



**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**

[ ]

**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** 2025

---

**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 web-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. The system successfully generates and transmits real cryptographically-signed Initial Attestation Tokens (IATs) from TF-M. These tokens are 300-500 bytes in size and contain actual device measurements, boot seeds, and cryptographic signatures. The device also implements challenge verification by printing received challenges and identifying sender devices by MAC address for security validation. 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

09/29/2025

---

# TABLE OF CONTENTS

Abstract .....	ii
Acknowledgements .....	iii
Table of Contents .....	iv
Table of Figures .....	v
List of Abbreviations .....	vi
Chapter 1 Introduction .....	1
1.1 Overview .....	1
1.2 Objectives .....	2
1.3 Scope .....	3
1.4 Procedures .....	4
1.5 Expected Project Results.....	5
1.6 Development Tools.....	5
Chapter 2 Literature Review/Theory Background .....	7
2.1 Bluetooth Low Energy (BLE) Pairing and Security .....	7
2.1.1 BLE Pairing Mechanisms and MITM Attack Prevention .....	7
2.1.2 Platform Security Architecture (PSA) Certified API .....	9
2.2 Related Work: Bluetooth Security Beyond Pairing.....	10
2.2.1 Public Key Infrastructur(PKI) in Bluetooth Security .....	11
2.3 Communication Protocols in BLE .....	11
2.3.1 GATT in BLE Projects .....	11
2.3.2 Nordic UART Service(NUS) for Data Exchange.....	12
Chapter 3 Proposed Solution/Project .....	1
3.1 Solution .....	12
3.1.1 Web Client .....	14
3.1.3 Secure World .....	14
3.1.3 Non-Secure World .....	14
3.2 Implementation Plan .....	14
3.3 Project Plan .....	29
Chapter 4 Results and Conclusion .....	30
4.1 Results .....	30
4.2 Conclusion .....	31
4.3 Future work .....	31
References .....	33

---

## **LIST OF FIGURES**

Figure 1.....	13
Figure 2.....	15
Figure 3.1.....	15
Figure 3.2.....	16
Figure 3.3.....	16
Figure 4.1.....	17
Figure 4.2.....	18
Figure 4.3.....	18
Figure 5.1.....	19
Figure 5.2.....	20
Figure 6.1.....	21
Figure 6.2.....	21
Figure 7.1.....	20
Figure 8.1.....	23
Figure 9.1.....	23
Figure 10.1.....	25
Figure 10.2.....	25
Figure 10.3.....	26
Figure 10.4.....	26
Figure 10.5.....	27

---

## LIST OF ABBREVIATIONS

- AES-CCM – Advanced Encryption Standard in Counter with CBC-MAC mode
- API – Application Programming Interface
- BLE – Bluetooth Low Energy
- CBOR – Concise Binary Object Representation
- DIS – Device Information Service
- ECDH – Elliptic Curve Diffie-Hellman
- GATT – Generic Attribute Profile
- IAT – Initial Attestation Token
- IoT – Internet of Things
- IPC – Inter-Process Communication
- MITM – Man-in-the-Middle
- NUS – Nordic UART Service
- PKI – Public Key Infrastructure
- PSA – Platform Security Architecture
- TEE – Trusted Execution Environment
- TF-M – Trusted Firmware-M
- UUID – Universally Unique Identifier
- Zephyr – Zephyr Real-Time Operating System

## 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 BLEbased 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 trustcritical 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 web browser users to seamlessly authenticate BLE-based IoT devices. The framework will build upon standard BLE protocols, allowing users to verify device integrity through a web 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 Nordic UART Service (NUS) to enable flexible bidirectional communication and test interaction between the web 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 real cryptographically-signed attestation tokens within TF-M, including secure random values and device measurements.
- Implementing a complete challenge-response mechanism, where challenges from the web client are processed and responded to with actual attestation data.
- Utilizing challenge verification features including challenge printing and device identification for security validation.
- Preparing the framework for enhanced verification capabilities including hash measurement validation and signature verification.

## **1.4 Procedures**

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

### **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**

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

### **Implementation of the Authentication Mechanism**

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

### **Development of the User-Friendly Framework**

- Designed and developed a web application that allows users to verify BLE IoT devices seamlessly using Web Bluetooth API.
- Implemented cryptographic verification tools within the app to ensure software integrity and authentication.
- Integrated the framework with BLE pairing processes for seamless usability.

### **Testing and Validation**

- Conducted controlled experiments using real-world BLE IoT applications.
- Evaluated the authentication system's effectiveness in detecting and preventing unauthorized access.

- Validated challenge-response mechanisms and attestation token generation.

## **Documentation and Deployment**

- Documented the findings, system architecture, and implementation details.
- Provide guidelines for integrating the authentication framework into different BLE-based IoT applications.
- Explored 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 web application framework that allows web browser 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 and Cursor: 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.

### **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 BLEenabled 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 BLEbased 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). The system successfully generates and transmits real cryptographically-signed Initial Attestation Tokens (IATs) from TF-M. These tokens are 300-500 bytes in size and contain actual device measurements, boot seeds, and cryptographic signatures. The device also implements challenge verification by printing received challenges and identifying sender devices by MAC address for security validation.

