ra gồm:

- Toàn vẹn (Integrity): chỉ firmware hợp lệ mới được phép chạy.
- Xác thực (Authenticity): firmware phải được ký bởi nhà phát triển tin cậy.
- Bí mật (Confidentiality): nội dung chương trình và dữ liệu flash phải được mã hóa.
- Không thể đảo ngược (Irreversibility): khi bật các tính năng bảo mật, chúng không thể bị tắt lại (one-time programmable via eFuse).

Từ đó, hệ thống bảo mật được thiết kế theo chuỗi tin cậy khởi động (Trusted Boot Chain) :

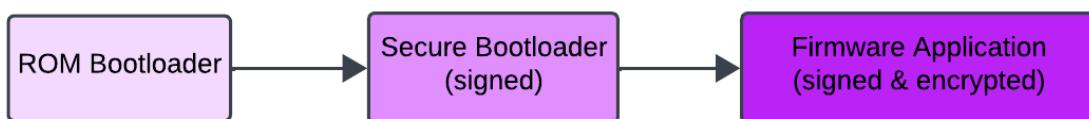


Figure 4: Chuỗi tin cậy khởi động

#### 3.2 Phân tích SoC ESP32 và các module phần cứng liên quan

Mạch Yolo:Bit sử dụng ESP32-Wrover B, thuộc dòng chip revision v1.1, một dòng đời đầu, do đó ROM trong SoC của ESP32, không hỗ trợ Secure Boot v2. Trong đề tài này, vì giới hạn phần cứng nên sẽ triển khai theo Secure boot v1. Các module SoC quan trọng có liên quan và cần làm rõ trong đề tài này bao gồm :

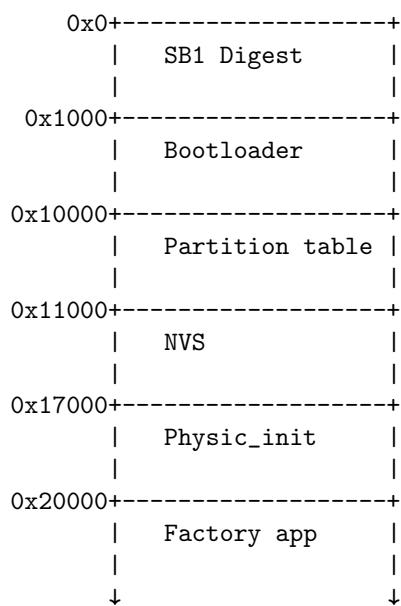
- Bộ nhớ Internal ROM Đây là bộ nhớ chỉ đọc chứa mã khởi động bất biến. Trên Revision 1, ROM tích hợp sẵn các hàm để gọi phần cứng SHA-512. Khi khởi động, ROM đóng vai trò "Root of Trust" với nhiệm vụ:
  - Sử dụng khóa bảo mật trong eFuse để mã hóa nội dung Bootloader bằng thuật toán AES-256.
  - Tính toán mã băm SHA-512 của dữ liệu đã mã hóa đó để tạo ra một chuỗi tóm tắt (Digest) dài 512-bit.
  - So sánh chuỗi này với Digest đã được nạp tại địa chỉ 0x0 trên Flash.
- Efuse: là vùng nhớ chỉ ghi một lần, đóng vai trò then chốt để lưu trữ cấu hình bảo mật không thể đảo ngược. Đối với Secure Boot V1, các Block quan trọng cần cấu hình gồm:
  - BLOCK\_0 (Security configuration):
    - \* ABS\_DONE\_0: Đây là bit cờ quan trọng nhất để kích hoạt Secure Boot V1 (khác với V2 dùng bit ABS\_DONE\_1). Khi bit này được đốt (burn), tính năng Secure Boot V1 sẽ có hiệu lực vĩnh viễn.
    - \* FLASH\_CRYPT\_CNT: Bộ đếm kiểm soát tính năng mã hóa Flash Encryption (nhóm sử dụng song song với Secure Boot). Đồng thời, bit DISABLE\_DL\_DECRYPT được bật để khóa cổng nạp debug.
    - \* DISABLE\_DL\_DECRYPT & DISABLE\_DL\_CACHE: Các bit khóa cổng nạp UART (Download Mode) để chống trích xuất dữ liệu trái phép.
  - BLOCK\_1: Lưu trữ khóa AES (256 bit) cho Flash Encryption.
  - BLOCK\_2: Lưu trữ AES-256 dùng để tính toán Bootloader Digest. Khóa này không thể đọc ra ngoài (Read Protected).

Các vùng này chỉ ghi một lần và không thể thay đổi sau khi ghi, đảm bảo an toàn tuyệt đối. Điều này cũng đáp ứng được yêu cầu đã đặt ra ở trên đó là "Không thể đảo ngược".

- Bộ tăng tốc phần cứng (Hardware Accelerators) SoC ESP32 Rev 1 tích hợp sẵn các module phần cứng chuyên dụng:

- SHA-512 Accelerator: Dùng cho ROM để xác thực Bootloader.
- SHA-256 Accelerator: Dùng cho Bootloader phần mềm để xác thực chữ ký ECDSA của Firmware ứng dụng.
- AES Accelerator: Dùng để giải mã dữ liệu Flash (Flash Encryption) trong thời gian thực

SoC ESP32 sẽ giao tiếp với bộ nhớ ngoài lớn hơn, để nạp code từ người dùng là SPI Flash. Trong dự án này, việc hiểu rõ bản đồ bộ nhớ (Memory Map) là bắt buộc để nạp các thành phần đã ký (signed binaries) vào đúng địa chỉ. Cấu trúc Flash trong đê tài này của nhóm bao gồm các thành phần chính tại các địa chỉ (Offset) cố định sau:



- Bootloader Digest (Offset  $0 \times 0000$ ): Đây là điểm khác biệt cốt lõi của Secure Boot V1. Vùng này chứa chuỗi tóm tắt (digest) của Bootloader. Khi khởi động, ROM sẽ tính toán lại digest của Bootloader đang nằm tại  $0 \times 1000$  và so sánh với digest lưu tại đây. Nếu khớp, Bootloader mới được phép chạy. (Secure Boot V2 không sử dụng vùng này).
- Second Stage Bootloader (Offset  $0 \times 1000$ ): Chương trình khởi động giai đoạn 2. Nhiệm vụ của nó là đọc bảng phân vùng và nạp Firmware ứng dụng. Trong đê tài này, Bootloader này được biên dịch hỗ trợ giải mã Flash Encryption để có thể đọc được partition table và app đã mã hóa.
- Partition Table (Offset  $0 \times 10000$ ): Bảng phân vùng đóng vai trò như bản đồ chỉ dẫn, định nghĩa địa chỉ bắt đầu và kích thước của các vùng dữ liệu khác (NVS, Factory App...). Việc xác định đúng địa chỉ này rất quan trọng để tránh ghi đè dữ liệu khi nạp chồng (Over-the-Air).
- NVS & PHY Data (Offset  $0 \times 11000$ ): Vùng lưu trữ dữ liệu cấu hình hệ thống (Non-Volatile Storage) và dữ liệu khởi tạo tầng vật lý (RF calibration).
- Factory App / Firmware (Offset  $0 \times 20000$ ): Đây là nơi chứa file nhị phân (.bin) của chương trình ứng dụng chính. Đối với Secure Boot V1, firmware tại đây phải được ký số (Digital Signature) bằng thuật toán ECDSA. Bootloader sẽ kiểm tra chữ ký này trước khi trao quyền điều khiển cho ứng dụng.

### 3.3 Kiến trúc bảo mật tổng thể

Trong hệ thống Smart Home được triển khai, vi điều khiển ESP32 DevKit V1 đóng vai trò trung tâm, vừa thu thập dữ liệu cảm biến, vừa điều khiển thiết bị đầu ra, đồng thời đảm nhiệm các cơ chế bảo mật phần cứng (hardware security). Cụ thể, hai cơ chế quan trọng được áp dụng là:

- Secure Boot (V1): đảm bảo chỉ firmware đã được ký hợp lệ mới có thể chạy.
- Flash Encryption: đảm bảo mã máy trong flash không thể bị đọc hoặc sao chép trái phép.

Toàn bộ quy trình hoạt động có thể mô tả qua kiến trúc sau:

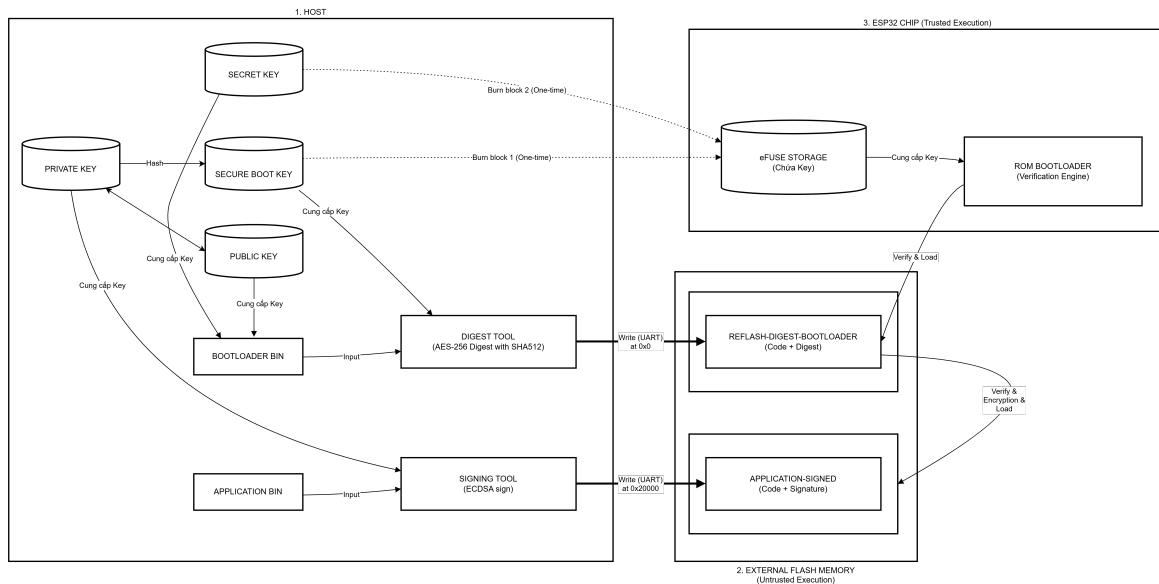


Figure 5: Sơ đồ kiến trúc bảo mật tổng thể

1. Phía Host (Quá trình biên dịch và ký số) Khác với quy trình nạp thông thường, Secure Boot V1 yêu cầu xử lý đặc biệt cho cả Bootloader và Firmware.

- Xử lý Bootloader (Cơ chế Digest):
  - File **bootloader.bin** sau khi biên dịch sẽ được xử lý tool **espsecure** cùng với khóa Secure Boot Key (256-bit).
  - Hệ thống tính toán theo công thức:  

$$Digest = SHA512(AES256(Bootloader, Key))$$
  - Kết quả tạo ra file **bootloader-reflash-digest.bin** để nạp vào địa chỉ **0x0**
  - Đồng thời, bootloader được cung cấp public key để xác thực firmware và được cấu hình để sử dụng AES Accelerator để mã hóa bộ nhớ flash.
- Xử lý Firmware (Cơ chế Ký số)
  - File **firmware.bin** được ký số bằng khóa riêng (Private Key) theo thuật toán ECDSA P-256.
  - Kết quả tạo ra file **firmware-signed.bin** chứa chữ ký số ở cuối file.

2. Phía Device (Quy trình khởi động an toàn) Khi cấp nguồn, ESP32 thực thi quy trình "Root of Trust" từ phần cứng, kết hợp với bộ giải mã Flash thời gian thực:

- Tầng 1: Hardware Root of Trust (ROM)
  - ROM (mã cứng bất biến trong silicon) khởi chạy đầu tiên.
  - ROM đọc **Bootloader Digest** từ địa chỉ Flash **0x0** và sử dụng khóa trong eFuse Block 2 để kiểm tra tính toàn vẹn của Bootloader phần mềm (tại **0x1000**).
  - Quá trình này sử dụng phần cứng SHA-512 Accelerator. Nếu xác thực thành công, ROM chuyển quyền điều khiển cho Bootloader.
- Tầng 2: Software Chain of Trust (Bootloader)
  - Bootloader (đã được tin cậy ở Tầng 1) tiếp tục kiểm tra chữ ký số của Firmware Ứng dụng (tại **0x10000**).
  - Nó sử dụng Public Key (được nhúng sẵn) để xác minh chữ ký ECDSA. Nếu hợp lệ, ứng dụng được phép khởi chạy.



