Design of an encryption protocol for BLE advertisements.

Ioana Teaca

Abstract

Bluetooth Low Energy is becoming one of the standard ways of communication in IoT devices. BLE advertising mode is commonly used in beacons to broadcast location services with the data being generally transmitted "in the clear", in a one-way communication. The following paper aims to explore ways of implementing encryption in a system where beacons advertise sensitive sensor data. A version of AES encryption that can be used to encrypt very small packets is described, and several security aspects are taken into consideration.

1 Introduction

BlueTooth Low Energy is a popular protocol usually used in low-power embedded devices. Due to the expected low computation capabilities in target devices, compromises are often made in order to simplify the protocol. BLE is also generally used in applications where sensitive information is being transmitted, such as fitness and medical devices or smartphones. As a consequence, BLE manufacturers have to take into consideration the security features and limitations of this technology[1].

The Generic Access Profile is a layer in the protocol stack that defines the access modes and communication procedures of BLE devices. GAP defines two ways these devices can communicate. The first is through an encrypted connection. When devices need to connect in order to transmit sensitive data they go through a pairing process to encrypt the link between them. A short term key is generated on both sides and the devices form a secure connection for the duration of the communication. This is a complex process that is highly dependent on the I/O capabilities and constraints of each device. The second way of sending data is by using advertisements, where a peripheral device broadcasts packets to all other devices. The listening devices can read this data or request to form a connection in order to receive more information [2].

This paper describes a system composed of several dozen peripheral devices that use advertisements to transmit sensor data. A central device acts as a MQTT gateway that reads the data and publishes it to a dashboard. BLE advertisements are used for communication due to the scalability, simplicity and ease of broadcasting sensor data without the need for a connection. The disadvantage built into using advertisements is that the data is transmitted with no encryption or protection. This paper describes a protocol for encrypting this communication and adding several protections without a severe overhead on the process. The aim of the research is to explore whether any of the classical AES encryption modes offered by the device's firmware can be used to effectively and safely encrypt the traffic. Due to the very small size of the packet a secondary goal is to analyse what protections can be included in the packet to prevent common attacks.

In the following section additional background is given on BLE advertising mode, and the nodes-gateway system is described, along with security implications. Section 3 describes the encryption possibilities offered by the firmware. Section 4 discusses related work that was relevant for the choices and design of the protocol. In the "Implementation", the solution and the requirements that motivated it are explained. In section 6 the protocol is evaluated and a risk analysis is performed. Finally, the paper concludes on features that could be added to improve safety or minimize overhead.

2 Background

2.1 BLE Advertising Mode

The Bluetooth Low Energy standard defines two types of Radio Frequency channels: data channels and advertising channels. Data channels are used for bidirectional communication between connected devices while advertising channels are mainly used for device discovery, connection establishment and broadcast transmission[3]. When a device is in advertising mode, it broadcasts data at regular intervals of time called advertising events. Within one event, the sender transmits a packet using each of the three defined advertising channels. Generic Access Profile (GAP) specifies four roles for devices implementing BLE: Central-Peripheral or Broadcaster-Observer[4].

The first two roles are connection-based, with a Central device tasked with initiating and managing connections with Peripheral devices. In order for a connection to be formed, the "Peripheral" device announces that it is a connectable device while the "Central" device scans for such advertisements. Once it finds one, it initiates a connection by sending a Connection Request message. The devices then go through a process of authentication and the temporary connection is encrypted with a short term key. This process is called "Pairing" and can be extended to a "Bond" by exchanging long-term keys[2].

When a device simply broadcasts data on the advertising channels without supporting connections to other devices, it is called a "Broadcaster". The complementary role to the "Broadcaster" is the "Observer", whose role is to receive the data transmitted by the Broadcaster[5]. In the following sections, the devices described in the system will only assume the roles of Broadcaster and Observer.

A BLE Broadcasting packet has the following structure according to the Bluetooth Core specification [6]:

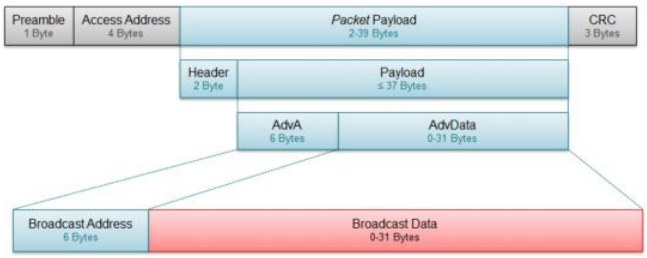

Figure 1: BLE Broadcasting Packet Format [6]