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

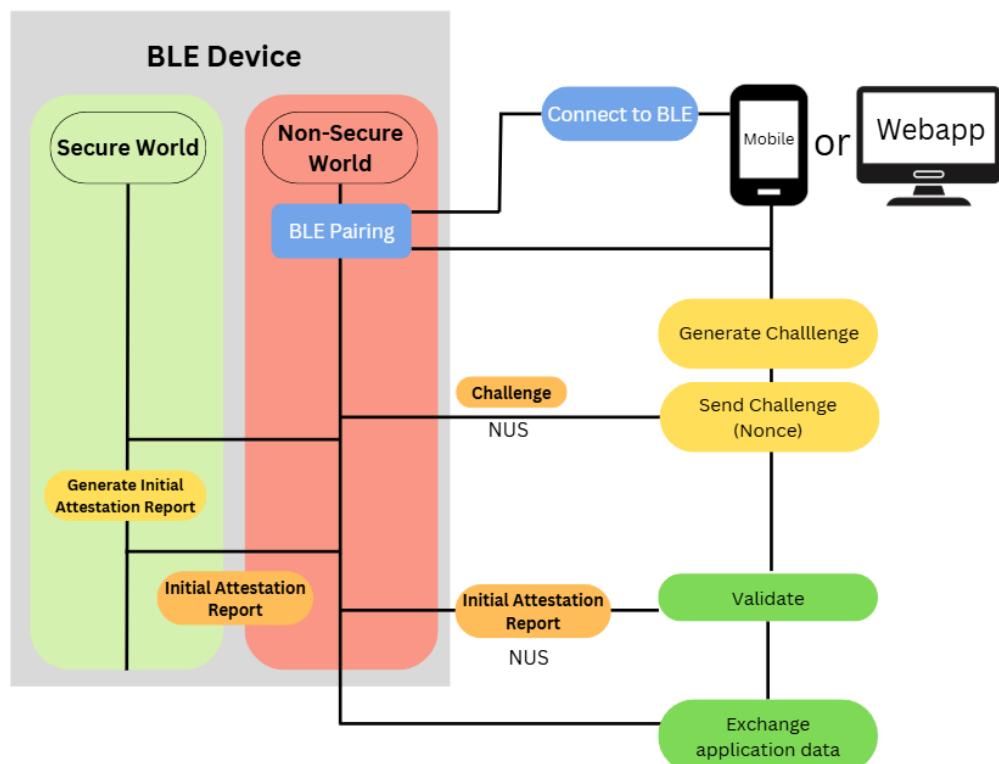


Figure 1 - How the system works: the phone/webapp sends a challenge, TF-M processes it, and the result comes back.

## **3.1 Component Breakdown (based on Figure 1)**

### **3.1.1 Web Client**

The web client acts as the verifier in the authentication process. After establishing a BLE connection via Nordic UART Service (NUS), the web application sends a random challenge (nonce) to the BLE device. The web client receives the complete attestation response containing the full cryptographically-signed token for verification.

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

Receives the challenge from the web client. Passes the challenge to the Secure World using IPC (Inter-Process Communication). Forwards the complete attestation response back to the web client without modification. This layer acts as a secure bridge and cannot generate or fake valid attestations. It also implements challenge verification by printing received challenges and identifying sender devices.

### **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 and generates a cryptographically-signed Initial Attestation Token containing device measurements, implementation IDs, boot seeds, and cryptographic signatures. The complete token is then sent to the non-secure world for transmission to the web client.

## **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 6digit 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

```

132     static void hrs_ntf_changed(bool enabled)
133     {
134         hrf_ntf_enabled = enabled;
135         printk("HRS notification status changed: %s\n", enabled ? "enabled" : "disabled");
136     }
137
138     static struct bt_hrs_cb hrs_cb =
139     {
140         .ntf_changed = hrs_ntf_changed,
141     };
142
143     static void bas_notify(void)
144     {
145         uint8_t battery_level = bt_bas_get_battery_level();
146         battery_level = (battery_level == 0) ? 100U : battery_level - 1;
147         bt_bas_set_battery_level(battery_level);
148     }
149
150     static void hrs_notify(void)
151     {
152         static uint8_t heartrate = 90U;
153         heartrate = (heartrate >= 160U) ? 90U : heartrate + 1;
154         if (hrf_ntf_enabled)
155             bt_hrs_notify(heartrate);
156     }
157

```

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.