- Cơ chế Flash Encryption (Bảo vệ dữ liệu):
  - Song song với quá trình khởi động, module AES-256 Accelerator hoạt động ở tầng vật lý để giải mã dữ liệu Flash "trong suốt" (Transparent Decryption).
  - Toàn bộ dữ liệu trên Flash (bao gồm Bootloader, Partition Table, Firmware và NVS) đều được lưu trữ dưới dạng mã hóa (Ciphertext). CPU chỉ nhìn thấy dữ liệu rõ (Plaintext) khi truy xuất qua bộ đệm Cache, trong khi hacker đọc trực tiếp Flash sẽ chỉ thu được dữ liệu vô nghĩa.
- 3. Phân bố dữ liệu trên Flash (Flash Layout) Để kiến trúc trên hoạt động đồng bộ, bộ nhớ Flash được tổ chức lại với các địa chỉ (Offset) cố định:
  - **0x0000:** `bootloader-reflash-digest.bin` (Chứa Digest 512-bit để ROM kiểm tra).
  - **0x1000:** `bootloader.bin` (Encrypted - Bootloader giai đoạn 2).
  - **0x10000:** `partitions.bin` (Encrypted - Bảng phân vùng).
  - **0x20000:** `firmware-signed.bin` (Encrypted & Signed - Ứng dụng chính).



## 4 Hiện thực và đánh giá

Dể xem chi tiết mã nguồn ứng dụng demo (bao gồm các tác vụ kết nối Wifi và MQTT), vui lòng tham khảo Phụ lục ở cuối báo cáo. Trong phạm vi phần này, nhóm chỉ tập trung phân tích các thiết lập bảo mật và kết quả thực thi.

### 4.1 Hiện thực secure boot version 1

- Cấu hình bootloader hỗ trợ secure boot version 1.

Trong phần hiện thực, nhóm đã thiết lập các thông số trong sdkconfig để kích hoạt Secure Boot V1 ở chế độ Reflashable. Cụ thể:

Tên cấu hình	Giá trị	Ý nghĩa và Tác dụng
CONFIG_SECURE_BOOT	y	Kích hoạt tính năng Secure Boot. Công tắc tổng bật quy trình khởi động an toàn.
CONFIG_SECURE_BOOT_V1_ENABLED	y	Chọn phiên bản Secure Boot V1 (cho ESP32 cũ/Rev 1). Dùng chữ ký ECDSA và digest AES-256.
CONFIG_SECURE_BOOTLOADER_REFFLASHABLE	y	Chế độ Bootloader có thể nạp lại. - Cho phép update bootloader sau khi bảo mật thay vì khóa chết bootloader (One-time flash). - Yêu cầu file <code>bootloader-reflash-digest.bin</code> tại 0x0.
CONFIG_SECURE_SIGNED_ON_BOOT	y	Bắt buộc kiểm tra chữ ký khi khởi động. Bootloader xác thực firmware, nếu sai sẽ không chạy.
CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES	y	Tự động ký Binary. Hệ thống build tự động ký <code>app.bin</code> và <code>bootloader.bin</code> khi biên dịch.
CONFIG_SECURE_BOOT_SIGNING_KEY	"secure_boot_signing_key.pem"	Đường dẫn đến khóa ký (Private Key). File này được tạo bằng lệnh <code>espsecure.py</code> .
CONFIG_SECURE_SIGNED_APPS_ECDSA_SCHEME	y	Thuật toán ký ECDSA. Sử dụng để ký và xác thực firmware ứng dụng.
CONFIG_SECURE_BOOTLOADER_KEY_ENCODING_256BIT	y	Độ dài khóa Bootloader. Sử dụng khóa AES-256 bit để bảo vệ Bootloader.

Table 5: Cấu hình secure boot version 1 trong sdkconfig

- Tạo khóa bí mật (Private key) cho việc hiện thực secure boot

```
1 espsecure.py generate_signing_key secure_boot_signing_key.pem
```

Ý nghĩa kĩ thuật của việc tạo khóa trên:

- Khởi tạo Khóa gốc (Root of Trust): Câu lệnh sử dụng công cụ `espsecure.py` để sinh ra một khóa bí mật (Private Key) mới và lưu vào file `secure_boot_signing_key.pem`. Theo mặc định của ESP-IDF, khóa này sử dụng thuật toán ECDSA (Elliptic Curve Digital Signature Algorithm), đây là thuật toán mã hóa bắt đầu xứng, đảm bảo độ bảo mật cao nhưng tối ưu về hiệu năng cho các thiết bị nhúng.
- Vai trò trong kiến trúc Secure Boot V1 (Reflashable Mode): Trong cấu hình "Reflashable" (có thể cập nhật lại Bootloader) mà hệ thống đang triển khai, khóa này đóng vai trò trung tâm và quan trọng nhất:
  - Dùng để ký Firmware: Khóa này được dùng để tạo chữ ký số cho các bản Firmware ứng dụng (`app.bin`) và Bootloader (`bootloader.bin`). Bootloader khi chạy sẽ dùng khóa công khai (Public Key) tương ứng để xác thực tính toàn vẹn và nguồn gốc của ứng dụng trước khi cho phép khởi động.
  - Sinh khóa phần cứng: Điểm đặc biệt của chế độ Reflashable V1 là Secure Boot Key (AES-256) – khóa được nạp cứng vào eFuse của chip – không được sinh ngẫu nhiên mà được tạo ra bằng cách băm (hash SHA-256) chính file `secure_boot_signing_key.pem` này.



- Cơ chế tin cậy: Chỉ người nắm giữ file này mới có thể tạo ra được khóa AES-256 hợp lệ để vượt qua lớp bảo vệ phần cứng (ROM Bootloader) và tạo ra chữ ký hợp lệ để vượt qua lớp bảo vệ phần mềm (Software Bootloader).
- Yêu cầu bảo mật: Việc mất file này đồng nghĩa với việc mất khả năng cập nhật firmware cho thiết bị trong tương lai (do không thể tạo ra chữ ký khớp với khóa đã nạp trong chip). Ngược lại, nếu lộ file này, kẻ tấn công có thể tạo ra phần mềm độc hại giả mạo hợp lệ để nạp vào thiết bị.



### 3. Build Bootloader

Tiếp theo ta tiến hành build bootloader với câu lệnh:

```
1 idf.py bootloader
```

```
Compiler supported targets: xtensa-esp32-elf
-- Looking for sys/types.h
-- Looking for sys/types.h - found
-- Looking for stdint.h
-- Looking for stdint.h - found
-- Looking for stddef.h
-- Looking for stddef.h - found
- Check size of time_t
- Check size of time_t - done
-- Adding linker script c:/users/quynh/esp/v5.1/esp-idf/components/soc/esp32/layersp32_peripherals.ld
- App "bootloader" version: v5.1.6-1361-g20dd7802e
-- Adding linker script c:/users/quynh/esp/v5.1/esp-idf/components/esp_rom/esp32/layersp32_rom.ld
-- Adding linker script c:/users/quynh/esp/v5.1/esp-idf/components/esp_rom/esp32/layersp32_rom.api.ld
-- Adding linker script c:/users/quynh/esp/v5.1/esp-idf/components/esp_rom/esp32/layersp32_rom.libgcc.ld
-- Adding linker script c:/users/quynh/esp/v5.1/esp-idf/components/esp_rom/esp32/layersp32_rom.newlib-funcs.ld
-- Adding linker script c:/users/quynh/esp/v5.1/esp-idf/components/bootloader/subproject/main/layersp32_bootloader.ld
-- Adding linker script c:/users/quynh/esp/v5.1/esp-idf/components/bootloader/subproject/main/layersp32_bootloader.ld
-- Components: bootloader bootloader_support efuse esp_app_format esp_common esp_hw_support esp_rom esp_system esptool_py freertos hal log main micro-ecc newlib partition_table soc spi_flash xtensa
-- Component paths: C:/users/quynh/esp/v5.1/esp-idf/components/bootloader C:/users/quynh/esp/v5.1/esp-idf/components/bootloader_support C:/users/quynh/esp/v5.1/esp-idf/components/efuse C:/users/quynh/esp/v5.1/esp-idf/components/esp_app_format C:/users/quynh/esp/v5.1/esp-idf/components/esp_common C:/users/quynh/esp/v5.1/esp-idf/components/esp_hw_support C:/users/quynh/esp/v5.1/esp-idf/components/esp_rom C:/users/quynh/esp/v5.1/esp-idf/components/esp_system C:/users/quynh/esp/v5.1/esp-idf/components/esptool_py C:/users/quynh/esp/v5.1/esp-idf/components/freertos C:/users/quynh/esp/v5.1/esp-idf/components/hal C:/users/quynh/esp/v5.1/esp-idf/components/log C:/users/quynh/esp/v5.1/esp-idf/components/bootloader/subproject/main C:/users/quynh/esp/v5.1/esp-idf/components/partition_table C:/users/quynh/esp/v5.1/esp-idf/components/components/soc C:/users/quynh/esp/v5.1/esp-idf/components/spi_flash C:/users/quynh/esp/v5.1/esp-idf/components/xtensa
-- Configuring done (7.7s)
-- Generating done (0.2s)
-- Build files have been written to: D:/Study/Cryptoraphy/YOLO_BIT/build/bootloader
[35/114] Generating secure-bootloader-key-256.binespsecure.py v4.10.0
SHA-256 digest of private key D:/Study/Cryptoraphy/YOLO_BIT/secure_boot_signing_key.pem written to D:/Study/Cryptoraphy/YOLO_BIT/build/bootloader/secure-bootloader-key-256.bin
[138/114] Generating signature_verification_key_binespsecure_nv_v4.10.0
D:/Study/Cryptoraphy/YOLO_BIT/secure_boot_signing_key.pem public key extracted to D:/Study/Cryptoraphy/YOLO_BIT/build/bootloader/esp-idf/bootloader_support/signature_verification_key.bn
```

Figure 6: Kết quả Build Bootloader (1)

```
* After first boot, only re-flashes of this kind (with same key) will be accepted.
* Not recommended to re-use the same secure boot keyfile on multiple production devices.
[112/114] Generating binary image from built executable esptool.py v4.10.0
Creating esp32 image...
Merged 1 ELF section
Successfully created esp32 image.
Generated D:/Study/Cryptoraphy/YOLO_BIT/build/bootloader/bootloader.bin
[113/114] C:\Windows\system32\cmd.exe /C "cd /D D:/Study/Cryptoraphy/YOLO_BIT/build\bootlo...et 0x1000 bootloader 0x1000 D:/Study/Cryptoraphy/YOLO_BIT/build/bootloader/bootloader.bin"
bootloader binary size 0x9e0 bytes, 0x5160 bytes (3%) free.
[114/114] Generating bootloader-reflash-digest.binDIGEST D:/Study/Cryptoraphy/YOLO_BIT/build/bootloader/bootloader-reflash-digest.bin
espssecure.py v4.10.0
Using 256-bit key
digest+image written to D:/Study/Cryptoraphy/YOLO_BIT/build/bootloader/bootloader-reflash-digest.bin
[10/10] Completed 'bootloader'
Bootloader build complete.
```

Figure 7: Kết quả Build Bootloader (2)

Dựa trên log hệ thống, các cơ chế bảo mật đã được tự động kích hoạt và xử lý theo trình tự sau:

- Tự động sinh khóa Secure Boot (Key Derivation):
  - Dẫn chứng (Hình 8): Dòng log SHA-256 digest of private key ... written to ... secure-bootloader-key-256.bin.
  - Giải thích: Do cấu hình ở chế độ Reflashable, hệ thống không sinh khóa AES-256 ngẫu nhiên. Thay vào đó, nó sử dụng thuật toán băm SHA-256 lên file khóa ký (secure\_boot\_signing\_key.pem) để tạo ra khóa bảo vệ Bootloader (secure-bootloader-key-256.bin). Đây là cơ chế cốt lõi giúp tái tạo lại khóa bảo mật nếu cần nạp lại Bootloader sau này.
- Trích xuất và nhúng khóa công khai (Public Key Embedding):
  - Dẫn chứng (Hình 8): Dòng log ...public key extracted to ... signature\_verification\_key.bn.
  - Giải thích: Hệ thống trích xuất khóa công khai (Public Key) từ cặp khóa ECDSA và biến dịch nó vào trong mã nguồn của Bootloader. Nhờ đó, khi khởi động, Bootloader có khả năng tự xác thực chữ ký số của Firmware ứng dụng mà không cần truy cập ra bên ngoài.
- Tạo file Digest cho Bootloader (Digest Generation):
  - Dẫn chứng (Hình 9): Dòng log Generating bootloader-reflash-digest.bin và Using 256-bit key.
  - Giải thích: Đây là bước đặc thù của Secure Boot V1. Công cụ espssecure.py đã sử dụng khóa AES-256 (vừa tạo ở bước 1) để tính toán chuỗi tóm tắt (digest) cho Bootloader. Kết quả là file bootloader-reflash-digest.bin.



