**977-302 Digital Engineering Project I**

Semester 2/2024

# A framework for secure BLE-enabled IoT application with T.E.E

Project Report

**Suttipon Rattana 6530613032**

Advisor: **ASST.PROF.DR. NORRATHEP RATTANAVIPANON**

*[Advisor's signature]* [illegible]

**College of Computing**

**Prince of Songkla University, Phuket Campus**

**Project Title**

**A framework for secure BLE-enabled IoT application with T.E.E**

| | |
|---|---|
| **Author** | Suttipon Rattana 6530613032 |
| **Major** | Digital Engineering |
| **Academic Year** | 2024 |

## Abstract

With the increasing adoption of Bluetooth Low Energy (BLE) devices in critical applications such as healthcare and secure communications, ensuring their security is essential. This project proposes a framework for a secure BLE-enabled IoT authentication mechanism utilizing Trusted Execution Environment (TEE) through Trusted Firmware-M (TF-M). The focus is on integrating security features to protect sensitive operations in Bluetooth-enabled devices, particularly those based on the nRF5340 System on Chip (SoC). By leveraging TF-M, the system ensures robust security through the separation of the secure world and non-secure world, mitigating risks such as unauthorized access and firmware tampering.
This implementation aims to enhance BLE security by introducing an authentication mechanism that enables cryptographic attestation and verification through a smartphone-based application. The project ensures that only authenticated devices can participate in BLE communication, thereby improving overall IoT security while maintaining efficiency and compatibility with existing BLE infrastructure.

Keywords: Bluetooth Low Energy (BLE), Trusted Firmware-M (TF-M), Trusted Execution Environment (TEE), nRF5340, IoT Security, Secure Authentication

# Acknowledgements

I would like to express my sincere gratitude to ASST. PROF. DR. NORRATHEP RATTANAVIPANON for his invaluable guidance, support, and expertise throughout this project. His insights and feedback have been instrumental in shaping the research and development of this work.

Additionally, I would like to thank my peers and colleagues for their encouragement and collaboration, as well as my family and friends for their unwavering support throughout this journey.

Suttipon Rattana

21/04/2025

# TABLE OF CONTENTS

# LIST OF FIGURES

## LIST OF ABBREVIATIONS

- AES-CCM – Advanced Encryption Standard in Counter with CBC-MAC mode
- BLE – Bluetooth Low Energy
- CA – Certificate Authority
- ECDH – Elliptic Curve Diffie-Hellman
- E2EE – End-to-End Encryption
- HCI – Host Controller Interface
- IPC – Inter-Process Communication
- MITM – Man-in-the-Middle
- NUS – Nordic UART Service
- OOB – Out-of-Band
- PKI – Public Key Infrastructure
- PSA – Platform Security Architecture
- TF-M – Trusted Firmware-M
- TEE – Trusted Execution Environment
- Zephyr – Zephyr Real-Time Operating System

# Chapter 1 Introduction

## 1 Introduction

## 1.1 Overview

The growing number of IoT devices has made Bluetooth Low Energy (BLE) a key technology for wireless communication in areas like medical monitoring, smart home systems, and wearable devices. According to Bluetooth SIG [1], BLE is widely used because of its low power consumption and efficient wireless communication capabilities. While BLE has built-in security features, such as authentication to prevent man-in-the-middle (MITM) attacks, these protections have some major limitations. As noted by Bluetooth SIG, if an attacker manages to compromise a device after or during authentication, they can bypass security measures and gain unauthorized access to data [2].

**BLE Pairing and Authenticating**
Pairing is the first step in establishing a secure connection between two BLE devices. According to Bluetooth SIG [1], BLE provides different pairing methods, including:

- Just Works: Easy to use, but offers no protection against MITM attacks.
- Passkey Entry: Requires both devices to enter a six-digit code, making it harder for an attacker to intercept.

Numeric Comparison: Displays a code on both devices that users must confirm, ensuring the connection is legitimate.

- Out-of-Band (OOB): Pairing uses an external secure channel like NFC to exchange encryption keys, making it one of the safest options.

Authentication plays a key role in preventing MITM attacks by ensuring that only trusted devices can communicate. However, BLE devices remain vulnerable to security risks after the pairing process is complete. As Bluetooth SIG points out, attackers who gain access to a device post-authentication can bypass these protections, making authentication ineffective in the long run.

**Why BLE Security Is Limited**
While BLE does implement authentication mechanisms, certain weaknesses still leave devices exposed. For example:

- Key Overwriting Attacks: The "BLURtooth" vulnerability, as reported by Threatpost, allows attackers to overwrite encryption keys in Bluetooth versions 4.0 to 5.0, potentially exposing data.

- Unauthorized Access After Initial Pairing: According to Bitdefender [3], once an attacker gains access to a device, they can manipulate connections and bypass authentication, rendering security useless.

These security gaps show that BLE's built-in defenses alone are not enough for protecting sensitive applications.

Improving Security with Trusted Execution Environments (TEE) To strengthen BLE security, Trusted Execution Environments (TEE) offer a more robust solution. Trusted Firmware-M (TF-M) is an open-source implementation of TEE for Arm Cortex-M processors. As described by Trusted Firmware, TF-M helps prevent attackers from accessing critical data, even if the main application is compromised.

For example, the nRF5340 microcontroller supports Arm TrustZone, which allows secure and non-secure applications to run separately. According to the Zephyr Project, this hardware-based isolation ensures that even if an attacker exploits vulnerabilities in the non-secure portion of the system, the secure area remains protected.

In this project, we leverage this separation to enable remote attestation over Bluetooth Low Energy (BLE). By integrating TF-M into BLE-based IoT devices, the system can generate cryptographic evidence, such as random values and attestation tokens, from the secure world and deliver it to a remote verifier (e.g., a mobile device). This allows the verifier to assess whether the device is running trusted firmware, providing stronger guarantees than traditional BLE pairing methods. As security risks continue to evolve, incorporating remote attestation into BLE communication is becoming an essential enhancement for trust-critical IoT applications.

## 1.2 Objectives

The primary objective of this project is to enhance the security of BLE-based IoT devices by introducing a backward-compatible authentication mechanism that ensures software integrity while maintaining interoperability with existing BLE systems. This mechanism will leverage Trusted Firmware-M (TF-M) and secure remote attestation to verify device authenticity without requiring hardware modifications to legacy devices.

Additionally, this project aims to develop a user-friendly framework that enables smartphone users to seamlessly authenticate BLE-based IoT devices. The framework will build upon standard BLE protocols, allowing users to verify device integrity through a mobile application that performs cryptographic checks and validates software authenticity in real time.

To validate the effectiveness of the proposed approach, the framework will be demonstrated through real-world BLE IoT applications, ensuring practical feasibility and security improvements. The demonstration will involve secure authentication in applications such as smart home automation, medical IoT devices, and industrial IoT systems, where BLE security is critical.

## 1.3 Scope

This project focuses on designing and implementing a secure BLE framework with integrated attestation using Trusted Firmware-M (TF-M). The main areas covered include:

- Implementing secure BLE pairing using Secure Connections Only (SC-Only) mode to establish encrypted connections and prevent unauthorized access.
- Integrating the Heart Rate Service (HRS) to simulate BLE data communication and test interaction between the mobile application and the BLE device.
- Porting TF-M into the project to enable communication between the non-secure BLE application and the secure world using Inter-Process Communication (IPC).
- Generating attestation-related data within TF-M, including secure random values and printing the public attestation key for verification purposes.
- Simulating a challenge-response mechanism, where a 16-byte challenge from the mobile device is processed and returned as a 64-byte dummy response.
- Utilizing the Nordic UART Service (NUS) as a flexible communication channel between the BLE device and the mobile application, with support for chunked BLE notifications.
- Preparing the framework for future integration of real attestation signing and mobile-side verification

## 1.4 Procedures

The project will follow a structured methodology to develop and validate the proposed BLE authentication mechanism. The key phases of the procedure include:

**Research and Requirement Analysis**

- Conduct an in-depth review of existing BLE authentication methods, their limitations, and security vulnerabilities.

- Identify relevant BLE security standards and industry best practices to inform the design of the authentication mechanism.

**System Design and Architecture**

- Design a secure authentication mechanism leveraging Trusted Firmware-M (TF-M) and secure remote attestation principles.
- Develop an architectural framework that integrates authentication with existing BLE communication protocols.
- Plan how the authentication mechanism will function within real-world BLE IoT environments.