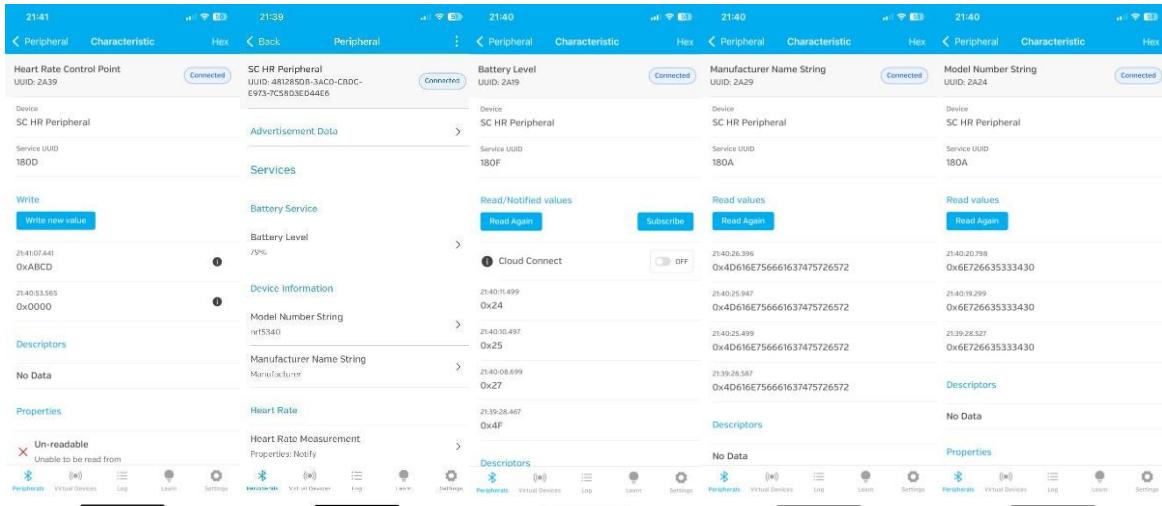


Figure 3.2 Data exchange on a mobile device using the Lightblue App  
Figure 3.2 shows the mobile data exchange functions: read, write, and subscribe. Which, in all read functions, will generate dummy values, and in Write, you can write anything, and it will show on the board's screen as seen in Figure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

```
158     static void tfm_get_version(void)
159     {
160         uint32_t version = psa_framework_version();
161         printk("PSA Framework API Version: %d\n", version);
162     }
163
164 #ifdef TFM_PSA_API
165     static void tfm_get_sid(void)
166     {
167         uint32_t version = psa_version(TFM_CRYPTO_SID);
168         printk("PSA Crypto service minor version: %d\n", version);
169     }
170
171     static void tfm_psa_crypto_rng(void)
172     {
173         psa_status_t status;
174         uint8_t outbuf[16] = {0};
175         status = psa_generate_random(outbuf, sizeof(outbuf));
176         printk("Generated random data:\n");
177         for (int i = 0; i < sizeof(outbuf); i++) {
178             printk("%02X ", outbuf[i]);
179         }
180         printk("\n");
181     }
182 #endif
183
```

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

```

184     int main(void)
185     {
186         int err = bt_enable(NULL);
187         if (err) {
188             printk("Bluetooth init failed (err %d)\n", err);
189             return 0;
190         }
191         att_test();
192         printk("Bluetooth initialized\n");
193         bt_conn_auth_cb_register(&auth_cb_display);
194         bt_conn_auth_info_cb_register(&auth_cb_info);
195         bt_hrs_cb_register(&hrs_cb);
196         start_adv();
197
198         printk("Checking TF-M now...\n");
199         tfm_get_version();
200 #ifdef TFM_PSA_API
201         tfm_get_sid();
202         tfm_psa_crypto_rng();
203 #endif
204
205         while (1) {
206             k_sleep(K_SECONDS(1));
207             hrs_notify();
208             bas_notify();
209         }
210         return 0;
211     }

```

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: ~/zephyrproject/zephyr/samples/bluetooth/peripheral_sc_only

** Booting Zephyr OS build v4.0.0-4463-g491bbcfcbd35 ***
[00:00:00.443,176] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)booting 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: llcr (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...$.X .....
00000000 BB BB BB BB BB BB CC CC CC .....$..;
000000D0 FF FF 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 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 1F 951 A5 01 63 53 50 b1....+...Q..c5P
00000140 45 04 65 30 2E 30 05 58 20 82 A5 B4 43 59 E.e0.0.0.X ...CY
00000150 48 53 04 BF 0F DD 89 A FD A9 06 66 53 48 41 ...t...>...fSHA
00000170 32 35 02 58 20 BE 02 EE 7E 39 3A 35 6A 4A A5 256.X ...-9:5jJ.
00000180 EF 22 AD FA 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 2E 6F 61 6B 6F 61 6B 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 initialized
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: 0x481fd3070b5a6d1824551abdeb691d4
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

```

7  #include <zephyr/kernel.h>
8  #include <zephyr/bluetooth/bluetooth.h>
9  #include <zephyr/bluetooth/services/nus.h>
10
11 #define DEVICE_NAME           CONFIG_BT_DEVICE_NAME
12 #define DEVICE_NAME_LEN        (sizeof(DEVICE_NAME) - 1)
13
14 #define CHUNK_SIZE            16          // Initial received data size (16 bytes)
15 #define DUPLICATED_SIZE       64          // Duplicated data size (64 bytes)
16 #define NOTIF_CHUNK_SIZE      20          // Max notification chunk size
17
18 static const struct bt_data ad[] = {
19     BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_AD_NO_BREDR)),
20     BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
21 };
22
23 static const struct bt_data sd[] = {
24     BT_DATA_BYTES(BT_DATA_UUID128_ALL, BT_UUID_NUS_SRV_VAL),
25 };
26
27 static void notif_enabled(bool enabled, void *ctx)
28 {
29     ARG_UNUSED(ctx);
30
31     printk("%s() - %s\n", __func__, (enabled ? "Enabled" : "Disabled"));
32 }
33