#### 4. Build Firmware

Bước kế tiếp ta tiến hành build firmware bằng câu lệnh:

```
1 idf.py build
```

```
[1561/1563] Generating binary image from built executable esp tool.py v4.10.0
Creating esp32 image...
Merged 27 ELF sections
Successfully created esp32 image
Generated D:/Study/Cryptology/YOLO_BIT/build/YOLO_BIT-unsigned.bin
[1562/1563] Generating signed binary image espsecure.py v4.10.0
Signed 917424 bytes of data from D:/Study/Cryptology/YOLO_BIT/build/YOLO_BIT-unsigned.bin
Generated signed binary image D:/Study/Cryptology/YOLO_BIT/build/YOLO_BIT.bin from D:/Study/Cryptology/YOLO_BIT/build/partition_table/partition-table.bin D:/Study/Cryptology/YOLO_BIT/build/YOLO_BIT.bin
[1563/1563] C:\Windows\system32\cmd.exe /C cd /D D:/Study/Cryptology/YOLO_BIT/build&esp-idf\esp_idf\components\esp tool_py\esp tool\esp tool.py -p (PORT) -b 460800 --before default_reset
--after no_reset --chip esp32 write_flash --flash_mode dio --flash_size 4MB --flash_freq 40m 0x10000 build\partition_table\partition-table.bin 0x20000 build\YOLO_BIT.bin
YOLO_BIT.bin binary size 0xdfff4 bytes. Smallest app partition is 0x10000 bytes. 0x2000c bytes (1%) free.

Project build complete. To flash, run this command:
C:\Users\quynh\.espresif\python_env\idfs\1.py3.11_env\Scripts\python.exe C:/Users/quynh/esp/v5.1/esp-idf\components\esp tool_py\esp tool\esp tool.py -p (PORT) -b 460800 --before default_reset
--after no_reset --chip esp32 write_flash --flash_mode dio --flash_size 4MB --flash_freq 40m 0x10000 build\partition_table\partition-table.bin 0x20000 build\YOLO_BIT.bin
or run 'idf.py -p (PORT) flash'
```

Figure 8: Build Firmware

Hệ thống build của ESP-IDF đã tự động thực hiện quy trình đóng gói và ký số bảo mật như đã cấu hình trong sdkconfig. Cụ thể:

- Tạo Binary thô (Unsigned Binary Generation):
  - Dẫn chứng: Dòng log Generated .../YOLO\_BIT-unsigned.bin.
  - Giải thích: Hệ thống biên dịch mã nguồn thành file nhị phân YOLO\_BIT-unsigned.bin. Tại thời điểm này, file chứa mã máy hoàn chỉnh nhưng chưa có chữ ký bảo mật. Nếu nạp file này vào một thiết bị đã bật Secure Boot, thiết bị sẽ từ chối khởi động.
- Ký số Firmware (Digital Signing Process):
  - Dẫn chứng: Dòng log Signing 917424 bytes of data... và Generating signed binary image espsecure.py....
  - Giải thích: Hệ thống tự động gọi công cụ espsecure.py và sử dụng khóa bí mật ECDSA Private Key (secure\_boot\_signing\_key.pem) đã cấu hình trước đó để tính toán và ký lên file binary thô. Đây là bước hiện thực hóa tính năng CONFIG\_SECURE\_BOOT\_BUILD\_SIGNED\_BINARIES đã bật trong cấu hình dự án.
- Tạo Binary hoàn chỉnh (Final Signed Binary):
  - Dẫn chứng: Dòng log Generated signed binary image .../YOLO\_BIT.bin from .../YOLO\_BIT-unsigned.bin.
  - Giải thích: Kết quả cuối cùng là file YOLO\_BIT.bin. File này đã được đính kèm chữ ký số ECDSA. Đây là file duy nhất được phép nạp vào thiết bị để vượt qua quá trình kiểm tra của Bootloader.

#### 5. Burn khóa bảo mật Bootloader vào trong eFuse (Block 2), sau đó tiến hành kích hoạt secure boot version 1.

Bước này khá là quan trọng, đó là khắc vĩnh viễn khóa bảo mật Bootloader vào trong eFuse, mọi thao tác đều không thể thực hiện lại:

```
1 espefuse.py -p COM3 burn_key secure_boot_v1 build/bootloader/secure-bootloader-key-256.bin
```

Tiếp theo ta tiến hành bật kích hoạt secure boot version 1 thực sự bằng lệnh:

```
1 espefuse.py -p COM3 burn_efuse ABS_DONE_0
```

Để kiểm tra secure boot đã được kích hoạt chưa, ta thực hiện lệnh sau để xem toàn bộ cấu hình:

```
1 espefuse.py -p COM3 summary
```



Security fuses:		
UART_DOWNLOAD_DIS (BLOCK0)	Disable UART download mode. Valid for ESP32 V3 and = False R/W (0b0)	newer: only
ABS_DONE_0 (BLOCK0)	Secure boot V1 is enabled for bootloader image	= True R/W (0b1)
ABS_DONE_1 (BLOCK0)	Secure boot V2 is enabled for bootloader image	= False R/W (0b0)
DISABLE_DL_ENCRYPT (BLOCK0)	Disable flash encryption in UART bootloader	= False R/W (0b0)
DISABLE_DL_DECRYPT (BLOCK0)	Disable flash decryption in UART bootloader	= False R/W (0b0)
KEY_STATUS (BLOCK0)	Usage of efuse block 3 (reserved)	= False R/W (0b0)
SECURE_VERSION (BLOCK3)	Secure version for anti-rollback	= 0 R/W (0x00000000)
BLOCK1 (BLOCK1)	Flash encryption key	
= 00 R/W		
BLOCK2 (BLOCK2)	Security boot key	
= ?? -/-		
BLOCK3 (BLOCK3)	Variable Block 3	
= 00 R/W		

Figure 9: Bảng tóm tắt cấu hình secure boot v1

- Kích hoạt thành công Secure Boot V1 (ABS\_DONE\_0):
  - Quan sát: Dòng ABS\_DONE\_0 (BLOCK0) hiển thị trạng thái = True R/W (0b1).
  - Ý nghĩa: Đây là bảng chứng xác nhận tính năng Secure Boot V1 đã được kích hoạt vĩnh viễn trên phần cứng. Bit này hoạt động như một "cầu chì" điện tử—một khi đã đốt (chuyển từ 0 sang 1), nó không thể khôi phục lại. Kể từ lần khởi động tiếp theo, ROM của ESP32 sẽ bắt buộc kiểm tra Bootloader Digest tại địa chỉ 0x0 trước khi cho phép hệ thống chạy.
- Cơ chế bảo vệ Khóa bảo mật (BLOCK2):
  - Quan sát: Tại dòng BLOCK2 (Security boot key), giá trị hiển thị là một chuỗi các ký tự ?? ?? ?? ... thay vì chuỗi Hex của khóa mà ta đã nạp.
  - Đây không phải là lỗi, mà là tính năng bảo mật cốt lõi gọi là Read Protection (Chống đọc ngược). Khi khóa Secure Boot (AES-256) được nạp vào Block 2, hệ thống đã đồng thời kích hoạt bit cấm đọc (Read Disable) đối với vùng nhớ này.
  - Mục đích: Ngăn chặn tin tặc hoặc các công cụ debug đọc trộm khóa bí mật từ phần cứng. Nếu khóa này hiển thị rõ ràng (plaintext), kẻ tấn công có thể sao chép nó để tạo ra các thiết bị giả mạo hoặc vượt qua cơ chế bảo mật.

## 6. Upload Bootloader vào bộ nhớ flash tại địa chỉ 0x1000

Dể tiến hành upload Bootloader vào bộ nhớ flash ta thực hiện câu lệnh sau:

```
1 esptool.exe -p COM3 -b 460800 --before default_reset --after no_reset
  --chip esp32 write_flash --flash_mode dio --flash_size 4MB --
  flash_freq 40m 0x1000 build\bootloader\bootloader.bin
```

Ghi file nhị phân bootloader.bin vào bộ nhớ Flash ngoài (SPI Flash) tại địa chỉ khởi đầu là 0x1000, file bootloader.bin vừa được nạp chứa hai thành phần quan trọng:

- Logic xác thực: Nó chứa các đoạn mã để kiểm tra chữ ký số của Firmware ứng dụng (nằm tại 0x20000) trước khi cho phép ứng dụng chạy.
- Khóa công khai (Public Key): Nó chứa signature\_verification\_key.bin (đã được nhúng khi build) để đối chiếu với chữ ký của Firmware

## 7. Upload Bootloader Digest vào bộ nhớ Flash tại địa chỉ 0x0

Tiếp theo ta tiến hành Upload Digest vào bộ nhớ Flash

```
1 esptool.exe -p COM3 -b 460800 --before default_reset --after no_reset
  --chip esp32 write_flash --flash_mode dio --flash_size 4MB --
  flash_freq 40m 0x0 build\bootloader\bootloader-reflash-digest.bin
```

Sau khi upload Digest vào, ta có thể kiểm tra nội dung bộ nhớ flash tại địa chỉ 0x0:

```
1 esptool.exe -p COM3 read_flash 0x0 0x1000 digest_secure_bootloader.
  bin
```



Decoded Text	Data Inspector
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
B6 AC 79 3A 46 86 81 30 CC 11 0C 61 B7 34 D4 95	. . y : F . . 0 . . a . 4 . binary 10100010
92 69 29 BD 08 DF 5F ED 7E C3 86 EF 8B 9B 37 5C	. i ) . . _ ~ . . . . 7 \ octal 242
31 7E 98 26 36 C5 3C D2 ED 00 B6 2A D3 FF 66 05	1 ~ . & 6 . < . . * . f . uint8 162
20 AB 76 3A 71 B0 F2 46 9D C1 F9 96 0A 80 B5 CF	. v : q . . F . . . . . int8 -94
FE 65 B5 CB F5 19 05 38 8A 71 63 CC 21 53 A7 64	. e . . . 8 . q c . ! s . d . uint16 50850
69 D6 9B CE 67 04 9E 2D 6B E7 10 6F F5 CB 0A E0	i . . g . . - k . o . . int16 -14686
BB DE C3 53 09 0B D7 DB 4D 51 57 81 68 07 FA 53	. . . s . . . M Q W . h . s . uint24 16762530
1B B3 1F 31 03 E0 22 59 A1 9E 90 99 A2 19 05 3E	. . 1 . " Y . . . . > . int24 -14686
21 E1 7C E7 DB 50 3A 3E 69 BB A9 97 02 FF 56 F9	! .   . P : > i . . . . V . uint32 4294952610
85 FC C5 01 39 DD 92 7F 89 C2 25 88 FE E9 2B 3B	. . . 9 . . . % . . + ; . int32 -14686
6A F6 74 31 32 04 B0 DC 7C 9D 7F 68 39 CF EA 6E	j . t 1 2 . .   . h 9 . . n . uint64 18446744073709536930
89 05 DF E7 8C 51 32 27 18 4E FB B6 32 41 A2 C6	. . . Q 2 . . N . . 2 A . . int64 -14686
FF	ULEB128
FF	SLEB128
FF	float16 -6.6328125
FF	bfloat16 -20736
FF	float32 NaN

Figure 10: Quan sát dữ liệu trong bộ nhớ flash tại địa chỉ 0x0

Nội dung file sẽ chứa hai thành phần:

- Vùng đệm ngẫu nhiên (Offset 0x00 - 0x7F), có các chức năng:
  - Vector khởi tạo: Trong phần cứng bảo mật của ESP32, khối AES và SHA hoạt động kết hợp. Để tính toán ra chuỗi Digest 512-bit (ở vùng màu đỏ phía dưới), phần cứng không chỉ đơn thuần băm dữ liệu Bootloader thô. Trước khi bắt đầu tính toán, 128 bytes ngẫu nhiên này được nạp vào phần cứng để làm trạng thái khởi đầu.
  - Đảm bảo tính duy nhất (Uniqueness) và chống phân tích mẫu: Nếu không có vùng ngẫu nhiên này, với cùng một file bootloader.bin và cùng một Secure Boot Key, hệ thống sẽ luôn tạo ra cùng một chuỗi Digest giống hệt nhau. Điều này tạo ra một "mẫu tĩnh" (static pattern). Kẻ tấn công có thể so sánh digest của nhiều thiết bị khác nhau để suy đoán về phiên bản bootloader.
  - Nhờ 128 bytes ngẫu nhiên này, mỗi lần ta chạy lệnh tạo digest (dù nội dung Bootloader không đổi), công cụ espsecure.py sẽ sinh ra một chuỗi ngẫu nhiên mới. Điều này dẫn đến chuỗi SHA-512 Digest (ở offset 0x80) cũng sẽ thay đổi theo, dẫn đến bản ghi digest tại 0x0 là độc nhất. Kẻ tấn công nhìn vào bộ nhớ Flash sẽ thấy dữ liệu hoàn toàn khác biệt ở mỗi lần nạp, làm vô hiệu hóa các phương pháp tấn công dựa trên việc so sánh mẫu dữ liệu (Pattern Analysis).
- Chuỗi tóm tắt SHA-512 (Offset 0x80 - 0xBF - Vùng khoanh đỏ), được dùng để xác thực Bootloader, cụ thể khi khởi động, ROM của ESP32 sẽ:
  - Đọc Bootloader thật đang nằm tại địa chỉ 0x1000.
  - Mã hóa Bootloader đó bằng khóa AES-256 trong eFuse (Block 2).
  - Tính toán lại mã băm SHA-512 của dữ liệu vừa giải mã.
  - So sánh kết quả tính toán với chuỗi 512-bit đang nằm trong vùng khoanh đỏ này (0x80 - 0xBF).
  - Nếu khớp hoàn toàn, Bootloader được coi là toàn vẹn và được phép chạy.