- 1 byte preamble with value 0xAA used for synchronization and timing estimation at the receiver.
- 4 byte access address with a value of 0x8E89BED6.
- a variable length packet payload consisting of a header that describes the type of packet and a payload, which includes the advertiser's address and advertised data
- 3 byte Cyclic Redundancy Check (CRC): used to check packet integrity

The packet can be used to advertise 31 bytes of user specified data. There are a number of Advertisement Data Types that can be specified, each with a different standard. Some of the common data types are [6]:

AD Data Type	Data Type Value	Description
Flags	0x01	Device discovery capabilities
Service UUID	0x02 - 0x07	Device GATT services
Local Name	0x08 - 0x09	Device name
TX Power Level	0x0A	Device output power
Manufacturer Specific Data	0xFF	User defined

Figure 2: Advertisement Data Types [6]

The Manufacturer Specific Data type indicated with 0xFF is of particular interest as it allows the flexibility of defining a custom payload. Beacons are a class of non-connectable devices that take advantage of this flexibility to define customized data structures.

Although there are 31 bytes in the Advertising data packet, only 22 are usable in practice, as can be seen in Figure 4:

- Flags(3): 3 bytes are reserved to encode information about the device's connection capabilities
- Manufacturer specific(4): the Length and Type occupy 2 bytes, with Company ID using two more bytes.
- Device name(2): takes a variable amount of bytes depending on whether it is shortened or not. It is possible to save space by broadcasting an empty name, but two bytes will still be used for Length and Type.

Beacons are BLE devices that function almost exclusively in advertising mode although connection-based operations may be needed for configuration purposes. Because they do not accept connection requests from clients, they consume very little power[7]. In addition, they are low-cost and easily deployed through a service area. They generally broadcast unencrypted basic identification and location data, but the customizable payload can potentially include sensor data. However, this is generally not the main purpose of Beacons, due to the small size of the payload and the unencrypted nature of broadcasting.

2.2 Thingy:52 and Espruino Setup

Thingy:52 is a microcontroller development board based on the Nordic Bluetooth Low Energy chip, nRF52832 and is used as a platform to prototype IoT solutions. It has an array of sensors that gather data about temperature, air quality, noise levels, color, as well as motion sensors. nRF52832 is an ultra low power microcontroller that allows devices to be battery efficient and broadcast for long periods of time [8].

Espruino is a JavaScript interpreter for microcontrollers used to ease development of embedded software. It exposes its entire API in its JavaScript environment and comes with a suite of tools that make uploading and executing code effortless on development boards such as nRF52832[9].

In this project the devices used as beacons are powered by the nRF52832 microcontroller using version 2v03 of the Espruino firmware. The entire system is composed of a few dozen Thingy:52 devices that collect environment sensor data that can be grouped as follows:

- air quality readings: CO2 levels and TVOC (Total Volatile Organic Compounds) - 4 bytes
- atmospheric data: temperature, humidity, pressure - 6 bytes
- color readings: red, green, blue and clear - 8 bytes
- acceleration measurements: x, y, z - 6 bytes
- battery and noise levels - 4 bytes

A pseudocode of the Espruino script running on the Thingy:52 devices is shown below:

```
function updateAdvertisingBuffer(sensorData, tag)
{
    new buffer[16];
    buffer.setUint16(0, tag)

    if (tag == 1) {
        buffer.setUint16(2, sensorData[TVOC]]
        buffer.setUint16(4, sensorData[eCO2]]
        buffer.setUint16(6, sensorData[press]);
        buffer.setUint16(8, sensorData[temperature]);
```

```
        buffer.setUint16(10, sensorData[humidity]);
        buffer.setUint16(12, sensorData[battery]);
        buffer.setUint16(14, sensorData[noise]);
    } else if (tag == 2) {
        buffer.setUint16(2, sensorData[color_r]]
        buffer.setUint16(4, sensorData[color_g]]
        buffer.setUint16(6, sensorData[color_b]);
        buffer.setUint16(8, sensorData[color_c]);
        buffer.setUint16(10, sensorData[acc_x]);
        buffer.setUint16(12, sensorData[acc_y]);
        buffer.setUint16(14, sensorData[acc_z]);
    }

    return advertisingBuffer;
}

function advertise(airQuality, atmosphere, colors, acceleration, noiseAndBattery) {

    if (firstAdvertisingRound){
        advertisingBuffer = updateAdvertisingBuffer([airQuality, atmosphere,
    noiseAndBattery], 1); // total 14 bytes
        NRF.setAdvertising({}, {manufacturer: 0x590, manufacturerData: advertisingBuffer
    });
    }
    else if (secondAdvertisingRound){
        updateAdvertisingBuffer([colors, acceleration], 2); // total 14 bytes
        NRF.setAdvertising({}, {manufacturer: 0x590, manufacturerData: advertisingBuffer
    });
    }
}

function startSensing() {
    while (device is on){
        airQuality = getAirQuality(); // CO2 and TVOC levels - 4 bytes
        atmosphere = getAtmospericData(); // temperature, humidity, pressure - 6 bytes
        colors = getColorReadings(); // r, g, b, c - 8 bytes
        acceleration = getAccelerationMeasurements(); // x, y, z - 6 bytes
        noiseAndBattery = getNoiseAndBatteryLevels(); - 4 bytes
        advertise(airQuality, atmosphere, colors, acceleration, noiseAndBattery);
    }
}


function onInit() {
    startSensing();
}
```