```

Figure 5.1 - Advertising Data Setup and Subscribe Callback

```

34     static void received(struct bt_conn *conn, const void *data, uint16_t len, void *ctx)
35     {
36         uint8_t rx_data[CHUNK_SIZE] = {0}; // Buffer to store received data
37         uint8_t tx_data[DUPLICATED_SIZE] = {0}; // Buffer to store duplicated data
38         int err;
39
40         printk("Received %d bytes from mobile\n", len);
41
42         // Copy received data into rx_data (up to 16 bytes)
43         memcpy(rx_data, data, MIN(len, CHUNK_SIZE));
44
45         // Duplicate the 16 bytes into 64 bytes
46         for (int i = 0; i < 4; i++) {
47             memcpy(&tx_data[i * CHUNK_SIZE], rx_data, CHUNK_SIZE);
48         }
49
50         // Send the duplicated data in chunks of 20 bytes
51         for (int i = 0; i < DUPLICATED_SIZE; i += NOTIF_CHUNK_SIZE) {
52             int chunk_size = MIN(NOTIF_CHUNK_SIZE, DUPLICATED_SIZE - i); // Remaining bytes to send
53             err = bt_nus_send(conn, &tx_data[i], chunk_size); // Send chunk
54             if (err != 0) {
55                 printk("Notification failed for chunk %d (err %d)\n", i / NOTIF_CHUNK_SIZE, err);
56                 break;
57             }
58             printk("Sent chunk %d of %d bytes\n", i / NOTIF_CHUNK_SIZE, chunk_size);
59         }
60     }
61
62     struct bt_nus_cb nus_listener = {
63         .notif_enabled = notif_enabled,
64         .received = received,
65     };
66
67     int main(void)
68     {

```

Figure 5.2 - NUS receives the callback.

```

67     int main(void)
68     {
69         int err;
70
71         printk("Sample - Bluetooth Peripheral NUS\n");
72
73         // Register the NUS callback
74         err = bt_nus_cb_register(&nus_listener, NULL);
75         if (err) {
76             printk("Failed to register NUS callback: %d\n", err);
77             return err;
78         }
79
80         // Enable Bluetooth
81         err = bt_enable(NULL);
82         if (err) {
83             printk("Failed to enable bluetooth: %d\n", err);
84             return err;
85         }
86
87         // Start advertising with NUS service
88         err = bt_le_adv_start(BT_LE_ADV_CONN_FAST_1, ad, ARRAY_SIZE(ad), sd, ARRAY_SIZE(sd));
89         if (err) {
90             printk("Failed to start advertising: %d\n", err);
91             return err;
92         }
93
94         printk("Initialization complete\n");
95
96         while (true) {
97             k_sleep(K_SECONDS(3));
98         }
99
100        return 0;
101    }

```

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

We then removed the Heart Rate Service and then fully ported the NUS functionality into our main project.

```
0
1
2     static struct bt_nus_cb nus_cb = {
3         .notif_enabled = nus_notif_enabled,
4         .received = nus_received,
5     };
6
```

Figure 6.1 - NUS callback implementation

```
312     // Register NUS callbacks
313     err = bt_nus_cb_register(&nus_cb, NULL);
314     if (err) {
315         printk("Failed to register NUS callback: %d\n", err);
316         return err;
317     }
318
319     bt_conn_auth_cb_register(&auth_cb_display);
320     bt_conn_auth_info_cb_register(&auth_cb_info);
321     start_adv();
322
323     printk("Checking TF-M now...\n");
324     tfm_get_version();
325 #ifdef TFM_PSA_API
326     tfm_get_sid();
327     tfm_psa_crypto_rng();
328 #endif
329
330     while (1) {
331         k_sleep(K_SECONDS(1));
332         // NUS handles notifications via callbacks, no periodic notify needed
333     }
334
335 }
```

Figure 6.2 - NUS callback implementation showing challenge reception and response handling

We then implemented the `psa_initial_attest_get_token()` function call to replace the dummy response mechanism. This function takes the 32-byte challenge received from the web client and generates a real cryptographically-signed attestation token from the TF-M secure world, containing actual device measurements and integrity data.

```

#ifndef TFM_PSA_API
    // Get Initial Attestation Token from TF-M
    printk("Requesting attestation token from TF-M...\n");
    status = psa_initial_attest_get_token(
        challenge, CHALLENGE_SIZE,           // Challenge input
        token_buf, TOKEN_BUF_SIZE,           // Token output buffer
        &token_size);                      // Actual token size

    if (status != PSA_SUCCESS) {
        printk("ERROR: Attestation failed with status: %d\n", status);
        // Send error response
        const char *error_msg = "ATTESTATION_ERROR";
        if (nus_ntf_enabled) {
            bt_nus_send(conn, error_msg, strlen(error_msg));
        }
        return;
    }

    printk("SUCCESS: Got attestation token of %d bytes\n", token_size);

    // Hash the attestation token using PSA Crypto
    uint8_t token_hash[32] = {0}; // SHA-256 hash
    size_t hash_length = 0;

    status = psa_hash_compute(PSA_ALG_SHA_256,
        token_buf, token_size,
        token_hash, sizeof(token_hash),
        &hash_length);

    if (status != PSA_SUCCESS) {
        printk("ERROR: Failed to hash token, status: %d\n", status);
        return;
    }

    printk("\n==== TOKEN HASH (TRIMMED FOR DEVICE DISPLAY) ====\n");
    printk("SHA-256 Hash: ");
    for (int i = 0; i < 32; i++) {
        printk("%02X", token_hash[i]);
        if (i % 8 == 7) printk(" ");
    }
    printk("\n");
    printk("=====\\n");