#### 8. Tiến hành thử nghiệm chạy firmware đã kí

Ta tiến hành nạp firmware đã kí sẵn vào bộ nhớ flash tại địa chỉ 0x20000 bằng câu lệnh:

```
1 esptool.exe -p COM3 -b 460800 --before default_reset --after no_reset
  --chip esp32 write_flash --flash_mode dio --flash_size 4MB --
  flash_freq 40m 0x20000 .\build\YOLO_BIT.bin
```



Figure 11: Kết quả chạy firmware đã kí

- Xác thực Chữ ký số (Signature Verification):

- Trước vùng khoanh đỏ đầu tiên, log hiển thị dòng boot: Verifying image signature.... Ngay sau đó là boot: Loaded app from partition at offset 0x20000.
- Ý nghĩa: Bootloader đã sử dụng khóa công khai (Public Key) nhúng trong nó để giải mã và



kiểm tra chữ ký ECDSA của file YOLO\_BIT.bin. Kết quả khớp nhau, chứng tỏ firmware này toàn vẹn và xuất phát từ nguồn tin cậy (người nắm giữ Private Key).

- Trạng thái Secure Boot:
  - Dòng log: secure\_boot\_v1: bootloader secure boot is already enabled. No need to generate digest. continuing..
  - Ý nghĩa: Bootloader nhận diện được bit ABS\_DONE\_0 trong eFuse đã được bật (=1). Nó xác nhận hệ thống đang chạy trong chế độ bảo mật phần cứng và bỏ qua bước sinh digest (vì digest đã được nạp sẵn ở địa chỉ 0x0). Đây là dấu hiệu cho thấy cấu hình phần cứng đã chính xác.
- Chuyển quyền điều khiển:
  - Hệ thống tiếp tục khởi tạo RAM, CPU và các ngoại vi (heap\_init, cpu\_start, spi\_flash). Điều này chỉ xảy ra khi bước xác thực chữ ký ở trên thành công tuyệt đối.
  - Ứng dụng hoạt động bình thường (Vùng khoanh đỏ 2 & 3), SECURE APP: System ready for Secure Boot! (Đây là log do mã nguồn ứng dụng in ra). Các dòng Wifi init và các dấu chấm "." liên tục xuất hiện.
  - Ý nghĩa: CPU đã nhảy vào địa chỉ 0x20000 và thực thi mã lệnh của ứng dụng người dùng. Việc ứng dụng chạy ổn định và in log chứng tỏ quá trình Secure Boot "trong suốt" với ứng dụng, không gây ảnh hưởng đến hiệu năng hay chức năng của phần mềm.

#### 9. Tiến hành thử nghiệm chạy firmware chưa kí

```
1 esptool.exe -p COM3 -b 460800 --before default_reset --after no_reset --
  chip esp32 write_flash --flash_mode dio --flash_size 4MB --
  flash_freq 40m 0x0 .\build\bootloader\bootloader-reflash-digest.bin 0
  x10000 .\build\partition_table\partition-table.bin 0x20000 .\build\
  YOLO_BIT-unsigned.bin --force
```



```
File Edit Selection View Go Run Terminal Help ← → 🔍 YOLO_BIT
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS ESP-IDF SERIAL MONITOR
S SERIAL MONITOR
+ Open an additional monitor
Monitor Mode Serial View Mode Text Port COM3 - USB-SERIAL CH340 (COM3) Baud rate 115200 Line ending None Start Monitoring
mode:DIO, clock div:2
load:0x3ff00b00, len:14600
load:0x00078000, len:26764
ho 0 tail 12 room 4
load:0x00080400, len:4
load:0x00080404, len:4256
entry 0x40000688
[...]
I (58) boot: compile time: Nov 29 2025 15:13:19
I (53) boot: Multicore bootloader
D (67) bootloader_flash: non-XXC chip detected by SFDP Read (00), skip.
D (73) bootloader_flash: mmu set block paddr=0x00000000 (was 0xffffffff)
I (80) boot: chip revision: v1.1
D (84) boot.esp32: magic eh
D (97) boot.esp32: spi mode 04
D (98) boot.esp32: spi mode 02
D (93) boot.esp32: spi speed 00
D (96) boot.esp32: spi size 02
I (99) boot.esp32: SPI Speed : 40MHz
I (104) boot.esp32: SPI Mode : DIO
I (109) boot.esp32: SPI Flash Size : 4MB
D (116) boot: Enabling RTCMDS (0x0000)
I (121) boot: RTCMDS entry source
D (123) bootloader_flash: codata starts from paddr=0x00010000, size=0xC00, will be mapped to vaddr=0x3f400000
V (123) bootloader_flash: after mapping, starting from paddr=0x00010000 and vaddr=0x3f400000, 0x1000 bytes are mapped
D (143) boot: mapped partition table at 0x3f400000
D (149) flash_parts: partition table verified, 4 entries
I (154) boot: Partition Table:
I (158) boot: #1 label: Usage type ST Offset Length
D (160) boot: #2 label: Usage type ST Offset Length
D (170) boot: #3 label: WiFi data type ST Offset Length
T (173) boot: #4 mvs type subtype=2
D (181) boot: load partition table entry 0x3f400000
D (186) boot: type=1 subtype=1
I (189) boot: 1 phy_init RF data 01 01 00017000 00006000
D (197) boot: load partition table entry 0x3f400000
D (202) boot: type=0 subtype=0
I (210) boot: factory app factory app 00 00 00020000 00100000
D (217) boot: Trying partition index -1 offs 0x20000 size 0x10000
D (223) esp_image: reading image header @ 0x20000
D (228) bootloader_flash: mmu set block paddr=0x00020000 (was 0xffffffff)
D (234) esp_image: image header: 0xe9 0x06 0x02 0x02 4008164
V (240) esp_image: loading segment header @ at offset 0x20018
V (246) esp_image: segment data length 0x2098 data starts 0x20020 load addr 0x3f400000
V (251) esp_image: segment 0 map segment 0 segment data offs 0x20020 size 0x0908h (139416) map
D (269) esp_image: free data page count 0x00000032
D (274) bootloader_flash: codata starts from paddr=0x00020000, size=0x2098, will be mapped to vaddr=0x3f400000
V (280) bootloader_flash: after mapping, starting from paddr=0x00020000 and vaddr=0x3f400000, 0x3000 bytes are mapped
V (346) esp_image: loading segment header 1 @ offset 0x20028
D (346) bootloader_flash: mmu set block paddr=0x0000400000 (was 0xffffffff)
esp_image: segment data length 0x4204 data starts 0x420c0
V (352) esp_image: segment map segment 0 segment data offs 0x420c0 load addr 0x3fb00000
T (354) esp_image: secure boot signature verification failed
D (382) esp_image: free data page count 0x00000032
D (387) bootloader_flash: codata starts from paddr=0x000f86cc, size=0x8b4, will be mapped to vaddr=0x3f400000
V (387) bootloader_flash: after mapping, starting from paddr=0x000f0000 and vaddr=0x3f400000, 0x1000 bytes are mapped
V (389) esp_image: image start 0x000020000 end of last section 0x000ff000
D (390) bootloader_flash: codata starts from paddr=0x000ffff0, size=0x4, will be mapped to vaddr=0x3f400000
V (392) bootloader_flash: after mapping, starting from paddr=0x000f0000 and vaddr=0x3f400000, 0x1000 bytes are mapped
E (939) secure_boot: image has invalid signature version field 0
xxffffffff (image without a signature?)
```

Type a message to send to the serial port

Figure 12: Kết quả chạy firmware không được kí

• Vùng khoanh đỏ 1: Phát hiện thiếu chữ ký

```
1 E (939) secure_boot: image has invalid signature version field 0
xxffffffff (image without a signature?)
```

- Bootloader tìm kiếm Signature Block (khối dữ liệu chữ ký) tại cuối firmware hoặc tại vị trí offset quy định.
- Giá trị 0xffffffff cho thấy Bootloader chỉ đọc được vùng flash trống (mặc định giá trị 0xFF).
- Điều này đồng nghĩa firmware không có chữ ký hợp lệ → “image without a signature”.
- Kết quả:

```
1 E (940) esp_image: Secure boot signature verification failed
```

• Vùng khoanh đỏ 2: Từ chối khởi động (Boot Rejection)

```
1 W (1225) esp_image: image valid, signature bad
```



- Bootloader xác nhận firmware có cấu trúc đúng (header hợp lệ, checksum hợp lệ) → image valid.
- Tuy nhiên chữ ký đi kèm lại sai hoặc thiếu → signature bad.
- Hành động của Bootloader:

```
1 E (1229) boot: Factory app partition is not bootable
2 E (1236) boot: No bootable app partitions in the partition table
```

- Vì không còn phân vùng OTA nào khác chứa firmware hợp lệ, hệ thống không thể tiếp tục boot.

- Vùng khoanh đỏ 3: Vòng lặp khởi động lại (Boot Loop)

```
1 rst:0x3 (SW_RESET),boot:0x13...
```

- Không tìm thấy ứng dụng đáng tin cậy để chuyển quyền điều khiển.
- Hệ thống thực hiện Software Reset để thử lại chu trình boot.
- Việc này dẫn đến boot loop, khiến thiết bị không thể sử dụng (soft brick).

#### 10. Tiến hành thử nghiệm chạy bootloader "lạ"

Thử nghiệm này nhằm kiểm chứng khả năng bảo vệ của Root of Trust (ROM) trước các cuộc tấn công vật lý hoặc nạp chồng (Over-the-Air). Cụ thể, ta giả định kịch bản kẻ tấn công đã tiếp cận được thiết bị và cố gắng ghi đè một Bootloader độc hại (bootloader\_attack.bin) vào địa chỉ 0x1000 để chiếm quyền điều khiển hệ thống ngay từ giai đoạn khởi động.

```
1 esptool.exe -p COM3 -b 460800 --before default_reset --after no_reset
    --chip esp32 write_flash --flash_mode dio --flash_size 4MB --
    flash_freq 40m 0x1000 .\bootloader_attack.bin --force
```

```
rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:1184
load:0x40078000,len:13232
load:0x40080400,len:3028
secure boot check fail
ets_main.c 371
ets Jun  8 2016 00:22:57
```

Figure 13: Kết quả chạy bootloader "lạ"



Dòng Log	Giải thích ý nghĩa kỹ thuật
rst:0x10 (RTCWDT_RTC_RESET)	<b>Nguyên nhân reset:</b> Chip bị reset bởi bộ đếm thời gian (Watchdog Timer). Do kiểm tra bảo mật thất bại, hệ thống bị treo hoặc rơi vào vòng lặp, khiến Watchdog kích hoạt để khởi động lại chip liên tục (Boot loop).
boot:0x13 (SPI_FAST_FLASH_BOOT)	<b>Chế độ khởi động:</b> Chip đang khởi động từ bộ nhớ Flash ngoài qua giao tiếp SPI (Fast Flash Boot). Điều này xác nhận Bootloader giả mạo đang nằm trên Flash.
configsip: 0, SPIWP:0xee mode:DIO, clock div:2	<b>Cấu hình phần cứng:</b> ROM đã thiết lập thành công giao tiếp SPI với chip Flash (chế độ DIO) để chuẩn bị đọc dữ liệu.
load:0x3fff0030,len:1184 load:0x40078000,len:13232 load:0x40080400,len:3028	<b>Quá trình nạp (Load):</b> ROM đang đọc các đoạn mã (segments) của Bootloader từ Flash vào RAM. Lưu ý: ROM mới chỉ đọc dữ liệu để kiểm tra, chưa cho phép <i>thực thi</i> .
secure boot check fail	<b>LỖI CHÍNH (Critical Error):</b> Cơ chế Secure Boot đã tính toán Digest của dữ liệu vừa nạp và so sánh với Digest chuẩn tại địa chỉ 0x0. Kết quả <b>KHÔNG KHỐP</b> , dẫn đến việc xác thực thất bại.
ets_main.c 371	<b>Nguồn gốc lỗi:</b> Lỗi được báo từ dòng 371 của file <code>ets_main.c</code> . Đây là mã nguồn của <b>ROM cứng</b> bên trong chip (Hardware ROM), khẳng định lớp bảo vệ phần cứng đã chặn cuộc tấn công.