Once the device boots, it starts collecting environmental data using its sensor. The data is then broadcast in two advertising rounds of 16 bytes each: 14 bytes of sensor data and a 2 byte tag that specifies the type of data it contains to the Observer. The advertisements are split in two rounds because air quality, atmosphere, noise and battery levels need to be broadcast more often than color and acceleration data. This size fits comfortably in the suggested maximum 22 bytes that can be advertised as manufacturerData using the Espruino firmware[10].

The Observer is a script running on a RaspberryPi that has BLE capabilities. It keeps a white-list of addresses for all known "Thingy" devices. It scans for advertisements broadcast by known devices and functions as a MQTT gateway to a dashboard where the sensor data is stored and published.

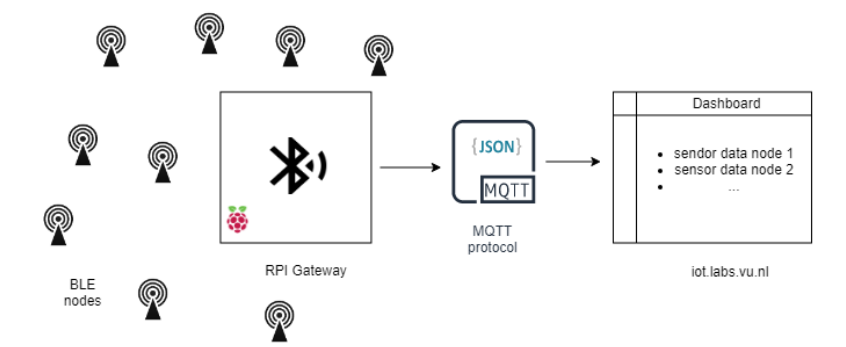

Figure 3: System diagram

2.3 Security concerns

All communication is unencrypted and the BLE advertised data can be easily captured without the need for specialized hardware. Figure 4 shows a WireShark example of a capture done with a smartphone running in Developer Mode[11].

The manufacturer specific data is advertised under the 0x090 manufacturer ID and with the shortened device name "Thin". This makes it easy for an external observer to fingerprint the device and deduce that this is sensor data coming from a Thingy:52 running Espruino firmware. Due to the repetitive nature of sensor data, it is possible to infer what kind of information is being broadcast, especially because acceleration data and color data are bundled together. Of particular concern are CO2 and noise levels, which directly reflect how many people are in a room, for example. If the devices are attached to a door, the acceleration data can also reflect when a door opens or closes.

In addition to sniffing, there are more sophisticated attacks that can be done using devices that are able to act as BLE receivers, as well as transmitters. Ubertooth and nRF52-DK are two of the most popular hardware tools of choice. Using such programmable devices allows for a wealth of possible attacks:

- Spoofing: The Observer keeps a list of white-listed devices based on their Bluetooth Device Address (or BD_ADDR). A message can be spoofed to appear as if it is coming from a white-listed address, and the Observer will read it and publish bad data to the dashboard.
- Replaying: A message can be captured and replayed at a later time. The Observer will read and publish out-of-date sensor measurements. Note that simple encryption will not prevent this attack. [1]
- Man in the middle: One of the advertised messages is intercepted and modified on-the-fly. Alternatively, a copy of a message is transmitted with some modifications. [1]

Although using advertisements to broadcast sensor data is a simple and efficient solution, it is clear that this is sensitive data that should not be sent unencrypted.