```

Figure 7.1 - TF-M attestation token generation with 32-byte challenge input

After successfully generating a 508-byte attestation token, we faced a challenge: displaying this large amount of data on the device's serial console would be overwhelming and hard to verify at a glance. We needed a way to provide a compact representation for debugging while still sending the complete token to the web app.

We integrated PSA Crypto's hashing function to compute a SHA-256 hash of the full attestation token. This creates a 32-byte fingerprint that uniquely identifies the token. The implementation uses `psa_hash_compute()` with `PSA_ALG_SHA_256` algorithm, taking the full token buffer as input and producing a fixed-size hash output.

```

198     // Hash the attestation token using PSA Crypto
199     uint8_t token_hash[32] = {0}; // SHA-256 hash
200     size_t hash_length = 0;
201
202     status = psa_hash_compute(PSA_ALG_SHA_256,
203                               token_buf, token_size,
204                               token_hash, sizeof(token_hash),
205                               &hash_length);
206
207     if (status != PSA_SUCCESS) {
208         printk("ERROR: Failed to hash token, status: %d\n", status);
209         return;
210     }
211
212     printk("\n==== TOKEN HASH (TRIMMED FOR DEVICE DISPLAY) ====\n");
213     printk("SHA-256 Hash: ");
214     for (int i = 0; i < 32; i++) {
215         printk("%02X", token_hash[i]);
216         if (i % 8 == 7) printk(" ");
217     }
218     printk("\n");
219     printk("=====\\n");

```

Figure 8.1 - SHA-256 hash calculation of 508-byte attestation token for device console display

After implementing real attestation tokens of 508 bytes, we needed an efficient way to transmit this payload over BLE, which has a maximum notification size of 20 bytes. We developed a chunked transmission approach that divides the 508-byte token into 26 packets (25 packets of 20 bytes plus one 8-byte packet). The system sends the complete attestation token to the web app via BLE notifications, while simultaneously calculating and displaying a SHA-256 hash of the token on the serial console for developer verification. A 10ms delay between chunks prevents BLE stack overflow.

```

241     // Send the FULL attestation token (300-500+ bytes for real, 256 for dummy)
242     printk("Sending FULL attestation token (%d bytes) in chunks...\n", token_size);
243     int total_sent = 0;
244     for (int i = 0; i < token_size; i += NOTIF_CHUNK_SIZE) {
245         int chunk_size = MIN(NOTIF_CHUNK_SIZE, token_size - i);
246         err = bt_nus_send(conn, &token_buf[i], chunk_size);
247         if (err != 0) {
248             printk("ERROR: Failed to send chunk %d (err %d)\n", i / NOTIF_CHUNK_SIZE, err);
249             break;
250         }
251         total_sent += chunk_size;
252         if (i < 60 || i >= token_size - 20) { // Only print first 3 and last chunk
253             printk("Sent chunk %d: %d bytes (total: %d/%d)\n",
254                   i / NOTIF_CHUNK_SIZE, chunk_size, total_sent, token_size);
255         }
256
257         // Small delay between chunks to avoid overwhelming BLE stack
258         k_msleep(10);
259     }

```

Figure 9.1 - Chunked BLE transmission sending full 508-byte token in 26 packets

During testing, we discovered that challenges were getting lost in the serial output due to message drops when too much data was printed at once. We needed a reliable way to verify that the correct challenge was received from the correct device without losing this critical information in the output stream. We modified the challenge handling to store the received challenge data at the beginning of processing, then print it at the very end after the "ATTESTATION COMPLETE" message. This ensures the challenge information appears when the serial output has settled down and is less likely to be dropped. The challenge is displayed in hex format along with the sender's MAC address. Please refer to Figure 10.5 to see the process.

Initially, we were testing with the LightBlue app, which required manual hex data entry and didn't provide a good user experience. We needed a custom web application that could automatically handle challenge generation, response parsing, and verification. The shift from mobile app to web app was made to leverage Web Bluetooth API for easier deployment and testing.

So, we created a complete web application that uses the Web Bluetooth API to connect to the Nordic UART Service. The app provides a user-friendly and very simple interface for connecting to the device, handling the 6-digit passkey authentication, sending challenges, and receiving attestation responses. It automatically reconstructs the chunked responses and display both the token and its hash.