Table 6: Phân tích chi tiết Log báo lỗi Secure Boot Check Fail

Như vậy, qua các bước thực nghiệm trên, chúng ta đã thiết lập thành công cơ chế Secure Boot, đảm bảo tuyệt đối tính toàn vẹn (Integrity) (firmware không bị sửa đổi) và Tính xác thực (Authenticity) (firmware do đúng chủ sở hữu ký); ngăn chặn triệt để mọi nỗ lực nạp Bootloader giả mạo hay Firmware không được ủy quyền.

Tuy nhiên, dù mã đọc không thể khởi chạy, nội dung Firmware và dữ liệu nhạy cảm lưu trên Flash vẫn đang tồn tại dưới dạng văn bản rõ (Plaintext). Nếu kẻ tấn công tháo chip Flash để đọc trực tiếp (Dump Flash), họ vẫn có thể sao chép công nghệ hoặc trích xuất thông tin người dùng. Để chứng minh điều này ta tiến hành dump dữ liệu flash ra, sau đó sử dụng lệnh strings kết hợp với sls (Select-String) để tìm kiếm các chuỗi ký tự có thể đọc được (ASCII) bên trong file nhị phân vừa trích xuất. phân tích:



```
PS D:\Study\Cryptoloraphy\YOLO_BIT> esptool.exe read_flash 0x20000 0x100000 flash_contents.bin
esptool.py v4.10.0
Found 1 serial ports
Serial port COM3
Connecting.....
Detecting chip type... ESP32
Chip is ESP32-D0WD (revision v1.1)
Features: WiFi, BT, Dual Core, 240MHz, VRef calibration in efuse, Coding Scheme None
Crystal is 40MHz
MAC: 08:f9:e0:ac:41:d4
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
1048576 (100 %)
1048576 (100 %)
Read 1048576 bytes at 0x000020000 in 95.6 seconds (87.8 kbit/s)...
Hard resetting via RTS pin...
PS D:\Study\Cryptoloraphy\YOLO_BIT> strings .\flash_contents.bin | sls "OPPO Reno4" -Context 6,6

?SECURE_APP
[0;32mI (%lu) %s: MQTT message on topic: %
[0;31mE (%lu) %s: Failed to allocate memory for payload
[0;32mI (%lu) %s: Payload: %
[0;32mI (%lu) %s: Connecting to WiFi...
hcmutk22
> OPPO Reno4
[0;32mI (%lu) %s: .
[0;32mI (%lu) %s: WiFi connected!
[0;32mI (%lu) %s: IP address: %
app.coreiot.io
[0;32mI (%lu) %s: Connecting to MQTT...
123456
```

Figure 14: Dánh giá hiện trạng bảo mật khi chưa mã hóa

Kết quả từ màn hình console cho thấy toàn bộ dữ liệu hằng số (Hardcoded data) trong Firmware đều hiển thị rõ ràng dưới dạng Plaintext. Cụ thể, các thông tin nhạy cảm sau đã bị lộ:

- Thông tin kết nối mạng:
  - SSID Wifi: "OPPO Reno4" (Mục tiêu tìm kiếm).
  - Mật khẩu Wifi (Dự đoán): "hcmutk22" (xuất hiện ngay bên cạnh SSID).
- Thông tin hạ tầng Server:
  - Địa chỉ máy chủ: "app.coreiot.io".
  - Giao thức: "Connecting to MQTT...".
  - MQTT Password: '123456'

Để giải quyết mảng ghép cuối cùng của bài toán bảo mật là Tính bí mật (Confidentiality), nhóm sẽ tiến hành triển khai lớp bảo vệ thứ hai: Mã hóa bộ nhớ Flash (Flash Encryption).

## 4.2 Hiện thực mã hóa bộ nhớ flash

### 1. Cấu hình mã hóa bộ nhớ flash

Dầu tiên ta sẽ tiến hành cấu hình mã hóa bộ nhớ flash trong sdkconfig, quy trình kích hoạt và vận hành mã hóa trên thiết bị diễn ra qua hai giai đoạn riêng biệt, được điều khiển bởi các tham số sau:

- Giai đoạn 1: Nạp lần đầu (First Boot - Auto Encryption)
  - Giai đoạn này người dùng nạp firmware dưới dạng Plaintext (chưa mã hóa). Sau khi khởi động lại, Bootloader tự động mã hóa toàn bộ nội dung Flash và cập nhật thanh ghi flash\_crypt\_cnt từ trạng thái tắt (số bit 1 chẵn) lên trạng thái bật (số bit 1 lẻ).
  - Cơ chế kỹ thuật: Ở lần khởi động đầu tiên, Bootloader phát hiện tính năng mã hóa đã được bật nhưng dữ liệu trong Flash vẫn là Plaintext. Nó sử dụng khóa AES-256 (đã nạp trong Block 1, sẽ được nói đến ở bước sau) để thực hiện mã hóa tại chỗ.
  - Cấu hình sdkconfig chi phối:
    - \* CONFIG\_SECURE\_FLASH\_ENC\_ENABLED=y: Tham số tiên quyết để kích hoạt bộ máy mã hóa/giải mã AES phần cứng. Nếu thiếu dòng này, chip sẽ chạy firmware plaintext như bình thường mà không thực hiện mã hóa.
    - \* CONFIG\_SECURE\_FLASH\_UART\_BOOTLOADER\_ALLOW\_ENC=y: Đây là cấu hình cho phép Bootloader thực hiện việc mã hóa dữ liệu được nạp qua cổng UART. Nhờ cấu hình này, ở lần đầu tiên, ta không cần mã hóa thủ công; Bootloader sẽ "làm hộ" việc đó và ghi lại dữ liệu đã mã hóa vào Flash.
    - \* CONFIG\_SECURE\_FLASH\_ENCRYPTION\_MODE\_DEVELOPMENT=y: Thiết lập chế độ Phát triển. Chế độ này cho phép ta thay đổi trạng thái eFuse flash\_crypt\_cnt nhiều lần (bằng cách đốt thêm bit) thay vì khóa cứng ngay lập tức. Đây là lý do flash\_crypt\_cnt có thể tăng từ giá trị cũ lên giá trị mới (ví dụ: ...011 → ...111) để đánh dấu việc mã hóa hoàn tất.
- Giai đoạn 2: Các lần cập nhật sau (Manual Encryption)
  - Để cập nhật firmware mới, người dùng phải mã hóa thủ công file firmware trên máy tính (tạo file ciphertext) trước khi nạp xuống thiết bị.
  - Cơ chế kỹ thuật: Do thanh ghi flash\_crypt\_cnt có giới hạn số bit (7 bit). Nếu tiếp tục nạp Plaintext và để Bootloader tự động mã hóa như Giai đoạn 1, thanh ghi này sẽ bị đốt hết bit (1111111) và thiết bị sẽ không thể cập nhật được nữa (hoặc bị khóa vĩnh viễn tùy cấu hình). Việc nạp file đã mã hóa sẵn giúp Bootloader bỏ qua bước "tự động mã hóa", do đó không đốt thêm bit nào vào flash\_crypt\_cnt, giữ nguyên trạng thái.
  - Cấu hình sdkconfig chi phối:
    - \* CONFIG\_SECURE\_FLASH\_ENCRYPTION\_MODE\_DEVELOPMENT=y: Mặc dù là chế độ phát triển, nhưng nó vẫn tuân theo quy tắc vật lý của eFuse. Việc chọn chế độ này cho phép chúng ta tiếp tục sử dụng thiết bị với flash\_crypt\_cnt dừng lại ở trạng thái đã bật ở giai đoạn 1 (nếu nạp thủ công), thay vì bị khóa cứng hoàn toàn như chế độ Release.
    - \* CONFIG\_SECURE\_FLASH\_ENC\_ENABLED=y: Vì phần cứng giải mã luôn bật, mọi dữ liệu nạp vào Flash bắt buộc phải là định dạng Ciphertext để khi CPU đọc ra (qua bộ giải mã AES), nó sẽ trở thành mã lệnh Plaintext hợp lệ để thực thi. Nếu nạp Plaintext vào lúc này (mà không qua cơ chế tự mã hóa), bộ giải mã sẽ biến nó thành dữ liệu rác và gây treo chip.

### 2. Tạo khóa mã hóa bộ nhớ flash

Ta tiến hành tạo khóa cho mã hóa flash bằng lệnh sau:

```
1 espsecure.exe generate_flash_encryption_key my_flash_encrypt_key.  
bin
```

Đây là khóa đối xứng (Symmetric Key). Nghĩa là cùng một file khóa này sẽ được dùng cho cả hai chiều:

- Mã hóa: Dùng để mã hóa firmware trên máy tính (hoặc do Bootloader tự mã hóa lần đầu tiên).

- Giải mã: Dùng nạp vào phần cứng để module AES Accelerator giải mã khi chạy thực tế.
3. Burn key mã hóa bộ nhớ flash vào eFuse (Block 1)  
Sau khi tạo khóa, ta tiến hành khắc khóa vĩnh viễn trong Block 1 để phục vụ cho việc mã hóa và giải mã bộ nhớ flash:

```
1 espefuse.py --port COM3 burn_key BLOCK1 my_flash_encrypt_key.bin
```

Trước khi tiến hành nạp chương trình lần đầu tiên, bạn em có tiến hành "Burn" vĩnh viễn các cấu hình sau:

- DISABLE\_DL\_DECRYPT (= True): Vô hiệu hóa tính năng giải mã qua cổng nạp (UART Download Mode). Đây là chốt chặn quan trọng nhất: nó đảm bảo khi ai đó cố tình dump dữ liệu từ Flash qua cổng USB, họ chỉ nhận được dữ liệu đã mã hóa (ciphertext/rác) thay vì dữ liệu gốc (plaintext).
- DISABLE\_DL\_CACHE (= True): Tắt bộ nhớ đệm (Cache) khi ở chế độ nạp. Khi Flash Encryption bật, dữ liệu trong Flash là mã hóa (Ciphertext). Khi CPU cần dùng, phần cứng giải mã và đưa dữ liệu "sạch" (Plaintext) vào Cache để CPU xử lý cho nhanh. Khi bật chế độ này sẽ ngăn chặn các cuộc tấn công kẽm bên (side-channel attacks), mọi truy xuất dữ liệu phải đi qua quy trình đọc Flash gốc, kẽm công không thể đo đếm thời gian phản hồi (vì mọi phản hồi đều chậm như nhau) và cũng không thể "trích xuất" những mảnh dữ liệu plaintext còn sót lại trong Cache từ phiên làm việc trước.