**Implementation of the Authentication Mechanism**

- Develop the authentication system using secure firmware attestation and cryptographic verification techniques.
- Implement the authentication protocol on an nRF5340 BLE device to test its functionality.
- Ensure compatibility with existing BLE infrastructure and various IoT devices.

**Development of the User-Friendly Framework**

- Design and develop a mobile application that allows users to verify BLE IoT devices seamlessly.
- Implement cryptographic verification tools within the app to ensure software integrity and authentication.
- Integrate the framework with BLE pairing processes for seamless usability.

**Testing and Validation**

- Conduct controlled experiments using real-world BLE IoT applications, such as smart home automation, medical IoT devices, and industrial IoT systems.
- Evaluate the authentication system’s effectiveness in detecting and preventing unauthorized access.

**Documentation and Deployment**

- Document the findings, system architecture, and implementation details.
- Provide guidelines for integrating the authentication framework into different BLE-based IoT applications.
- Explore potential deployment strategies for real-world adoption.

## 1.5 Expected Project Results

This project is expected to deliver the following results:

- A secure BLE authentication mechanism that enhances device integrity verification beyond traditional BLE pairing methods.
- A functional and user-friendly mobile framework that allows smartphone users to verify BLE IoT devices securely.
- Improved security for BLE-based IoT applications, ensuring that only trusted devices can connect and communicate.
- Successful validation through real-world applications, demonstrating that the proposed authentication mechanism can enhance security in smart home automation and medical IoT.

## 1.6 Development Tool

The development of this project will rely on the following tools and platforms:

- Ubuntu Linux: The primary operating system for development and testing, providing a stable and safe environment in case anything fails.
- nRF5340 Development Kit: The BLE-compatible hardware for implementing and testing the authentication mechanism.
- nRF Connect App: A tool for testing BLE connectivity, debugging interactions, and monitoring device authentication.
- LightBlue App: A BLE scanning and debugging application used to analyze BLE connections and validate authentication functionality.
- Visual Studio Code (VS Code): The integrated development environment (IDE) used for coding, debugging, and managing firmware and software development.

These tools will support the implementation, testing, and validation of the BLE security framework, ensuring seamless integration with real-world BLE IoT applications.

# Chapter 2 Literature Review

## 2.1 Bluetooth Low Energy (BLE) Pairing and Security

### 2.1.1 BLE Pairing Mechanisms and MITM Attack Prevention

Bluetooth Low Energy (BLE) pairing is a crucial security mechanism that allows devices to establish a mutually authenticated and encrypted communication channel. Its primary purpose is to prevent unauthorized access, ensure data confidentiality, and mitigate threats such as Man-in-the-Middle (MITM) attacks. The security provided by BLE pairing is vital for applications that require integrity and privacy, including IoT devices, medical equipment, and secure access control systems [2].

**How BLE Pairing Works**

The BLE pairing process consists of many stages that ensure a secure connection between two devices:

- Pairing Feature Exchange: During this phase, devices communicate their security capabilities and establish an appropriate pairing method.
- Key Generation and Distribution: Cryptographic keys are generated and exchanged to enable a secure session.
- Encryption and Authentication: A secure session is established based on the generated keys, assuring the confidentiality and integrity.

BLE supports different pairing methods to accommodate various security requirements, such as:

- Just Works: A simple pairing process with no authentication, making it highly vulnerable to MITM attacks [2].
- Passkey Entry: The user must enter a 6-digit key on both devices, adding an authentication step to prevent MITM threats [4].
- Numeric Comparison: Both devices display a 6-digit code, and users confirm that they are matching, ensuring mutual authentication [5].
- Out of Band Pairing (OOB): Uses an external secure channel (like NFC) to exchange cryptographic keys, improving the security significantly [5]

These pairing methods are designed to balance usability and, most importantly, security. Allowing devices to choose the most appropriate method based on their capabilities and the required security level. Some implementations also incorporate time-based key expiration and additional re-authentication mechanisms to enhance the security even more.

## Man-in-the-Middle (MITM) Attacks and Prevention

MITM attacks happen when an unauthorized entity intercepts and potentially alters communication between two devices. BLE mitigates these attacks through robust security measures, including:

- Elliptic Curve Diffie-Hellman Key Exchange (ECDH): Makes sure that only the intended devices can compute the shared encryption keys, preventing any intermediaries from deciphering the communication
- AES-CCM Encryption: Applies authenticated encryption to protect data during transmission, maintaining both confidentiality and integrity.
- Session Key Derivation:  Generated unique session keys for each connection, reducing the risk of replay attacks and session hijacking [4].
- Dynamic Key Rotation: Frequent key rotation prevents prolonged exposure of cryptographic keys, mitigating the risks of brute-force attacks.
- Multi-Factor Authentication (MFA): Some BLE security implementations integrate additional authentication layers such as biometric validation and user confirmation to further prevent unauthorized access.

### Mechanisms to Prevent Network Adversary Attacks

Ble doesn't only prevent MITM attacks. It also incorporates additional mechanisms to counteract various network security threats, such as eavesdropping, device impersonation, and replay attacks:

- Mutual Authentication: Devices verify each other's identity before establishing communication, ensuring that only authorized devices can connect.
- End-to-End Encryption (E2EE): Data is encrypted from the source to the destination, ensuring that only authorized parties can access the transmitted information.
- Tamper-Resistant Key Storage: Ensures physical attacks on BLE-enabled devices cannot easily extract cryptographic keys.

### 2.1.2 Platform Security Architecture (PSA) Certified API

### What is PSA?

Platform Security Architecture (PSA) is an industry standard security framework developed by ARM to provide a structured approach to securing embedded and IoT systems [6].

PSA establishes security best practices, including:

- Threat Modeling and Risk Assessment: Identifying potential security threats and assessing associated risks to inform the design of security measures.
- Secure boot and Firmware Integrity: Ensuring that devices boot with authenticated firmware, preventing the execution of malicious code.
- Device Authentication and Attestation: Verifying device identities and integrity to make sure that only the trusted devices participate in the network.

These practices provide a comprehensive security foundation for devices, enabling them to operate securely within diverse ecosystems.

#### What is PSA Certified API?

PSA Certified APIs offer standardized security services, ensuring compliance with PSA principles across embedded applications.

These APIs provide functionalities such as:

- Cryptographic Operations: Standardized algorithms for encryption, hashing, and digital signatures, facilitating secure data handling.
- Device Attestation: Mechanisms to verify device authenticity and integrity, ensuring that devices have not been tampered with.
- Remote Firmware Update Verification: Enables devices to validate software updates before installation, ensuring that updates are from trusted sources.

By adhering to PSA Certified APIs, developers can build applications with consistent and robust security features, simplifying integration and enhancing trustworthiness.

**PSA Initial Attestation Module**

Device attestation is a crucial function within PSA that enables devices to prove their identity and integrity to remote parties.

The PSA Attestation API follows a structured process:

- Input: The attestation request includes the device's identity, a cryptographic nonce, and security claims.
- Processing: The device generates a signed attestation token using a private key, encapsulating the provided information.
- Output: A signed attestation token is produced, which can be verified by a trusted authority to confirm the device's authenticity and integrity.
- Extended Verification Layers: Attestation data may also include additional security states, such as detected anomalies and recent firmware integrity checks.

## 2.2 Related Work: Bluetooth Security Beyond Pairing

While Bluetooth Low Energy (BLE) includes built-in security mechanisms such as pairing, bonding, and link-layer encryption, these features are primarily designed to secure communication channels, not to verify the integrity of the devices themselves. As a result, several research efforts and prototypes have attempted to extend BLE security beyond the pairing process, introducing stronger trust models based on cryptographic device identity, remote attestation, or hardware-based validation.

One notable direction involves integrating certificate-based authentication using Public Key Infrastructure (PKI). These systems allow devices to present signed certificates during or after pairing, enabling more reliable identification than traditional BLE methods. For example, the Certificate-Based Authentication Protocol (CBAP) introduces X.509-based device authentication layered on top of BLE GATT communication [12]. Silicon Labs also outlines a similar approach using Bluetooth Certificates [14]. Additionally, a patent by Intel proposes a BLE authentication method using asymmetric key cryptography and certificates [15]. However, none of these PKI-based methods are part of the official BLE specification and typically require custom implementation and firmware modification.

In parallel, more recent research has explored using remote attestation to verify device integrity over BLE. These approaches rely on Trusted Execution Environments (TEEs) to generate cryptographic responses that prove the firmware running on a device has not been tampered with. For instance, [13] presents a BLE attestation framework that uses TEE-backed challenge-response protocols to provide runtime integrity guarantees. While promising, such mechanisms are still experimental and have not yet been adopted into the BLE standard.