The application consists of:

- index.html - Main HTML structure with BLE connection interface and NUS serviceUUIDs
- script.js - JavaScript implementation for Web Bluetooth API, challenge handling, and attestation response processing
- styles.css - User interface styling for a professional presentation

With all components integrated, we needed to validate the entire challenge-response attestation flow from end to end. This meant verifying that challenges sent from the web app were correctly received, processed through TF-M, and resulted in valid attestation tokens being returned.

We conducted comprehensive testing by sending various challenges through the web app and monitoring the complete flow on the device console. The

testing confirmed that each unique challenge produces a different attestation token (different hash), the challenge is correctly included as the nonce in the token, and all 508 bytes are successfully transmitted and reconstructed. The following figures show the entire procedure.

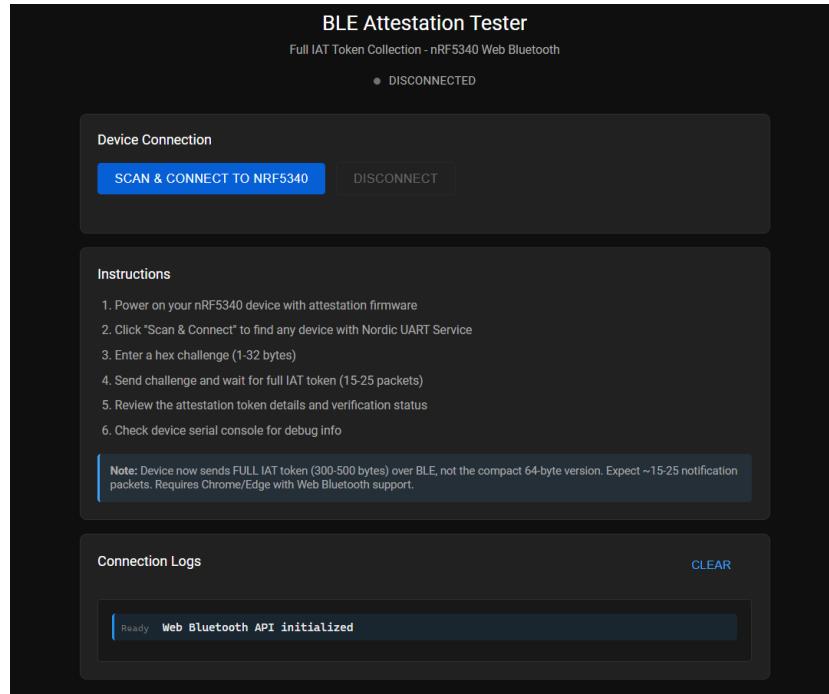


Figure 10.1 – The BLE Webapp landing page

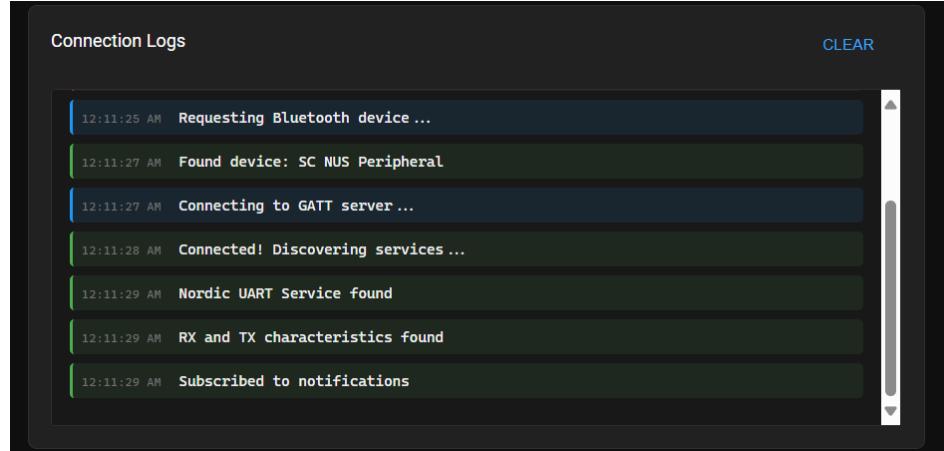
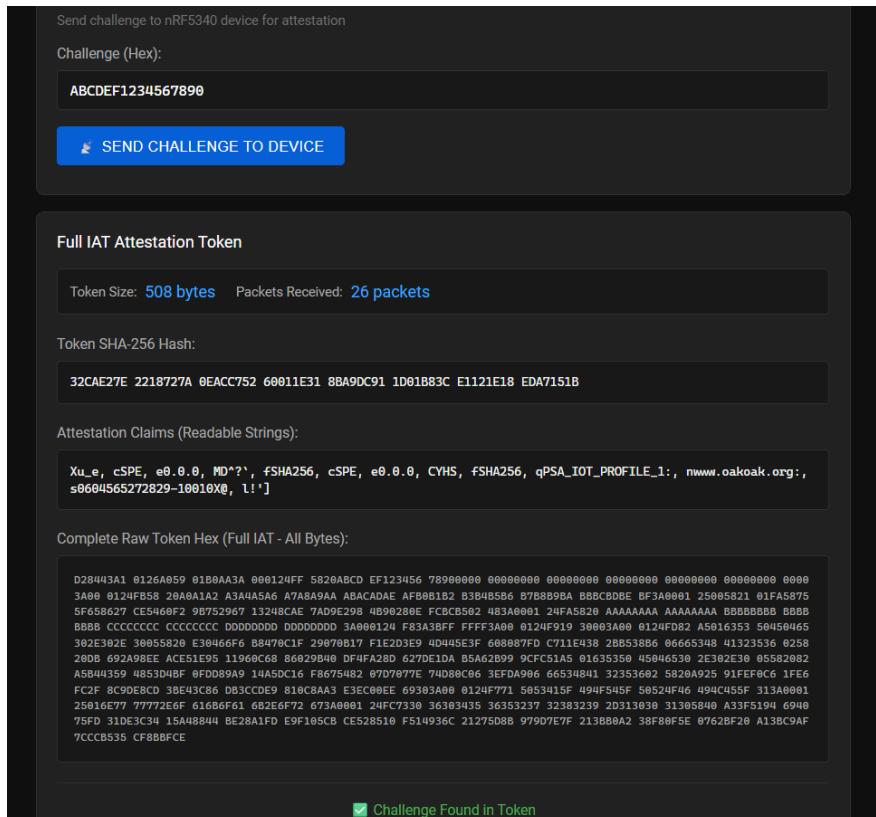