4. Tiến hành build chương trình lần đầu tiên sau khi cấu hình flash

```
1 idf.py build
```

Khi build chương trình lần đầu tiên, thiết bị chuyển từ trạng thái "không bảo mật" (Plaintext) sang "đã bảo mật" (Encrypted) lần đầu tiên. Sau khi phân tích log được tạo ra, ta có thể chia nội dung phân tích trong báo cáo thành 4 giai đoạn diễn ra trong log:

- Giai đoạn Phát hiện và Mã hóa (Encryption Process)

```
D (1587) flash_encrypt: bootloader is plaintext. Encrypting...
D (2497) flash_encrypt: Encrypting secure bootloader IV & digest...
I (2567) flash_encrypt: bootloader encrypted successfully
D (2567) flash_parts: partition table verified, 4 entries
D (2567) flash_encrypt: partition table is plaintext. Encrypting...
I (2645) flash_encrypt: partition table encrypted and loaded successfully
```

Figure 15: Tự động mã hóa Bootloader, Partition Table

```
I (3696) flash_encrypt: Encrypting partition 2 at offset 0x20000 (length 0x10000)...
I (21008) flash_encrypt: Done encrypting
D (21008) flash_encrypt: All flash regions checked for encryption pass
```

Figure 16: Tự động mã hóa firmware

- Bootloader kiểm tra các vùng nhớ quan trọng (Bootloader, Partition Table, App Partition).
- Nó phát hiện dữ liệu đang ở dạng Plaintext (chưa mã hóa).
- Vì đây là lần chạy đầu tiên, Bootloader tự động kích hoạt quy trình mã hóa tại chỗ (In-place encryption).
- Thời gian từ I (3696) đến I (21008), đây là thời gian chip thực hiện mã hóa vùng firmware (offset 0x20000) bằng phần cứng.
- Giai đoạn Cập nhật eFuse (Burning eFuse)

```
I (21034) efuse: BURN_BLOCK0
I (21048) efuse: BURN_BLOCK0 - OK (all write block bits are set)
I (21048) flash_encrypt: Flash encryption completed
```

Figure 17: Tự động cập nhật eFuse - Block0

- Sau khi mã hóa xong dữ liệu, hệ thống tiến hành cập nhật trạng thái bảo mật vào phần cứng.
  - BURN BLOCK0: Ta thấy log flash\_encrypt: CRYPT\_CNT 2 -> 1 tức là phản ánh số lượng chu kỳ nạp firmware Plaintext còn lại (Remaining Flashes) trong chế độ Development (giá trị nhị phân của flash\_encrypt\_cnt trước khi build là 0b0000011 - đang tắt mã hóa, sau khi build chuyển sang 0b0000111 - bật mã hóa) bằng cách tự thực hiện ghi (đốt) bit vào thanh ghi flash\_crypt\_cnt trong eFuse Block 0.
  - Hành động này đánh dấu rằng: "Dữ liệu trong Flash bây giờ đã là Ciphertext".
- Giai đoạn Tự động Reset (Software Reset)

```
I (21064) boot: Resetting with flash encryption enabled...
ets Jun 8 2016 00:22:57

rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
```

Figure 18: Tự động reset sau khi build lần đầu tiên

- Ngay sau khi mã hóa và đốt eFuse xong, thiết bị bắt buộc phải khởi động lại.
  - Lý do: Để kích hoạt bộ giải mã phần cứng AES (AES Accelerator). Trước khi reset, CPU đọc Flash trực tiếp. Sau khi reset, mọi truy xuất từ CPU ra Flash sẽ đi qua bộ giải mã.
- Giai đoạn Khởi động lại với Bảo mật (Secured Boot)

```
V (1330) flash_encrypt: CRYPT_CNT 3, write protection 0
I (1335) flash_encrypt: flash encryption is enabled (2 plaintext flashes left)
```

Figure 19: Khởi động lại kết hợp với secure boot

- Đây là lần khởi động thành công với chế độ bảo mật đã kích hoạt.
- Giá trị của flash\_encrypt\_cnt hiện tại là 0b0000111 - đang bật flash và không còn bảo vệ ghi nữa, tức là từ những lần flash sau ta được phép mã hóa thủ công.

Sau khi khởi tạo lại thành công, chương trình được giải mã qua bộ tăng tốc phần cứng AES và thực thi được:

```
I (2720) SECURE_APP: Connecting to WiFi...
W (2723) wifi:wifi osi_nvs_open fail ret=4353

W (2724) wifi_init: Failed to unregister Rx callbacks
E (2725) wifi_init: Failed to deinit Wi-Fi driver (0x3001)
E (2730) wifi_init: Failed to deinit Wi-Fi (0x3001)
[ 1192][E][WiFiGeneric.cpp:248] wifiLowLevelInit(): esp_wifi_init 4353
[ 1210][E][STA.cpp:298] begin(): STA enable failed!
I (3758) SECURE_APP: .
I (4758) SECURE_APP: .
I (5758) SECURE_APP: .
I (6758) SECURE_APP: .
I (7758) SECURE_APP: .
```

Figure 20: Chương trình được mã hóa tự động sau lần boot đầu tiên đã thực thi

## 5. Tiến hành build chương trình lần thứ hai - mã hóa thủ công

Để kiểm chứng khả năng cập nhật và thực thi firmware mới trên thiết bị đã bật Flash Encryption, nhóm thực hiện một thay đổi nhỏ trong mã nguồn (file main.cpp). Cụ thể, thay đổi ký tự trong

thông báo kết nối wifi từ dấu . thành dấu , trong đoạn ESP\_LOGI ( TAG , "."). Sự thay đổi ký tự này đóng vai trò là dấu hiệu nhận biết (flag) để xác nhận firmware mới đã được nạp đè thành công lên phân vùng cũ, chứng tỏ quy trình mã hóa thủ công trước khi nạp vào chip là chính xác.

- Vấn đề kỹ thuật: Sai khác về Endianness. Khi chuyển sang giai đoạn nạp firmware thủ công (để không làm tăng bộ đếm eFuse FLASH\_CRYPT\_CNT), nhóm gặp phải một thách thức kỹ thuật về định dạng khóa:
  - Phần cứng ESP32 (Hardware): Khi thực hiện mã hóa tự động, khối AES Accelerator đọc khóa từ eFuse theo định dạng Little Endian (byte thấp đứng trước).
  - Công cụ trên Host (espsecure.py): Khi thực hiện mã hóa thủ công trên máy tính, công cụ mặc định xử lý khóa theo định dạng Big Endian (hoặc đọc tuần tự xuôi chiều).
  - Hệ quả: Nếu sử dụng trực tiếp file khóa gốc (my\_flash\_encrypt\_key.bin) để mã hóa trên máy tính, dữ liệu đầu ra sẽ sai lệch hoàn toàn so với thuật toán giải mã trong chip (vốn đang dùng khóa đảo ngược byte). Kết quả là chip sẽ giải mã ra dữ liệu rác và không thể khởi động.
- Giải pháp: Đảo ngược khóa (Key Reversal), để đồng bộ hóa giữa công cụ phần mềm và phần cứng, nhóm thực hiện đảo ngược thứ tự byte của file khóa gốc trước khi sử dụng bằng cách thực hiện câu lệnh:

```
1 espsecure.exe encrypt_flash_data --keyfile .\my_flash_encrypt_key
   .bin.reversed --output .\build\YOLO_BIT.bin.encrypted --
   address 0x20000 .\build\YOLO_BIT.bin
```

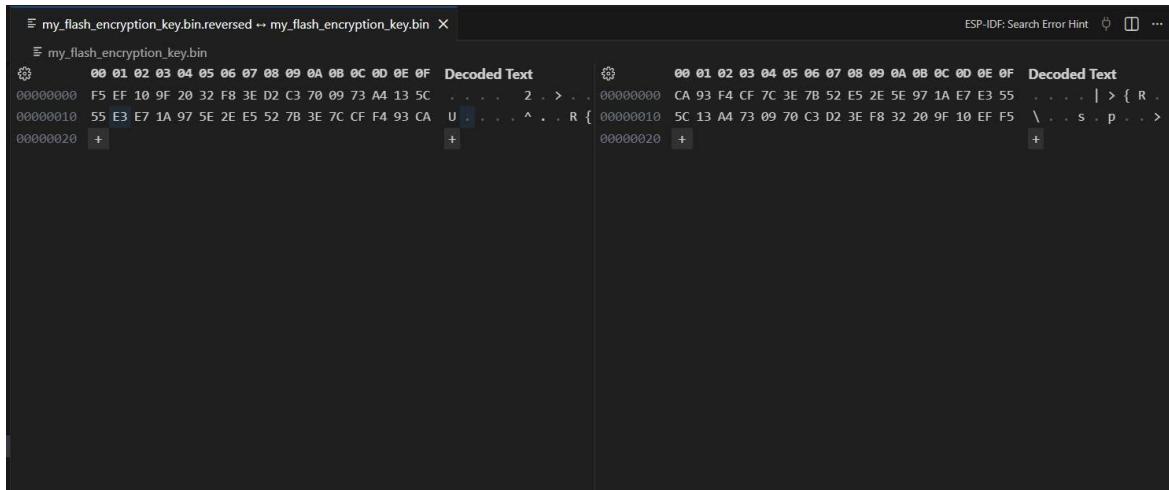


Figure 21: Đảo key để phục vụ mã hóa thủ công

- Quy trình Mã hóa Firmware thủ công (Pre-encryption)  
Sau khi có khóa tương thích, nhóm tiến hành mã hóa file firmware (YOLO\_BIT.bin) trên máy tính để tạo ra file Ciphertext (YOLO\_BIT.bin.encrypted).

```
1 espsecure.exe encrypt_flash_data --keyfile .\my_flash_encrypt_key
   .bin.reversed --output .\build\YOLO_BIT.bin.encrypted --
   address 0x20000 .\build\YOLO_BIT.bin
```

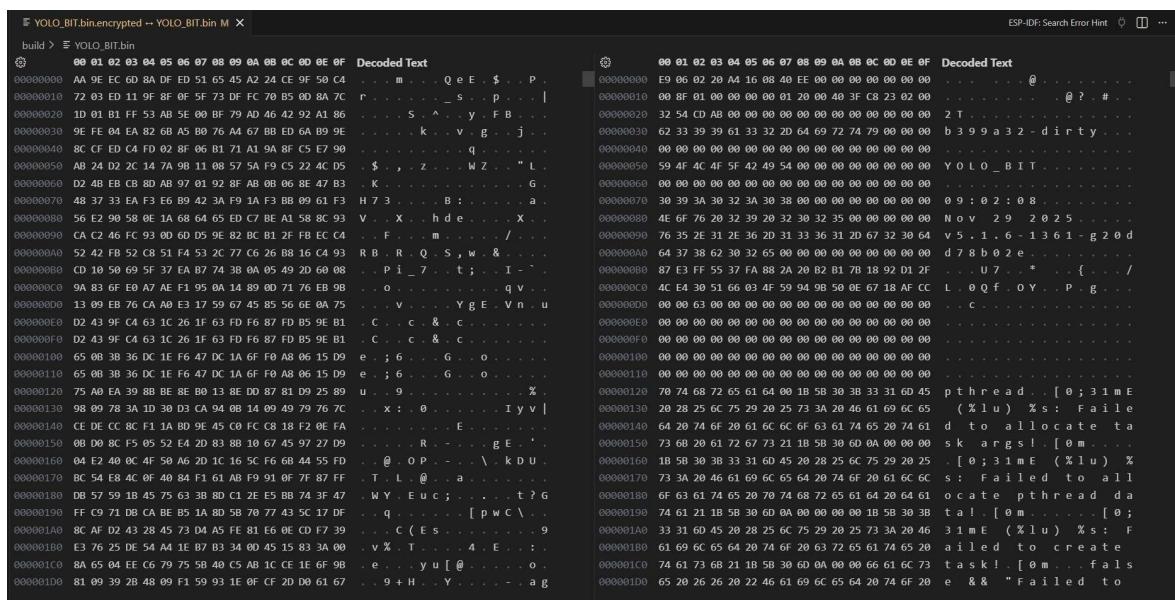
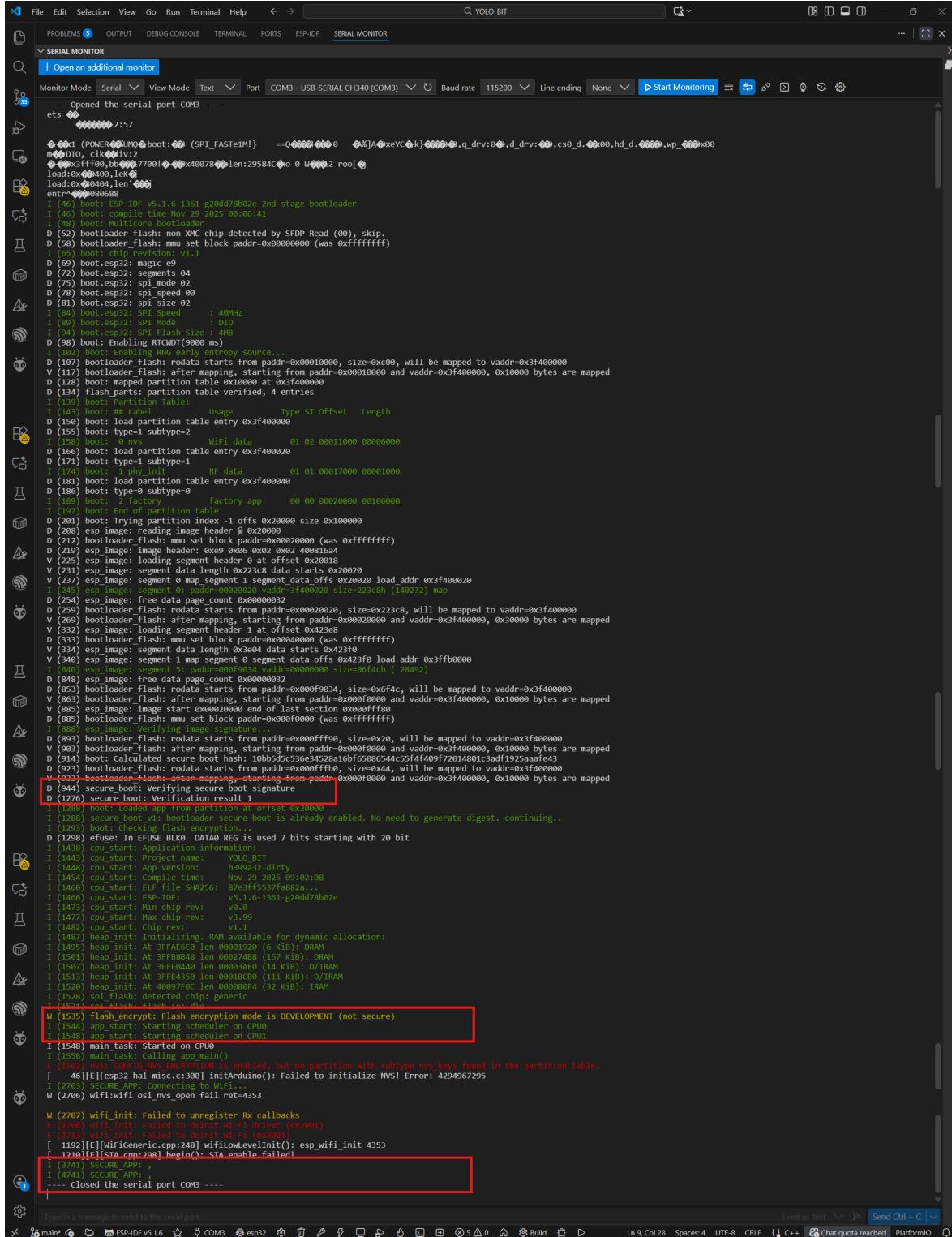