Despite these advancements, there is no standardized support for PKI or attestation in BLE today. This project directly addresses that gap by embedding a lightweight TF-M-based attestation mechanism into BLE communication, enabling mobile verifiers to validate device integrity without modifying the BLE stack.

### 2.2.1 Public Key Infrastructure (PKI) in Bluetooth Security

Public Key Infrastructure (PKI) is a well-established model for device authentication in secure communications, relying on asymmetric cryptography and certificate authorities. In the BLE context, several projects and proposals have explored using PKI to authenticate devices more robustly than traditional pairing methods.

CBAP, for example, adds X.509 certificate exchange on top of BLE GATT to enable certificate-based authentication [14]. Silicon Labs has also published guidance for implementing Bluetooth certificate exchange at the application level [16]. Additionally, a patent by Intel outlines a BLE-based certificate authentication method using asymmetric key cryptography [17].

However, none of these PKI-based solutions are part of the official BLE specification. They typically require custom firmware, changes to the BLE stack, and assume that both devices support the same PKI framework. This project does not attempt to implement full PKI but assumes the verifier already possesses the necessary public key. The focus is instead on verifying device integrity via TF-M and delivering attestation tokens to the verifier through standard BLE channels.

## 2.3 Communication Protocols in BLE

### 2.3.1 GATT in BLE Projects

GATT (Generic Attribute Profile) is the standard way most BLE devices send and receive data [7]. It organizes information into services and characteristics, which is perfect for things like heart rate monitors or battery level readings. A lot of existing BLE projects rely on GATT because it works well with mobile apps and is supported out of the box by most platforms.

In the early stages of this project, we used the Heart Rate Service (HRS) to test GATT-based communication. This helped us understand how BLE devices advertise data and how mobile apps can read or write to them. That setup worked fine for simple data exchange, and it matched what other developers have done in similar experiments [16]. But once we started planning how to send custom attestation messages back and forth, GATT started to feel limiting. It's not built for flexible, back-and-forth communication like what we needed for challenge-response.

### 2.3.2 Nordic UART Service (NUS) for Flexible Data Exchange

To get around those GATT limitations, we looked at other ways BLE projects handle custom data. One popular option is the Nordic UART Service (NUS), which lets a BLE device behave like a Bluetooth-connected serial port. We found that NUS is commonly used in prototypes or testing setups where developers want to send raw data without creating a whole custom GATT profile [17].

In this project, switching to NUS made things a lot easier. Instead of building custom characteristics, we could just send a 16-byte challenge from the phone and get a 64-byte response back, chunked into multiple notifications. This kind of flow would've been awkward to set up with GATT. Using NUS, let us focus more on testing the attestation logic, rather than fighting with the BLE service definitions.

## 3.1 Solution

This project aims to improve Bluetooth Low Energy (BLE) security for IoT applications by implementing a lightweight attestation protocol using Trusted Firmware-M (TF-M) on the nRF5340 DK, with communication over the Nordic UART Service (NUS) instead of standard GATT characteristics. Unlike traditional BLE pairing mechanisms that only secure the communication channel, this solution enables runtime software integrity checks through challenge-response authentication using a Trusted Execution Environment (TEE).

Here's Figure 1, which illustrates how the system works:

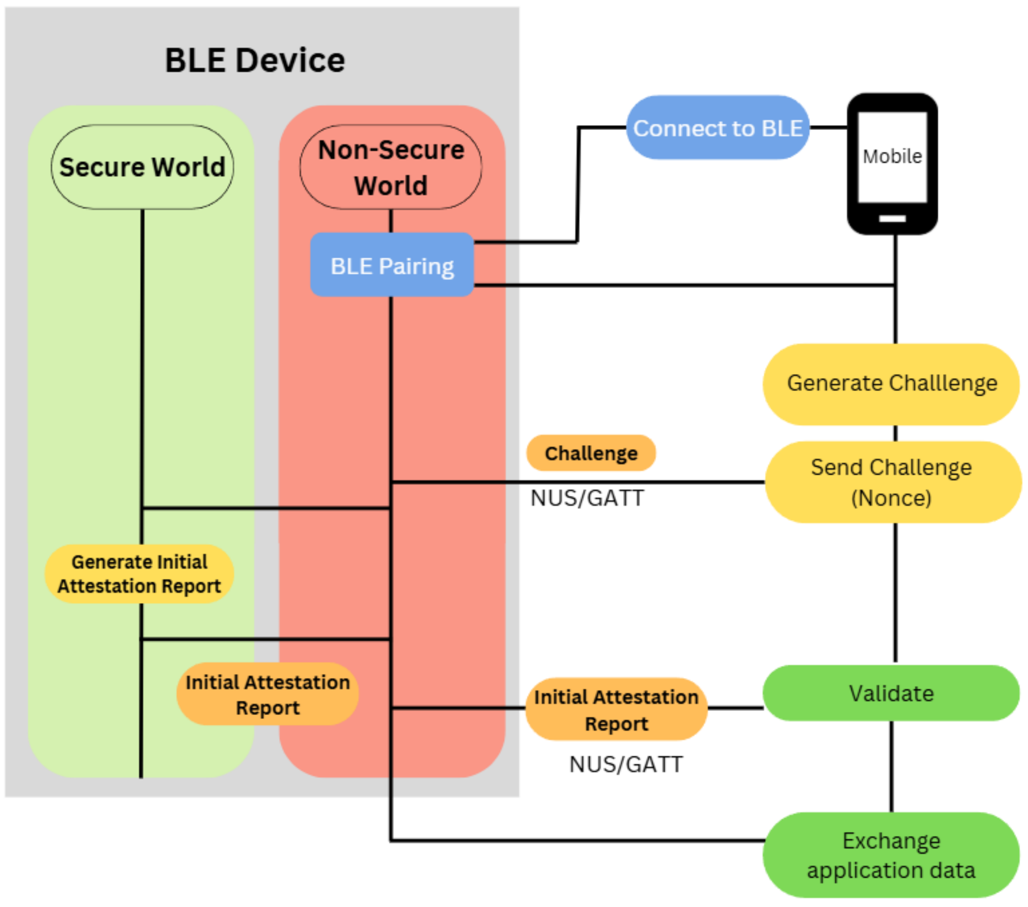


Figure 1 - How the system works: the phone sends a challenge, TF-M processes it, and the result comes back.

## 3.1 Component Breakdown (based on Figure 1)

### 3.1.1 Mobile Device

The mobile device acts as the verifier in the authentication process. After establishing a BLE connection (currently via GATT but NUS is pending), the mobile app sends a 16-byte random challenge (nonce) to the BLE device. Later, the mobile device receives a response (currently a dummy 64-byte value) for verification.

### 3.1.2 Non-Secure World (nRF5340 Application Core)

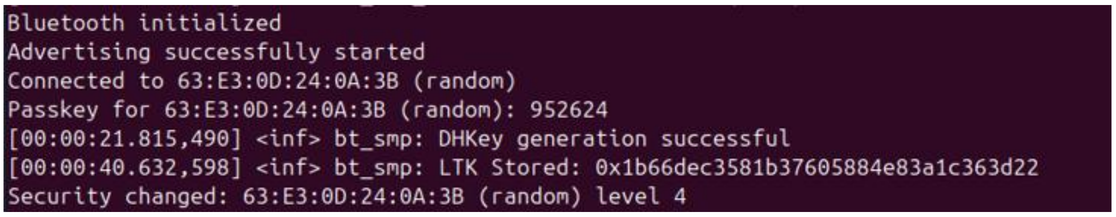
Receives the challenge from the mobile device. Passes the challenge to the Secure World using IPC (Inter-Process Communication). Forwards the attestation response back to the mobile device without modification. This layer acts as a secure bridge. It cannot generate or fake valid attestations.

### 3.1.3 Secure World (TF-M Secure Core)

This is where the actual attestation happens.
Once the challenge is received through IPC, TF-M validates the challenge format (replay protection can be added). It generates a response (Signed Initial Attestation report) and pushes it out to the non-secure world, which continues all the remaining processes.

## 3.2 Implementation Plan

In the beginning, we just ran the existing, peripheral_sc_only [8], which is a minimal BLE project that focuses on secure pairing by requiring a 6-digit passkey before any connection or data exchange can happen. This works, it asks for a 6-digit OTP when I try to connect to the board built with this project via Bluetooth LightBlue App.