Figure 10.2 – The Connection logs from the webapp while initiating the connection and verifying the 6-digit passkey



---

Figure 10.3 - Web Application Interface Showing Complete Attestation Flow

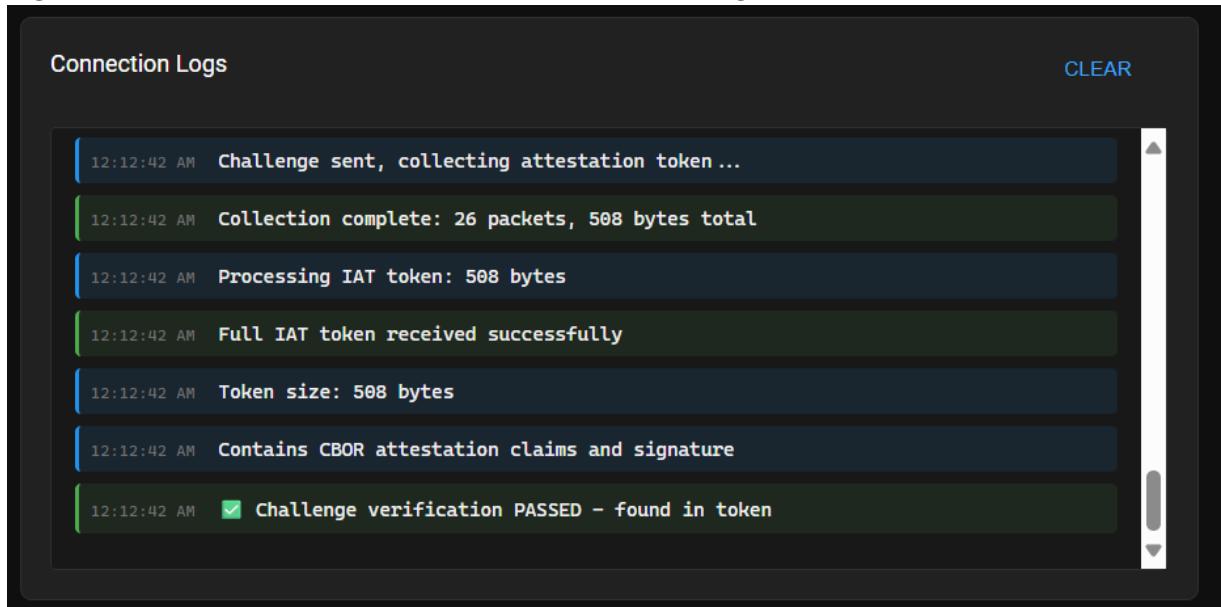


Figure 10.4 – Connection Logs after the challenge verification was successful

```

00000050 EE FF 3A 00 01 24 FB 58 20 A0 A1 A2 A3 A4 A5 A6 ...$.X .....
00000060 A7 A8 A9 AABB BC BD BE BF 3A 00 01 25 00 58 21 .....%.X!
00000080 01 FA 58 75 5F 65 86 27 CE 54 60 F2 9B 75 29 67 ..Xu_e.'T'..u)g
00000090 13 2 AA AA AA AA H:..$.X .....
000000B0 BB BB BB BB BB CC CC CC CC CC CC CC CC .....
000000C0 DD DD DD DD DD DD 3A 24 FD 82 ...$.0...$..
000000E0 A5 01 63 53 50 45 04 65 30 2E 30 05 58 20 ..cSPE.e0.0.0.X
000000F0 E3 04 66 F6 B8 47 0C 1F 29 07 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 95 DF 4F A2 B0 .....h...@.0..
00000130 62 7D E1 DA B5 A6 2B 99 9C FC 51 A5 01 63 53 50 b}....+...Q..cSP
00000140 45 04 65 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 A9 14 A5 DC 16 F8 67 54 82 HS.....gT.
00000160 07 C6 1F E6 FC 2F 256.X %. .....
00000180 8C 9D E8 CD 3B E4 3C 86 DB 3C CD E9 81 0C 8A A3 ....;<.<.....
00000190 E3 EC 00 EE 69 30 3A 00 0qPSA_
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 %.n0001C0 67 3A 00 01 24 FC 73 30 36 30 34 35 36 35 32 37 g:..$.s06
000001D0 32 38 32 39 2D 31 30 30 31 30 58 40 01 DB 64 C8 2829-10010X@
000001F0 99 2A D8 C7 07 B2 21 8A 43 2C 11 69 2A 08 50 C
00000200 C0 7B 68 09 72 03 07 D8 8D 5E D5 BD 53 6A B5 1C .{h.r.... L.g..T...{n?

Bluetooth initializeddddddd
Advertising successfully started
Checking TF-M now...
PSA Framework API Version: 257
PSA Crypto service minor vconnected to 1C:CE:51:C6:10:12 (public)
Passkey for 1C:CE:51:C6:10:12 (public): 751485
[00:00:57.822,937] <inf> bt_smp: DHKey generation sNUS notification status changed: enabled
[00:01:02.413,116] <wrn> bt_conn: Timeout discarded. No blocking in syswq.
[00:01:22.679,534] <inf> bt_smp: LTK Stored: 0xca4d9a29cc41faadafbc2c388807cdd
Security changed: 1C:CE:51:C6:10:12 (public) level 4
PTA--- 11 messages dropped ---
0EAC75260011E31 BBA9DC911D01B83C E1121E18EDA7151B
=====tes (total: 40/508)
Sent chunk 2: 20 bytes (total: 60/508)
Sent chunk 25: 8 bytes (total: 508/508)
==== ATTESTATION COMPLETE ====
SENT TO DEVICE: 1C:CE:51:C6:10:12 (public)
FULL IAT SIZE SENT: 508 bytes=====
*** CHALLENGE FROM 1C:CE:51:C6:10:12 (public) ***
Raw challenge (8 bytes): ABCDEF1234567890

```

Figure 10.5 – The entire process from the console in the device side

### **3.3 Project Plan**

**Table 3-1.** Project plan 2025-2026 Academic year.

TFA Integration & Testing																				
CBOR Token Validation																				
Security Edge Case Testing																				

### **4.1 Results**

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

- BLE pairing works using the peripheral\_sc\_only example with 6-digit passkey authentication, demonstrating functional secure connections.
- A fully functional attestation system has been built using TF-M that generates real cryptographically-signed Initial Attestation Tokens containing device measurements, implementation IDs, boot seeds, and cryptographic signatures.
- Communication between the non-secure and secure world (TF-M) is operational through Inter-Process Communication (IPC), enabling secure attestation token generation.