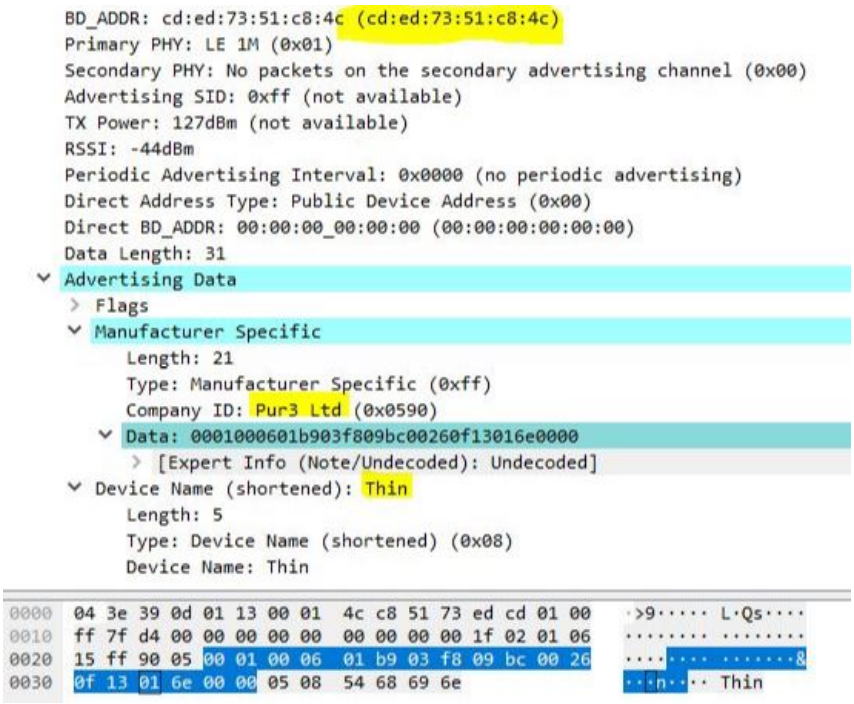

Figure 4: Unencrypted BLE Capture

3 AES Modes

To decide on the most suitable encryption mode the following requirements are taken into consideration:

1. The sensors are broadcasting sensitive data "in the clear"; encryption needs to be added while keeping the overhead low .
2. The advertising packet has a maximum size of 22 bytes; this needs to fit sufficient sensor data and any necessary encryption mechanisms and protections against tampering.
3. The size of the "plaintext" to be encrypted - the sensor data - doesn't exceed 16 bytes. Splitting it into consecutive blocks is not necessary; stream ciphers are preferable to block ciphers.
4. Due to the small size of the packet, encryption modes that require padding, such as CBC and ECB, are not desirable, as they take up valuable packet space.
5. Packets are not guaranteed to be delivered in order and they can drop.
6. Implementing persistent storage on the beacons is a complication that should be avoided if possible.
7. The encryption keys are preshared; every device has its own unique key, while the Observer stores the corresponding key for each beacon. When the keys need to be renewed, an encrypted connection between the beacon and the central device is made and the key is written securely
8. A message integrity mechanism is required to prevent wrong data from being accepted by the Observer.
9. Some of the sensor data needs to be advertised as a bundle, such as color and acceleration data.
10. Implementation needs to be reasonably simple and efficient; the battery consumption needs to be sufficiently low.

The Espruino crypto module implicitely uses mbedTLS which is a C implementation of SSL and TSL protocols and their respective cryptographic algorithms. Only some of the utility is exposed through the firmware API:

- crypto.AES with modes CBC, CFB, CTR, OFB, ECB
- crypto.PBKDF2 which is a password-based key derivation algorithm, using SHA512
- crypto.SHA with modes SHA1, SHA224, SHA384, SHA512

The Advanced Encryption Standard (AES) is one of the standard encryption algorithms and is considered secure when implemented correctly. AES is a block cipher capable of handling blocks of 128 bits using encryption keys of varying lengths: 128, 192, and 256 bits. Encryption is done in several rounds, each of them consisting of byte substitution, permutation, arithmetic operations and an XOR operation with the encryption key [12].

AES has multiple modes of operation but the scope of this project is restricted to only AES modes offered by the firmware. AES is a symmetrical encryption algorithm where the same key is used to encrypt and decrypt the message[12]. Each device starts with its own preshared encryption key, while the Observer keeps track internally of the key corresponding to each device.