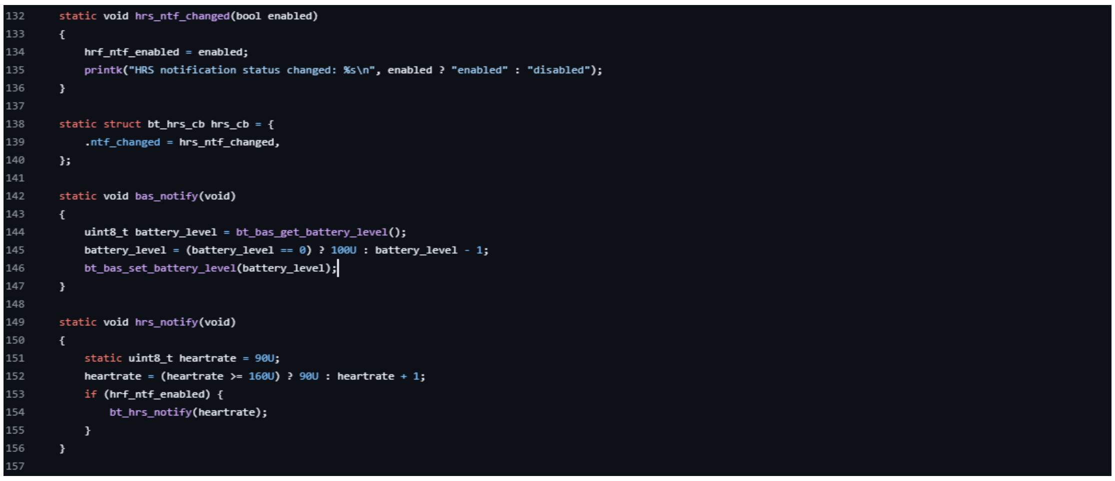


```
Bluetooth initialized
Advertising successfully started
Connected to 63:E3:0D:24:0A:3B (random)
Passkey for 63:E3:0D:24:0A:3B (random): 952624
[00:00:21.815,490] <inf> bt_smp: DHKey generation successful
[00:00:40.632,598] <inf> bt_smp: LTK Stored: 0x1b66dec3581b37605884e83a1c363d22
Security changed: 63:E3:0D:24:0A:3B (random) level 4
```

Figure 2 - BLE pairing with a passkey shown during a test using the LightBlue app.

As shown in Figure 2, a passkey is generated (952624), and once you successfully enter that passkey on your mobile device, you will be able to connect and exchange data inside.

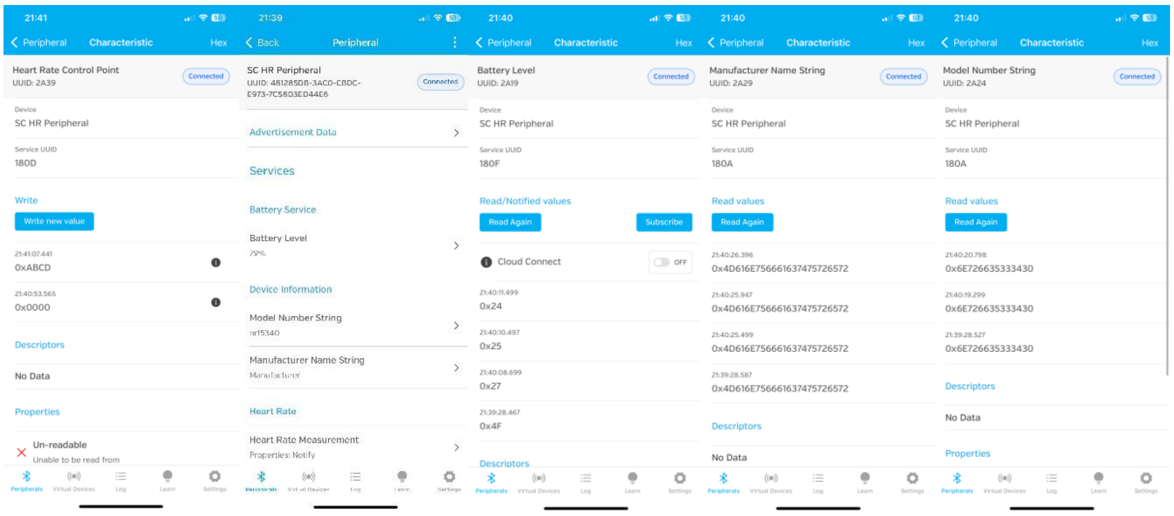
Now we do not have any data inside, so we ported in another project, peripheral_hr [9], which is a dummy heart rate monitor that sends dummy data and can receive what the user writes into it. This data transmission is done through GATT



```c
static void hrs_ntf_changed(bool enabled)
{
    hrf_ntf_enabled = enabled;
    printk("HRS notification status changed: %s\n", enabled ? "enabled" : "disabled");
}

static struct bt_hrs_cb hrs_cb = {
    .ntf_changed = hrs_ntf_changed,
};

static void bas_notify(void)
{
    uint8_t battery_level = bt_bas_get_battery_level();
    battery_level = (battery_level == 0) ? 100U : battery_level - 1;
    bt_bas_set_battery_level(battery_level);
}

static void hrs_notify(void)
{
    static uint8_t heartrate = 90U;
    heartrate = (heartrate >= 160U) ? 90U : heartrate + 1;
    if (hrf_ntf_enabled) {
        bt_hrs_notify(heartrate);
    }
}
```

Figure 3.1 - Shows the hrs_notify and bas_notify functions used in the peripheral_hr example.

hrs_notify() updates the heart rate value and sends a notification if enabled. And bas_notify() tweaks and updates the battery level. This aligns with your caption: dummy data generation for heart rate and receiving user-written data.



Heart Rate Control Point
UUID: 2A39
Connected
Device
SC HR Peripheral
Service UUID
180D
Write
21:41:07.441
0xABCD
21:40:53.565
0x0000
Descriptors
No Data
Properties
Un-readable
Unable to be read from

SC HR Peripheral
UUID: 481285DB-3AC0-CBDC-E973-7C58D3ED44E6
Connected
Advertisement Data
Services
Battery Service
Battery Level
79%
Device Information
Model Number String
nrf5340
Manufacturer Name String
Manufacturer
Heart Rate
Heart Rate Measurement
Properties: Notify

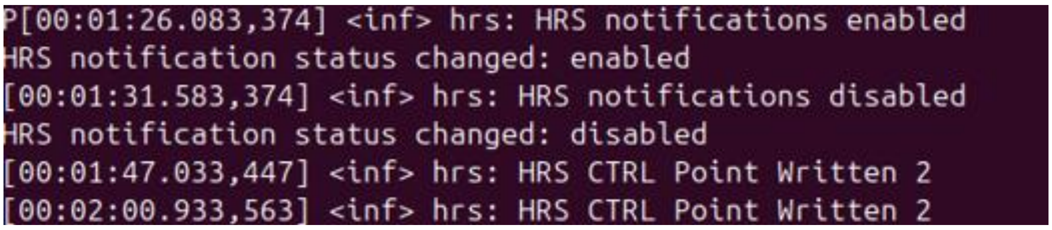
Battery Level
UUID: 2A19
Connected
Device
SC HR Peripheral
Service UUID
180F
Read/Notified values
Cloud Connect OFF
21:40:11.499
0x24
21:40:10.497
0x25
21:40:08.699
0x27
21:39:28.467
0x4F
Descriptors

Manufacturer Name String
UUID: 2A29
Connected
Device
SC HR Peripheral
Service UUID
180A
Read values
21:40:26.396
0x4D616E756661637475726572
21:40:25.947
0x4D616E756661637475726572
21:40:25.499
0x4D616E756661637475726572
21:39:28.587
0x4D616E756661637475726572
Descriptors
No Data

Model Number String
UUID: 2A24
Connected
Device
SC HR Peripheral
Service UUID
180A
Read values
21:40:20.798
0x6E726635333430
21:40:19.299
0x6E726635333430
21:39:28.527
0x6E726635333430
Descriptors
No Data
Properties

Figure 3.2 Data exchange on a mobile device using the Lightblue App

Figure 3.2 shows the mobile data exchange functions: read, write, and subscribe. Which, in all read functions, will generate dummy values, and in Write, you can write anything, and it will show on the board's screen as seen in Figure 3.3.



```
P[00:01:26.083,374] <inf> hrs: HRS notifications enabled
HRS notification status changed: enabled
[00:01:31.583,374] <inf> hrs: HRS notifications disabled
HRS notification status changed: disabled
[00:01:47.033,447] <inf> hrs: HRS CTRL Point Written 2
[00:02:00.933,563] <inf> hrs: HRS CTRL Point Written 2
```