- The system successfully generates 508-byte Initial Attestation Tokens and transmits them completely to the web application via BLE notifications (26 packets). For developer convenience, the device also calculates and displays a SHA-256 hash of the token on the serial console, providing both complete security verification for the web application and compact debugging information for developers.
- Challenge verification has been implemented with the device printing received challenges and identifying sender devices by MAC address for security validation.
- The web application has been developed and successfully tested using Web Bluetooth API, providing challenge sending and attestation verification capabilities. The system generates real PSA Initial Attestation Tokens that can prove device integrity to remote verifiers, providing a foundation for trustworthy IoT device authentication

## 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 relying solely on traditional BLE pairing, this setup verifies whether a device is running trusted firmware using secure attestation.

The key components are now fully operational:

- TF-M can be accessed from the main BLE application.
- Secure communication between the web application and the device is working.
- Real attestation token generation with cryptographic signatures.
- Challenge verification and device identification capabilities.
- A flexible messaging system via NUS for efficient data exchange The system generates real PSA Initial Attestation Tokens that can prove device integrity to remote verifiers

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 next steps we are looking to to finish and improve the system:

- Complete the verification workflow using tf-m-tools for CBOR token parsing.
- Implement hash measurement verification to detect firmware tampering.
- Add cryptographic signature validation using device public key
- Ensure rejection of tokens with correct signatures but wrong measurements.
- Ensure rejection of tokens with correct measurements but wrong signatures
- Ensure rejection of tokens with correct hash and key but wrong signature

- Implement proper random challenge generation to prevent replay attacks
- Develop the web application interface to resemble a production-ready application rather than a technical demonstration tool
- Create test scenarios covering both compromise detection cases (when attestation should fail) and legitimate access cases (when attestation should succeed)
- Implement comparative performance testing by creating two versions of the web application: one with the TF-M attestation solution and one without. And conduct performance measurements between the two and compare them
- Determine how many additional bytes are required for the board implementation and investigate this further

## 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-energyvulnerability-exposes-millions-devices-man-middle-attacks](https://www.bitdefender.com/en-us/blog/hotforsecurity/bluetooth-low-energyvulnerability-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](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](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](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-securitycbap/>. 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/applicationnotes/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](https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/samples/bluetooth/peripheral_uart/README.html)