Figure 22: So sánh kết quả mã hóa thủ công và chưa mã hóa

Ta nhận thấy dữ liệu trong file mã hóa thủ công đã được mã hóa - không còn nhìn thấy các plaintext như file firmware chưa mã hóa.

- Tiếp theo ta tiến hành nạp firmware đã mã hóa thủ công:

```
1  esptool.exe -p COM3 -b 460800 --before default_reset --after
   no_reset --chip esp32 write_flash --flash_mode dio --
   flash_size 4MB --flash_freq 40m 0x200000 .\build\YOLO_BIT.bin.
   encrypted
```



The screenshot shows the PlatformIO IDE's serial monitor window during the boot process of an ESP32. The log output details the initial setup, including loading partitions from SPIFFS, configuring SPI modes, and attempting to enable RTCM32. It also shows the loading of the factory app and the start of the main application. A notable message is highlighted with a red box: 'W (1535) flash.encrypt: Flash encryption mode is DEVELOPMENT (not secure)'. This indicates that the user must manually enable encryption by running 'nvs\_flash\_encrypt()' or through the 'Secure APP' menu option before the device can be used.

Figure 23: Kết quả sau khi nạp firmware đã mã hóa thủ công

Log khởi động lần 2 chứng minh thành công của kỹ thuật Mã hóa thủ công (Pre-encryption) thông qua hai dấu hiệu:

- Không có hành động đốt eFuse: Bootloader không thực hiện các bước Encrypting partition... hay BURN BLOCK0, xác nhận rằng nó đã nhận diện đúng dữ liệu nạp vào là Ciphertext hợp lệ.
- Giải mã trong suốt thành công: Các thông báo Verification result 1 và app\_start... cho thấy phần cứng AES Accelerator đã giải mã chính xác dữ liệu từ Flash, cho phép Secure



Boot xác thực chữ ký và CPU thực thi ứng dụng bình thường.

6. Kiểm chứng khả năng bảo vệ tính bí mật (Confidentiality) của Flash Encryption bằng cách so sánh khả năng đọc hiểu dữ liệu giữa file gốc (ELF) và file trích xuất từ Flash (Dump).

```
PS D:\Study\Cryptology\YOLO_BIT> xtensa-esp32-elf-objdump -s -j .flash.rodata .\build\YOLO_BIT.elf | sls "OPPO Reno4" -Context 5,10
3f404750 75292025 733a2050 61796c6f 61643a20 u) %s: Payload:
3f404760 25731b5b 306d0a00 1b5b303b 33326d49 %s.[0m...[0;32mI
3f404770 2028256c 75292025 733a2043 6f6e6e65 (%lu) %s: Conne
3f404780 6374696e 6720746f 20576946 692e2e2e cting to WiFi...
3f404790 1b5b306d 0a000000 68636d75 746b3232 .[0m....hcmtk22
> 3f4047a0 00000000 4f50504f 2052656e 6f340000 ....OPPO Reno4...
3f4047b0 1b5b303b 33326d49 2028256c 75292025 .[0;32mI (%lu) %
3f4047c0 733a202c 1b5b306d 0a000000 1b5b303b s: ,,[0m.....[0;
3f4047d0 33326d49 2028256c 75292025 733a2057 32mI (%lu) %s: W
3f4047e0 69466920 636f6e6e 65637465 64211b5b iFi connected!-[

3f4047f0 306d0a00 1b5b303b 33326d49 2028256c 0m...[0;32mI (%l
3f404800 75292025 733a2049 50206164 64726573 u) %s: IP addres
3f404810 733a2025 731b5b30 6d0a0000 6170702e s: %s.[0m...app.
3f404820 636f7265 696f742e 696f0000 1b5b303b coreiot.io...[0;
3f404830 33326d49 2028256c 75292025 733a2043 32mI (%lu) %s: C
3f404840 6f6e6e65 6374696e 6720746f 204d5154 onnecting to MQTT
```

```
PS D:\Study\Cryptology\YOLO_BIT> strings .\flash_contents_encryption.bin | sls "OPPO Reno4" -Context 0,6
PS D:\Study\Cryptology\YOLO_BIT> █
```

Figure 24: So sánh khả năng bảo mật dữ liệu khi mã hóa và khi chưa mã hóa

• Phân tích File gốc chưa mã hóa (YOLO\_BIT.elf)

```
1 xtensa-esp32-elf-objdump -s -j .flash.rodata .\build\YOLO_BIT.elf |
   sls "OPPO Reno4"
```

Quan sát:

- Công cụ objdump đọc nội dung phân vùng .flash.rodata (chứa dữ liệu hằng số) trong file ELF.
- Kết quả hiển thị dữ liệu dưới dạng văn bản rõ (plaintext). Có thể đọc trực tiếp:
  - \* Tên WiFi (SSID): "OPPO Reno4"
  - \* Password WiFi: hcmtk22
  - \* Các chuỗi log như: "Connecting to WiFi...", "IP address", "connecting to MQTT"
  - \* Địa chỉ máy chủ: "coreiot.io"
- **Kết luận:** Không có mã hóa → bất kỳ ai có file firmware hoặc dump được flash đều có thể xem được dữ liệu nhạy cảm.

• Phân tích dữ liệu trên Flash sau khi mã hóa (flash\_contents\_encryption.bin)

```
1 esptool.exe read_flash 0x20000 0x100000 flash_contents_encryption.bin
2 strings .\flash_contents_encryption.bin | sls "OPPO Reno4"
```

Quan sát:

- Không tìm thấy chuỗi "OPPO Reno4" (kết quả rỗng).
- Dữ liệu Flash đã bị phần cứng AES-256 mã hóa → biến thành byte ngẫu nhiên (ciphertext).
- Các chuỗi ASCII trước đây (chữ cái, số...) đều bị mã hóa và không thể đọc bằng công cụ strings.

Việc kết hợp Secure Boot V1 (đảm bảo tính toàn vẹn/xác thực) và Flash Encryption (đảm bảo tính bí mật) đã tạo nên một lớp vỏ bảo vệ toàn diện cho thiết bị IoT. Hệ thống giờ đây có khả năng chống lại cả các cuộc tấn công giả mạo phần mềm (Malware injection) lẫn các cuộc tấn công vật lý trích xuất dữ liệu (Physical readout), hoàn thiện mô hình bảo mật.



## 5 Kết luận

### 5.1 Kết luận chung

Trong bối cảnh các hệ thống IoT ngày càng phổ biến và len sâu vào nhiều lĩnh vực như nhà thông minh, công nghiệp, y tế, nông nghiệp... việc bảo vệ an toàn cho tầng thiết bị trở thành yêu cầu trọng tâm, bởi đây là điểm dễ bị tấn công vật lý và là nơi lưu trữ trực tiếp firmware cùng các tham số nhạy cảm. Báo cáo đã hệ thống hóa cơ sở lý thuyết về bảo mật IoT, phân tích kiến trúc 5 tầng, các mối đe dọa tương ứng, cùng các tiêu chuẩn bảo mật quốc tế và tiêu chuẩn Việt Nam. Đây là nền tảng để định hình yêu cầu và lựa chọn cơ chế bảo vệ phù hợp cho thiết bị ESP32 trong mô hình Smart Home.

Trên cơ sở đó, đề tài tập trung hiện thực hai cơ chế bảo mật phần cứng của ESP32: *Secure Boot V1* và *Flash Encryption*, kết hợp với *ECDSA* làm thuật toán chữ ký số. Secure Boot V1 dựa trên ROM Bootloader của ESP32 như một *Hardware Root of Trust*: ROM sử dụng khóa AES-256 trong eFuse Block 2 để tính toán và kiểm tra *Bootloader Digest* lưu tại địa chỉ 0x0; nếu khớp mới cho phép bootloader giai đoạn 2 (tại 0x1000) chạy. Ở tầng tiếp theo, bootloader sử dụng khóa công khai ECDSA được nhúng sẵn để xác thực chữ ký của firmware ứng dụng trước khi chuyển quyền điều khiển. Song song đó, Flash Encryption sử dụng AES-256 để mã hóa toàn bộ nội dung flash (bootloader, partition table, firmware, NVS), khiến mọi truy cập trực tiếp vào chip flash chỉ thu được ciphertext vô nghĩa.

Về mặt thuật toán, báo cáo đã lựa chọn ECDSA cho phép dùng kích thước khóa nhỏ hơn rất nhiều, từ đó giảm tải tài nguyên CPU, bộ nhớ và năng lượng phù hợp với đặc thù thiết bị IoT, vì điều khiển và các hệ thống nhúng. Đồng thời, ECDSA được ESP-IDF hỗ trợ trực tiếp trong quy trình ký và xác minh firmware, giúp tích hợp thuận lợi vào Secure Boot V1. Tuy nhiên, ECDSA cũng có nhược điểm nhất định: cấu trúc toán học phức tạp, nhạy cảm với chất lượng số ngẫu nhiên (nonce) và dễ sinh lỗi nếu triển khai sai, do đó đòi hỏi quy trình quản lý khóa và môi trường build phải được bảo vệ chặt chẽ.

Trong cơ chế Secure Boot V1 ở chế độ *Reflashable* mà đề tài triển khai, khóa bí mật ECDSA giữ vai trò trung tâm:

- Thứ nhất, khóa này dùng để ký số bootloader và firmware.
- Thứ hai, ESP-IDF dùng SHA-256 của file khóa ký để sinh ra Secure Boot Key AES-256 nạp vào eFuse Block 2, phục vụ việc tính toán Bootloader Digest.

Cách thiết kế này giúp tái lập được khóa bảo mật nếu cần nạp lại bootloader, nhưng cũng tạo ra sự phụ thuộc chặt giữa khóa ký và khóa phần cứng: nếu file khóa bị lộ, kẻ tấn công vừa có thể ký firmware giả mạo, vừa có khả năng tái tạo khóa bảo vệ bootloader. Ngược lại, nếu khóa bị mất, nhà phát triển mất hoàn toàn khả năng cập nhật firmware hợp lệ cho thiết bị. Đây là ưu điểm về tính đơn giản triển khai nhưng cũng là hạn chế về mặt quản lý vòng đời khóa.

Tuy nhiên, log phân tích trước khi bật Flash Encryption cho thấy dù Secure Boot đã ngăn không cho firmware giả mạo chạy, nội dung flash vẫn ở dạng plaintext: việc dump flash và chạy *strings* cho phép thu được SSID, mật khẩu WiFi, địa chỉ máy chủ MQTT và các chuỗi log nhạy cảm. Sau khi bật Flash Encryption, các thử nghiệm dump flash lại cho thấy không còn đọc được các chuỗi này, xác nhận rằng tính bí mật (Confidentiality) đã được đảm bảo. Kết hợp lại, Secure Boot V1 + ECDSA bảo đảm tính toàn vẹn và xác thực, trong khi Flash Encryption bảo đảm tính bí mật của firmware và dữ liệu.