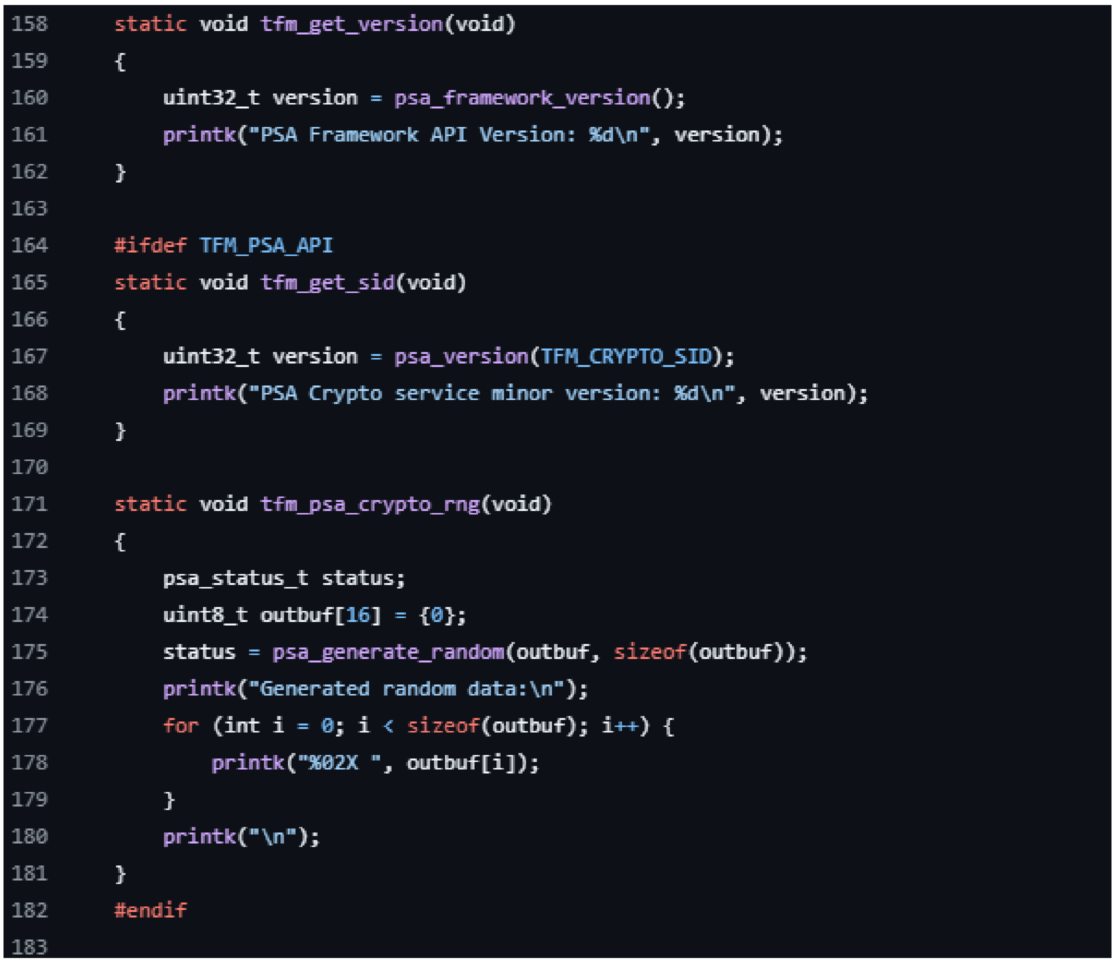
Figure 3.3 - Confirms that writing from the mobile app triggers logs on the device.

After setting up the BLE connection and dummy data exchange using the Heart Rate (HR) service, the next major step was integrating Trusted Firmware-M (TF-M) [10] into the project. This was done to test whether TF-M could be used to generate attestation results from the secure world, and whether those results could be accessed by the BLE application and eventually sent to the mobile device. The goal was to confirm that BLE and TF-M could work together for remote attestation.

We then ported a function called att_test(), which is called in main() after BLE initialization. This function acts as a proof-of-concept for calling secure services via TF-M. Inside this function, we added:

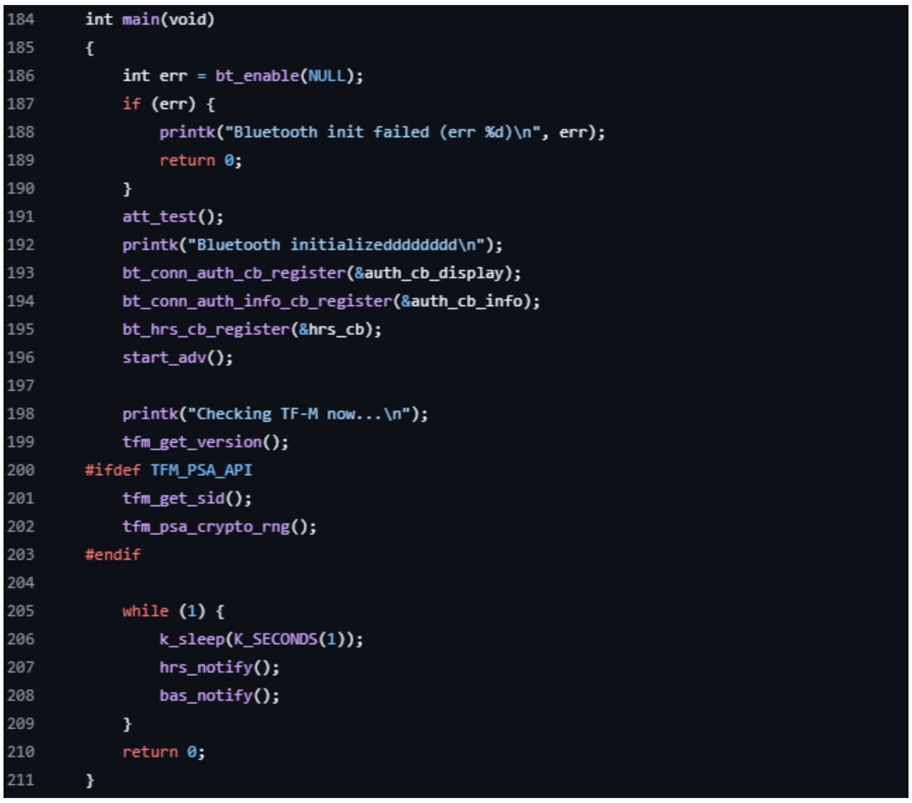
- A call to psa_generate_random(), which creates a 16-byte random value inside the secure world
- A function to print the attestation public key using LOG_HEXDUMP_INF() (used for future verification).
- Calls to psa_framework_version() and psa_version() to check if TF-M services are accessible.

To check if we could access the secure world from our BLE application, we wrote a few simple test functions. These included getting the TF-M framework version, checking the crypto service version, and generating random numbers from the secure side. The code we used for this is shown in Figure 4.1.



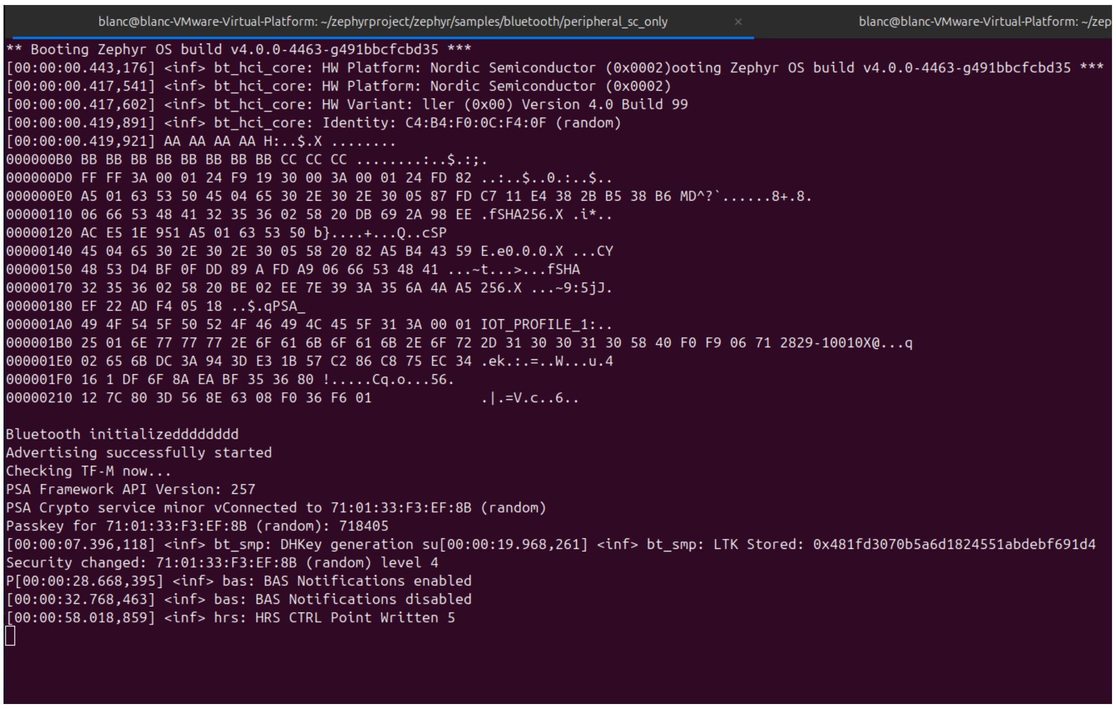
```c
static void tfm_get_version(void)
{
    uint32_t version = psa_framework_version();
    printk("PSA Framework API Version: %d\n", version);
}

#ifdef TFM_PSA_API
static void tfm_get_sid(void)
{
    uint32_t version = psa_version(TFM_CRYPTO_SID);
    printk("PSA Crypto service minor version: %d\n", version);
}

static void tfm_psa_crypto_rng(void)
{
    psa_status_t status;
    uint8_t outbuf[16] = {0};
    status = psa_generate_random(outbuf, sizeof(outbuf));
    printk("Generated random data:\n");
    for (int i = 0; i < sizeof(outbuf); i++) {
        printk("%02X ", outbuf[i]);
    }
    printk("\n");
}
#endif

```

Figure 4.1 - TF-M Secure World Function Calls for Version Checking and Random Number Generation



```c
int main(void)
{
    int err = bt_enable(NULL);
    if (err) {
        printk("Bluetooth init failed (err %d)\n", err);
        return 0;
    }
    att_test();
    printk("Bluetooth initializedddddddd\n");
    bt_conn_auth_cb_register(&auth_cb_display);
    bt_conn_auth_info_cb_register(&auth_cb_info);
    bt_hrs_cb_register(&hrs_cb);
    start_adv();

    printk("Checking TF-M now...\n");
    tfm_get_version();
#ifdef TFM_PSA_API
    tfm_get_sid();
    tfm_psa_crypto_rng();
#endif

    while (1) {
        k_sleep(K_SECONDS(1));
        hrs_notify();
        bas_notify();
    }
    return 0;
}
```

Figure 4.2 - Main BLE Application with TF-M Integration and Attestation Test Execution


blanc@blanc-VMware-Virtual-Platform: ~/zephyrproject/zephyr/samples/bluetooth/peripheral_sc_only
blanc@blanc-VMware-Virtual-Platform: ~/zep

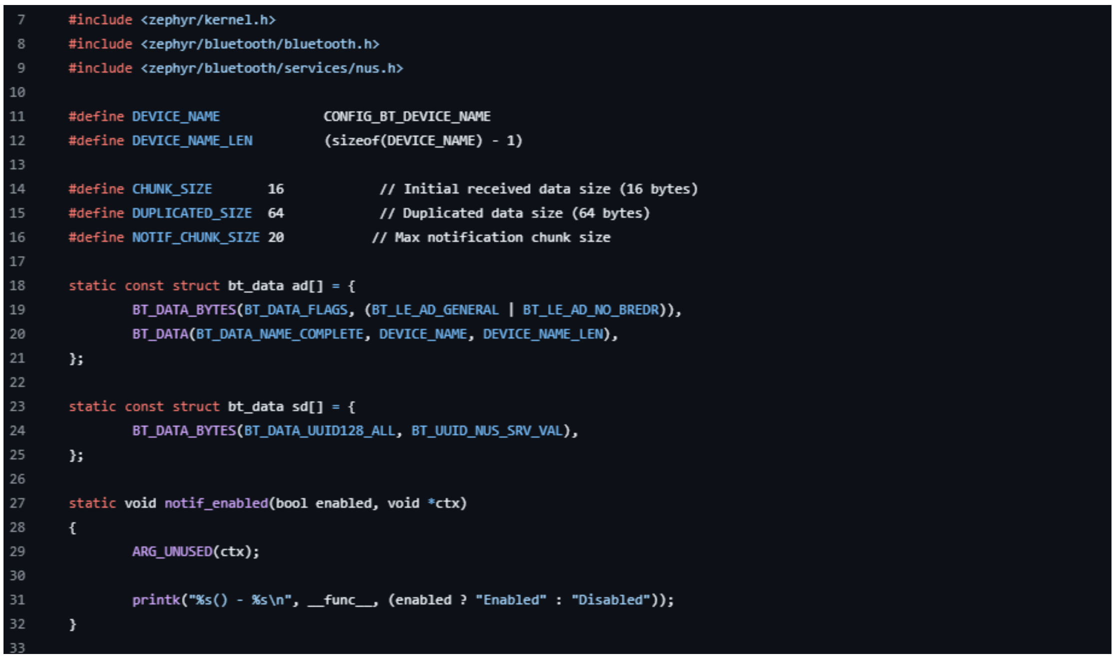
```
** Booting Zephyr OS build v4.0.0-4463-g491bbcfcbd35 ***
[00:00:00.443,176] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)ooting Zephyr OS build v4.0.0-4463-g491bbcfcbd35 ***
[00:00:00.417,541] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)
[00:00:00.417,602] <inf> bt_hci_core: HW Variant: ller (0x00) Version 4.0 Build 99
[00:00:00.419,891] <inf> bt_hci_core: Identity: C4:B4:F0:0C:F4:0F (random)
[00:00:00.419,921] AA AA AA AA H:..$.X ........
000000B0 BB BB BB BB BB BB BB BB CC CC CC ........:..$.:;.
000000D0 FF FF 3A 00 01 24 F9 19 30 00 3A 00 01 24 FD 82 ..:..$..0.:..$..
000000E0 A5 01 63 53 50 45 04 65 30 2E 30 2E 30 05 87 FD C7 11 E4 38 2B B5 38 B6 MD^?`......8+.8.
00000110 06 66 53 48 41 32 35 36 02 58 20 DB 69 2A 98 EE .fSHA256.X .i*..
00000120 AC E5 1E 951 A5 01 63 53 50 b}....+...Q..cSP
00000140 45 04 65 30 2E 30 2E 30 05 58 20 82 A5 B4 43 59 E.e0.0.0.X ...CY
00000150 48 53 D4 BF 0F DD 89 A FD A9 06 66 53 48 41 ...~t...>...fSHA
00000170 32 35 36 02 58 20 BE 02 EE 7E 39 3A 35 6A 4A A5 256.X ...~9:5jJ.
00000180 EF 22 AD F4 05 18 ..$.qPSA_
000001A0 49 4F 54 5F 50 52 4F 46 49 4C 45 5F 31 3A 00 01 IOT_PROFILE_1:..
000001B0 25 01 6E 77 77 77 2E 6F 61 6B 6F 61 6B 2E 6F 72 2D 31 30 30 31 30 58 40 F0 F9 06 71 2829-10010X@...q
000001E0 02 65 6B DC 3A 94 3D E3 1B 57 C2 86 C8 75 EC 34 .ek.:.=..W...u.4
000001F0 16 1 DF 6F 8A EA BF 35 36 80 !.....Cq.o...56.
00000210 12 7C 80 3D 56 8E 63 08 F0 36 F6 01             .|.=V.c..6..

Bluetooth initializedddddddd
Advertising successfully started
Checking TF-M now...
PSA Framework API Version: 257
PSA Crypto service minor vConnected to 71:01:33:F3:EF:8B (random)
Passkey for 71:01:33:F3:EF:8B (random): 718405
[00:00:07.396,118] <inf> bt_smp: DHKey generation su[00:00:19.968,261] <inf> bt_smp: LTK Stored: 0x481fd3070b5a6d1824551abdebf691d4
Security changed: 71:01:33:F3:EF:8B (random) level 4
P[00:00:28.668,395] <inf> bas: BAS Notifications enabled
[00:00:32.768,463] <inf> bas: BAS Notifications disabled
[00:00:58.018,859] <inf> hrs: HRS CTRL Point Written 5
```

Figure 4.3 – Entire Processed Displayed (with TFM and printing Attestation)

After successfully testing attestation using TF-M, the next goal was to send and receive data between the BLE device and a mobile phone more flexibly. To do this, we switched from using the standard GATT services (like Heart Rate Service) to using the Nordic UART Service (NUS), which is more developer-friendly and allows custom messages to be exchanged over BLE.
We used the official NUS code [11]