Từ góc độ đánh giá tổng thể, có thể tóm lược:

#### Ưu điểm của ECDSA trong bối cảnh đề tài:

- Bảo mật cao với kích thước khóa nhỏ, phù hợp hạn chế tài nguyên của ESP32.
- Được ESP-IDF hỗ trợ trực tiếp, tích hợp tốt với chuỗi *build-sign-flash*.
- Thích hợp cho mô hình Smart Home và các hệ thống IoT cần tối ưu hiệu năng.

#### Nhược điểm/Thách thức của ECDSA:

- Việc sinh khóa và quản lý file khóa yêu cầu quy trình bảo mật nghiêm ngặt; nếu lô khóa, toàn bộ chuỗi tin cậy bị phá vỡ.
- Thuật toán nhạy với việc sử dụng nonce; nếu môi trường build, công cụ hoặc thư viện bị lỗi, có thể dẫn đến lỗ khóa riêng.



### Ưu điểm của Secure Boot V1 trên ESP32 Rev.1:

- Tận dụng trực tiếp ROM Bootloader làm Root of Trust phần cứng, không cần thêm phần cứng ngoài.
- Cơ chế Bootloader Digest giúp phát hiện mọi thay đổi dù rất nhỏ trong bootloader.
- Kết hợp eFuse để đảm bảo tính “một lần cấu hình – không đảo ngược”.

### Hạn chế của Secure Boot V1 so với V2 (ở mức độ khái quát):

- Phụ thuộc vào vùng digest riêng tại 0x0, làm quy trình nạp và cập nhật bootloader phức tạp hơn (phải sinh và nạp đúng digest tương ứng).
- Ràng buộc giữa khóa ký và khóa AES của bootloader (trong chế độ Reflashable) khiến rủi ro quản lý khóa cao hơn.
- Không hỗ trợ đầy đủ các cơ chế quản lý nhiều khóa/public key, xoay vòng hoặc thu hồi khóa linh hoạt như Secure Boot V2 trên các revision mới hơn của ESP32.

Như vậy, đề tài đã hiện thực thành công một mô hình bảo mật cho thiết bị IoT dựa trên ESP32 Rev.1, khai thác tối đa khả năng của Secure Boot V1, ECDSA và Flash Encryption để đạt được ba thuộc tính cốt lõi: *tồn vẹn – xác thực – bí mật* cho firmware và dữ liệu trong hệ thống Smart Home.

## 5.2 Hướng phát triển

Từ các kết quả đạt được và những hạn chế đã phân tích, một số hướng phát triển tiếp theo có thể được đề xuất như sau:

### Nâng cấp phần cứng để sử dụng Secure Boot V2

Về lâu dài có thể xem xét chuyển sang các dòng ESP32 revision mới có hỗ trợ Secure Boot V2. So với V1, V2 giảm phụ thuộc vào Bootloader Digest riêng tại 0x0, hỗ trợ nhiều slot khóa công khai, cơ chế thu hồi và xoay vòng khóa linh hoạt hơn, tách bạch tốt hơn giữa khóa bảo vệ bootloader và khóa ký firmware. Điều này giúp giảm rủi ro khi một khóa bị lộ, đồng thời tạo điều kiện xây dựng quy trình cập nhật/OTA an toàn trong suốt vòng đời thiết bị.

### Tách bạch và củng cố quản lý khóa ECDSA

Trong mô hình hiện tại, file Hiện tại, cùng một khóa ECDSA được dùng để ký firmware và đồng thời sinh ra khóa AES bảo vệ bootloader trong Secure Boot V1, tạo ra rủi ro nếu khóa bị lộ hoặc thất lạc. Hướng phát triển là tách riêng các vai trò khóa, sử dụng một khóa root chỉ cho bootloader và các khóa con dành cho firmware. Các khóa này cần được lưu trữ trong HSM hoặc Secure Element thay vì file .pem, nhằm tăng cường an toàn và quản lý vòng đời khóa hiệu quả hơn.

### Xây dựng cơ chế OTA an toàn trên nền Secure Boot và Flash Encryption

Hiện tại, firmware được nạp thủ công bằng esptool. Một hướng phát triển quan trọng là tích hợp OTA an toàn theo khuyến nghị IETF SUIT: firmware OTA được ký số ECDSA, đóng gói manifest phiên bản, hash, metadata; thiết bị kiểm tra chữ ký, phiên bản và tính toàn vẹn trước khi ghi lên flash (đã mã hóa). Khi kết hợp với Secure Boot V2, hệ thống sẽ hình thành chuỗi tin cậy xuyên suốt từ server phát hành đến thiết bị cuối, đáp ứng tốt yêu cầu quản lý vòng đời thiết bị IoT.

### Tự động hóa toolchain và quy trình kiểm thử bảo mật

Quy trình hiện tại yêu cầu nhiều thao tác thủ công (tạo khóa, burn eFuse, encrypt flash thủ công lần sau...). Một bước phát triển tự nhiên là xây dựng script tự động hóa (Python/CI pipeline) để giảm sai sót con người, đi kèm với bộ kiểm thử tự động: kiểm tra log secure boot, thử nạp firmware không ký, bootloader “là”, dump flash sau mã hóa... Điều này giúp mô hình dễ dàng nhân rộng cho nhiều thiết bị và nhiều phiên bản firmware.



## 6 Tài liệu tham khảo

1. Zenarmor, "What Is IoT Security?", Zenarmor Network Security Tutorials, 04 July 2024. Truy cập: [link](#)
2. Viblo, "ECDSA – Hệ mật dựa trên đường cong Elliptic và ứng dụng trong Blockchain", Viblo, truy cập: [link](#)
3. SSL.com, "Comparing ECDSA vs RSA: A Simple Guide", SSL.com, truy cập: [link](#)
4. PBearson, "ESP32\_Secure\_Boot\_Tutorial," GitHub, 2021. Truy cập: [link](#)



## 7 Phụ lục

Mã nguồn firmware (Source code) . File: main.cpp

```
1  /**
2   * File Name: main.cpp
3   * Description: Demo Firmware Source Code for IoT Security Project (Secure
4   *               Boot & Flash Encryption).
5   * Features:
6   * 1. Connects to WiFi and MQTT Server (contains sensitive data requiring
7   *     protection).
8   * 2. Displays status on I2C LCD screen.
9   * 3. Sends periodic "heartbeat" messages to the Server to verify system
10    stability.
11 */
12
13 #include <Arduino.h>
14 #include "esp_log.h"
15 #include <Wire.h>
16 #include <LiquidCrystal_I2C.h>
17 #include <WiFi.h>
18 #include <PubSubClient.h>
19
20 // =====
21 // SENSITIVE DATA / SECRETS
22 // Note: These are hardcoded Plaintext credentials.
23 // The goal of Flash Encryption is to encrypt these strings in Flash memory
24 // to prevent attackers from extracting them (via Flash Dump).
25 // =====
26 #define WIFI_SSID          "OPPO Reno4"      // WiFi SSID (Exposed in Figure 14 if
27 // not encrypted)
28 #define WIFI_PASSWORD       "hcmutk22"        // WiFi Password (Exposed in Figure
29 // 14 if not encrypted)
30
31 #define MQTT_BROKER         "app.coreiot.io"
32 #define MQTT_PORT           1883
33 #define MQTT_USER            "iot_device_test"
34 #define MQTT_PASSWORD        "123456"          // MQTT Password
35 #define MQTT_CLIENT_ID       "IOT_DEVICE_TEST_SECURE"
36
37 // TOPICS
38 #define MQTT_TOPIC_PING      "v1/devices/me/telemetry"
39
40 static const char *TAG = "SECURE_APP";
41
42 // I2C LCD Configuration (Address 0x27 or 0x21 depending on the board)
43 LiquidCrystal_I2C lcd(0x21, 16, 2);
44
45 // Initialize MQTT Client
46 WiFiClient espClient;
47 PubSubClient mqttClient(espClient);
48
49 // =====
50 // FUNCTION: CONNECT WIFI
51 // =====
52 void connectWiFi() {
53     ESP_LOGI(TAG, "Connecting to WiFi...");
54     WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
55
56     // Wait for connection
57     while (WiFi.status() != WL_CONNECTED) {
58         vTaskDelay(1000 / portTICK_PERIOD_MS);
59         ESP_LOGI(TAG, ".");
60     }
61 }
```



```
55 }
56
57     ESP_LOGI(TAG, "WiFi connected!");
58     // Print IP for verification (also sensitive data if unprotected)
59     ESP_LOGI(TAG, "IP address: %s", WiFi.localIP().toString().c_str());
60 }
61
62 // =====
63 // FUNCTION: HANDLE INCOMING MQTT MESSAGES (CALLBACK)
64 // =====
65 void mqttCallback(char *topic, byte *payload, unsigned int length) {
66     ESP_LOGI(TAG, "MQTT message on topic: %s", topic);
67
68     // Dynamically allocate memory for payload processing
69     char* payload_str = (char*)malloc(length + 1);
70     if (!payload_str) {
71         ESP_LOGE(TAG, "Failed to allocate memory for payload");
72         return;
73     }
74     memcpy(payload_str, payload, length);
75     payload_str[length] = '\0'; // Null-terminate string
76
77     ESP_LOGI(TAG, "Payload: %s", payload_str);
78     free(payload_str); // Free memory to avoid memory leaks
79 }
80
81 // =====
82 // FUNCTION: CONNECT TO MQTT BROKER
83 // =====
84 void connectMQTT() {
85     mqttClient.setServer(MQTT_BROKER, MQTT_PORT);
86     mqttClient.setCallback(mqttCallback);
87
88     while (!mqttClient.connected()) {
89         ESP_LOGI(TAG, "Connecting to MQTT...");
90
91         // Attempt connection with Client ID and User/Pass
92         if (mqttClient.connect(MQTT_CLIENT_ID, MQTT_USER, MQTT_PASSWORD)) {
93             ESP_LOGI(TAG, "MQTT connected!");
94         } else {
95             // If failed, print error code and retry after 2 seconds
96             ESP_LOGE(TAG, "Failed, rc=%d. Retrying in 2 seconds...", mqttClient.state());
97             vTaskDelay(2000 / portTICK_PERIOD_MS);
98         }
99     }
100 }
101
102 // =====
103 // FUNCTION: SETUP (SYSTEM INITIALIZATION)
104 // =====
105 void setup() {
106     Serial.begin(115200);
107
108     // 1. Initialize LCD screen
109     Wire.begin();
110     lcd.init();
111     lcd.backlight();
112     lcd.clear();
113
114     // Display welcome message
115     lcd.setCursor(0, 0);
116     lcd.print("Secure Boot App");
```



```
117     lcd.setCursor(0, 1);
118     lcd.print("System Ready");
119
120     // 2. Establish network connections
121     connectWiFi();
122     connectMQTT();
123
124     // 3. CRITICAL LOG: Marks system passing Secure Boot check
125     // Used in Figure 11 and Figure 20 of the report to prove
126     // Firmware authentication and execution success.
127     ESP_LOGI(TAG, "System ready for Secure Boot!");
128 }
129
130 // =====
131 // FUNCTION: LOOP (MAIN EXECUTION LOOP)
132 // =====
133 void loop() {
134     // Ensure MQTT connection persists
135     if (!mqttClient.connected()) {
136         connectMQTT();
137     }
138     mqttClient.loop();
139
140     // Send heartbeat every 5 seconds to indicate device is "Alive"
141     static unsigned long lastPing = 0;
142     unsigned long now = millis();
143     if (now - lastPing > 5000) {
144         lastPing = now;
145         // Simple JSON Payload
146         const char *payload = "{\"status\":\"alive\",\"app\":\"secure-boot-test\"}";
147         mqttClient.publish(MQTT_TOPIC_PING, payload);
148         ESP_LOGI(TAG, "MQTT ping: %s", payload);
149     }
150
151     // Blinking character effect on LCD to indicate non-blocking execution
152     static bool toggle = false;
153     lcd.setCursor(15, 1);
154     lcd.print(toggle ? "*" : " ");
155     toggle = !toggle;
156
157     delay(500);
158 }
159
160 // =====
161 // APP MAIN: BRIDGE BETWEEN ESP-IDF AND ARDUINO CORE
162 // Since the project uses ESP-IDF framework but code is written in Arduino
163 // style,
164 // this function is required to start the Arduino environment.
165 // =====
166 extern "C" void app_main() {
167     initArduino();          // Initialize Arduino Core
168     setup();                // Call user setup()
169     while (true) {
170         loop();              // Call user loop() repeatedly
171         vTaskDelay(1 / portTICK_PERIOD_MS); // Yield CPU to background
172             FreeRTOS tasks (Watchdog, WiFi...)
173     }
174 }
```