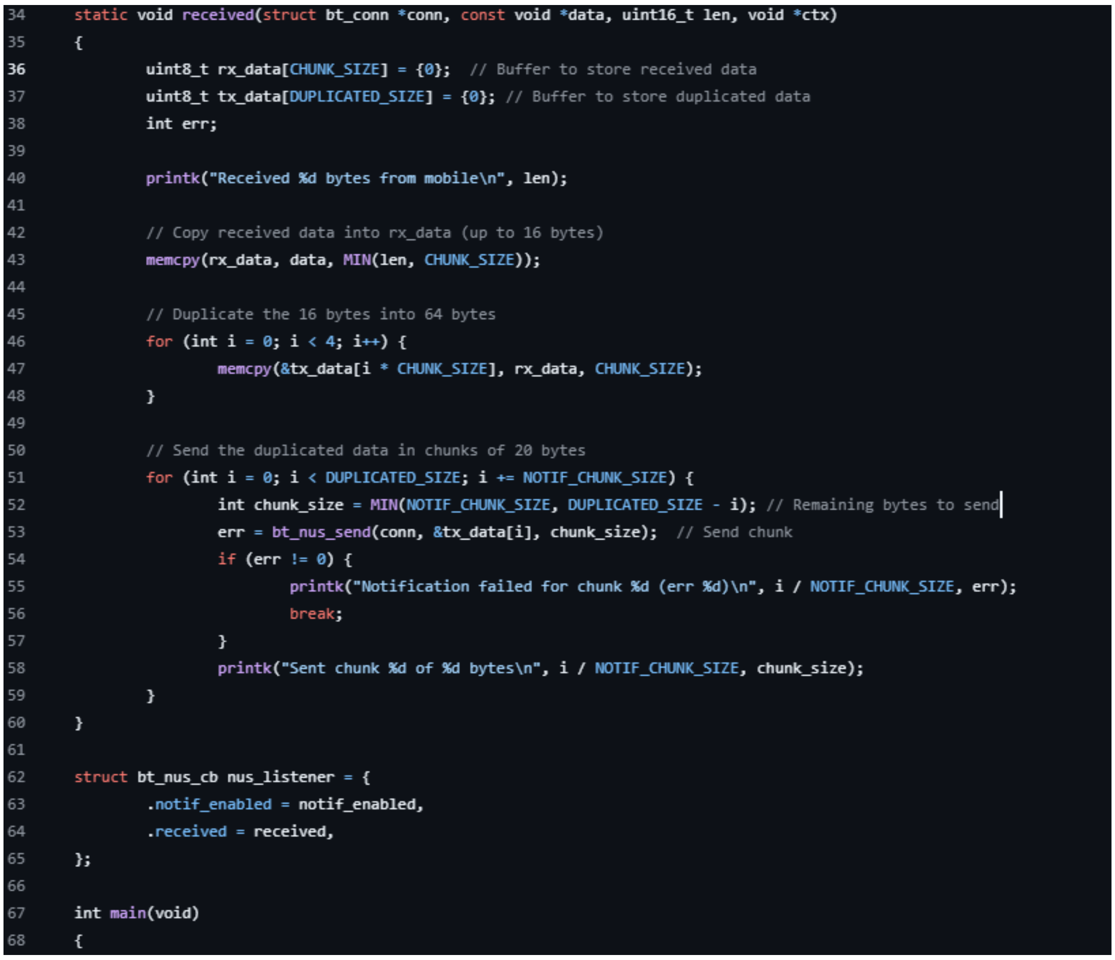
In this version of the project:

- The mobile phone sends a 16-byte challenge to the BLE device using the NUS write function.
- The BLE device receives that challenge inside the bt_nus_received() callback.
- The challenge is passed from the non-secure application to TF-M via IPC, which processes the data and generates a 64-byte dummy response (for now, we just duplicated the input).
- That response is then sent back to the mobile phone using bt_nus_send().
- The mobile will see the notification if the Subscribe feature is enabled



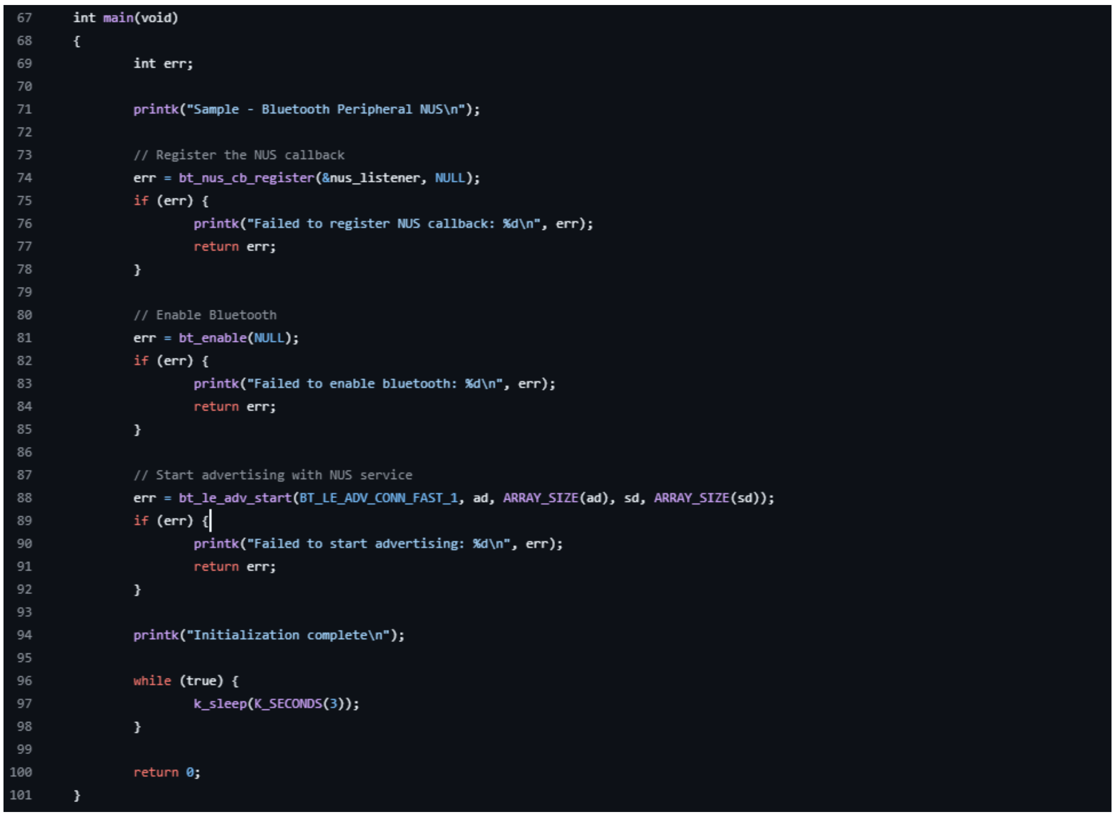
```c
#include <zephyr/kernel.h>
#include <zephyr/bluetooth/bluetooth.h>
#include <zephyr/bluetooth/services/nus.h>

#define DEVICE_NAME             CONFIG_BT_DEVICE_NAME
#define DEVICE_NAME_LEN         (sizeof(DEVICE_NAME) - 1)

#define CHUNK_SIZE       16           // Initial received data size (16 bytes)
#define DUPLICATED_SIZE  64           // Duplicated data size (64 bytes)
#define NOTIF_CHUNK_SIZE 20          // Max notification chunk size

static const struct bt_data ad[] = {
        BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_AD_NO_BREDR)),
        BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
};

static const struct bt_data sd[] = {
        BT_DATA_BYTES(BT_DATA_UUID128_ALL, BT_UUID_NUS_SRV_VAL),
};

static void notif_enabled(bool enabled, void *ctx)
{
        ARG_UNUSED(ctx);

        printk("%s() - %s\n", __func__, (enabled ? "Enabled" : "Disabled"));
}
```

Figure 5.1 - Advertising Data Setup and Subscribe Callback



```c
static void received(struct bt_conn *conn, const void *data, uint16_t len, void *ctx)
{
        uint8_t rx_data[CHUNK_SIZE] = {0};  // Buffer to store received data
        uint8_t tx_data[DUPLICATED_SIZE] = {0}; // Buffer to store duplicated data
        int err;

        printk("Received %d bytes from mobile\n", len);

        // Copy received data into rx_data (up to 16 bytes)
        memcpy(rx_data, data, MIN(len, CHUNK_SIZE));

        // Duplicate the 16 bytes into 64 bytes
        for (int i = 0; i < 4; i++) {
                memcpy(&tx_data[i * CHUNK_SIZE], rx_data, CHUNK_SIZE);
        }

        // Send the duplicated data in chunks of 20 bytes
        for (int i = 0; i < DUPLICATED_SIZE; i += NOTIF_CHUNK_SIZE) {
                int chunk_size = MIN(NOTIF_CHUNK_SIZE, DUPLICATED_SIZE - i); // Remaining bytes to send
                err = bt_nus_send(conn, &tx_data[i], chunk_size);  // Send chunk
                if (err != 0) {
                        printk("Notification failed for chunk %d (err %d)\n", i / NOTIF_CHUNK_SIZE, err);
                        break;
                }
                printk("Sent chunk %d of %d bytes\n", i / NOTIF_CHUNK_SIZE, chunk_size);
        }
}

struct bt_nus_cb nus_listener = {
        .notif_enabled = notif_enabled,
        .received = received,
};

int main(void)
{
```

Figure 5.2 - NUS receives the callback.



```c
int main(void)
{
        int err;

        printk("Sample - Bluetooth Peripheral NUS\n");

        // Register the NUS callback
        err = bt_nus_cb_register(&nus_listener, NULL);
        if (err) {
                printk("Failed to register NUS callback: %d\n", err);
                return err;
        }

        // Enable Bluetooth
        err = bt_enable(NULL);
        if (err) {
                printk("Failed to enable bluetooth: %d\n", err);
                return err;
        }

        // Start advertising with NUS service
        err = bt_le_adv_start(BT_LE_ADV_CONN_FAST_1, ad, ARRAY_SIZE(ad), sd, ARRAY_SIZE(sd));
        if (err) {
                printk("Failed to start advertising: %d\n", err);
                return err;
        }

        printk("Initialization complete\n");

        while (true) {
                k_sleep(K_SECONDS(3));
        }

        return 0;
}
```

Figure 5.3 - Main Function Flow: Registering NUS Callback, Enabling Bluetooth, and Starting Advertising

## 3.3 Project Plan

**Table 3-1.** Project plan.

| TASK | 2025 | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | January | | | | February | | | | March | | | | April | | | | May | | | |
| | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| Meeting with Advisor | | 🟦 | 🟦 | 🟦 | 🟦 | | 🟦 | 🟦 | | 🟦 | | 🟦 | | 🟦 | 🟦 | 🟦 | | | | |
| Study BLE Security Concepts | 🟦 | 🟦 | 🟦 | 🟦 | 🟦 | 🟦 | | | | | | | | | | | | | | |
| Run peripheral_sc_only sample | | | 🟦 | 🟦 | | | | | | | | | | | | | | | | |
| Port peripheral_hr for dummy data | | | | | 🟦 | 🟦 | | | | | | | | | | | | | | |
| Set up TF-M & test PSA API calls. | | | | | | | 🟦 | 🟦 | 🟦 | | | | | | | | | | | |
| Begin BLE ↔ TF-M communication (IPC) | | | | | | | | | | 🟦 | | | | | | | | | | |
| Implement att_test() with random + key print | | | | | | | | | | | 🟦 | 🟦 | | | | | | | | |
| Integrate and test Nordic UART Service (NUS) | | | | | | | | | | | | | 🟦 | 🟦 | | | | | | |
| Optimize BLE response handling (chunked send) | | | | | | | | | | | | | | | 🟦 | | | | | |
| Prepare figures, results, and documentation. | | | | | | | | | | | | | | | 🟦 | 🟦 | | | | |

# Chapter 4 Results and Conclusion

## 4.1 Results

So far, the project has made solid progress toward building a secure BLE authentication system. Here's what's been done:

- BLE pairing works using the peripheral_sc_only example. When connecting via apps like LightBlue, the board asks for a 6-digit code, showing basic BLE security is functional.
- A working prototype of secure attestation has been built using TF-M. It can generate random values and access the public key for future signature-based verification.
- Communication between the non-secure and secure world (TF-M) is up and running through Inter-Process Communication (IPC). This proves that the secure core can be reached from the BLE application.
- The system was also tested with a mobile phone. Through the Nordic UART Service (NUS), the phone sends a random challenge to the device, and the device responds back (currently with a dummy value).
- The response gets sent over BLE using the NUS service, and the phone receives it if subscribed, just like a basic challenge-response interaction.

While the attestation response is still a placeholder and not cryptographically verified yet, the flow from mobile to TF-M and back is functional, laying the groundwork for a complete attestation system.

## 4.1 Conclusion

This project set out to improve BLE security for IoT devices by introducing a lightweight, hardware-backed authentication system using Trusted Firmware-M (TF-M). Instead of just relying on traditional BLE pairing, this setup verifies whether a device is running trusted firmware using secure attestation.

Although it's still a work in progress, the key components are now in place:

- TF-M can be accessed from the main BLE application.
- Secure communication between the mobile and the device is working.

- A flexible messaging system via NUS is set up for easier data exchange.

Once fully integrated, this approach could make BLE-based IoT devices a lot safer, especially in areas like healthcare, smart homes, and industrial control, by preventing unauthorized or tampered devices from joining the network.

## 4.3 Future Work

Looking ahead, there are a few clear next steps to finish and improve the system:

- Merge NUS with the secure BLE project: Right now, the NUS code is separate from the peripheral_sc_only BLE app. The goal is to combine them so we get secure BLE connections *and* support for runtime attestation in one clean setup.
- Build a proper mobile app: Right now, we're using LightBlue for testing, but a custom app would let us automate verification, check signatures, and offer a smoother user experience.

## References

[1] Bluetooth SIG. (2022). *The Bluetooth® Low Energy Primer*. Retrieved from https://www.bluetooth.com/bluetooth-le-primer/

[2] K. Haataja and J. Toivanen, "Two practical man-in-the-middle attacks on Bluetooth secure simple pairing and countermeasures," in Transactions on Wireless Communications, 2010.

[3] Bitdefender. "Bluetooth Low Energy Vulnerability Exposes Millions of Devices to Man-in-the-Middle Attacks." *Hot for Security*, 18 Apr. 2024, www.bitdefender.com/en-us/blog/hotforsecurity/bluetooth-low-energy-vulnerability-exposes-millions-devices-man-middle-attacks

[4] C. Gomez, J. Oller, and J. Paradells, "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology," in Sensors, 2020.

[5] Bluetooth SIG, "Numeric Comparison for Secure Pairing," in Bluetooth Security Whitepaper, 2023.

[6] PSA Certified. (2023). *What is PSA Certified?* Retrieved from https://www.psacertified.org/what-is-psa-certified/

[7] Bluetooth SIG. (2022). *GATT Specification Overview*. Retrieved from https://www.bluetooth.com/specifications/gatt/

[8] Zephyr Project. (n.d.). *peripheral_sc_only: main.c*. GitHub. https://github.com/zephyrproject-rtos/zephyr/blob/main/samples/bluetooth/peripheral_sc_only/src/main.c

[9] Zephyr Project. (n.d.). *peripheral_hr: main.c*. GitHub. https://github.com/zephyrproject-rtos/zephyr/blob/main/samples/bluetooth/peripheral_hr/src/main.c

[10] Zephyr Project. (n.d.). *tfm_ipc: main.c*. GitHub. https://github.com/zephyrproject-rtos/zephyr/blob/main/samples/tfm_integration/tfm_ipc/src/main.c

[11] Zephyr Project. (n.d.). *NUS service: nus.c*. GitHub https://github.com/zephyrproject-rtos/zephyr/blob/main/subsys/bluetooth/services/nus/nus.c

[12] NovelBits, *Bluetooth LE Security: Certificate-Based Authentication Protocol (CBAP)*. Available at: https://novelbits.io/bluetooth-le-security-cbap/. Accessed April 2025.

[13] P. Pandey et al., *An Improved Authentication Scheme for BLE Devices with no I/O Capabilities*, arXiv preprint arXiv:2204.13640, 2022.

[14] Silicon Labs, *Bluetooth Certificates Application Note (AN1396)*, 2021. Available: https://www.silabs.com/documents/public/application-notes/an1396-bluetooth-certificates.pdf

[15] U.S. Patent No. US20190281449A1, *Secure Authentication Over Bluetooth Using Certificates*, 2019.

[16] Bluetooth SIG, *GATT Specification Overview*. Available at: https://www.bluetooth.com/specifications/gatt

[17] Nordic Semiconductor, *Nordic UART Service Documentation*. Available at: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/samples/bluetooth/peripheral_uart/README.html