3.1 ECB

Electronic Code Book is a simple encryption scheme where the plaintext is divided into blocks and each block is encrypted with the same key. The biggest disadvantage of this mode is the lack of diffusion: the same plain text value will always encrypt to the same cipher value. This mode is generally considered weak[13], but is particularly inappropriate when the plaintext has repeating values, as is the case for advertised packets. The packets always begin with a header that has value 1 or 2, to signify the packet type. In addition, sensor data generally has little variance from packet to packet. The patterns in the plaintext will emerge, making it an excellent "launchpad" for codebook attacks[13].

3.2 CBC

Cipher Block Chaining solves the problem of ECB: every encryption of the same plain text leads to a different cipher text. Each block of plaintext is XOR-ed with the previous cipher block before going through encryption. This way, a one bit change in the initial plaintext will cascade down to all following blocks. The first block is XORed with an Initialization Vector of the same size as the block size(16 bytes)[14]. This mode is considered secure and is the most commonly used mode of operation. In spite of its advantages, it has disadvantages that make it unusable for encrypting advertising data. For a receiver to be able to decrypt the message it needs to know the Initialization Vector used for encryption. One of the requirements of CBC is that the block size is to be padded to 16 bytes[12]. As shown in Fig. 6 padding the sensor data to 16 bytes means only 6 bytes will be left free in the packet.

This means only 6 bytes of the IV can possibly fit in the leftover space. If the IV were to be generated implicitly on both sides from some seed this would go against one of the crucial requirements of CBC: the IV must be perfectly unpredictable. Using a seed would make the IV predictable, and the encryption would become vulnerable to plain-text attacks. Because of the small size of the advertising packet, the 16 bytes cipher text concatenated with the 16 byte IV cannot possibly fit into the available 22 bytes.

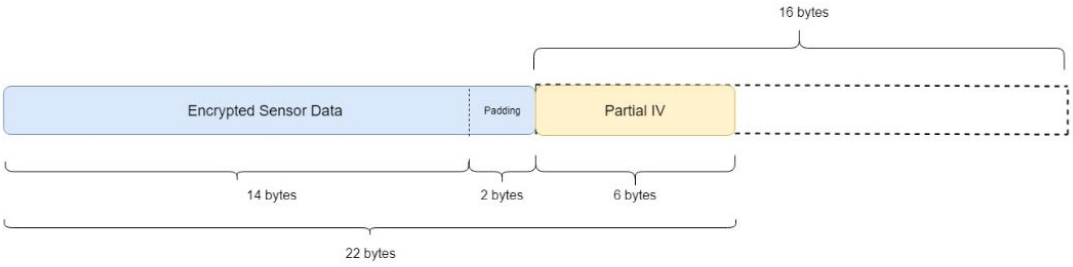

Figure 5: CBC packet structure

3.3 CFB

The Cipher Feedback (CFB) mode is a close relative of CBC with a similar mode of operation. In particular, it turns a block cipher into a self-synchronizing stream cipher, which makes it possible to recover if part of the ciphertext is lost due to transmission errors [15].

Similarly to CBC, at the beginning of the encryption process a random 16 byte IV is generated, which needs to be sent explicitely along with the ciphertext[14]. As opposed to CBC, the initial plaintext does not need to be padded to 16 bytes. This is an attractive quality when working with small sized packets: 22 bytes is enough to send the IV with every packet along with 6 bytes of encrypted sensor data. Although it is secure enough, sending 6 bytes at a time is not very efficient and leaves no room for additional protections to be included in the packet. For example, because of CFB's malleability[13], an HMAC would need to be included in the packet to assure message integrity and authentication. Suppose a SHA1 was used; its 20 byte output will not be able to fit in the packet.

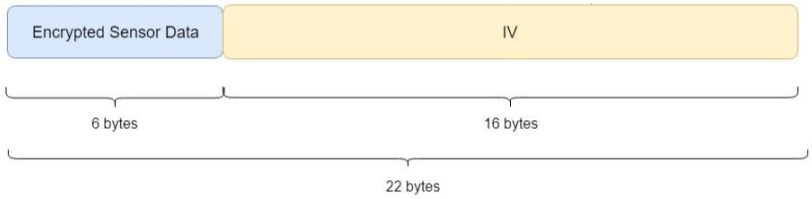

Figure 6: CFB packet structure

3.3.1 OFB

Output Feedback (OFB) turns a block cipher into a synchronous stream cipher. The algorithm uses a block cipher to generate a keystream that is XORed with the plaintext to obtain the final ciphertext. Each block of the OFB keystream depends on all previous keystream blocks, therefore cannot be performed in parallel[14].

Similarly to CFB, the plaintext does not need to be padded to a specific block size, as it is simply XORed with the generated keystream. Unlike CBC and CFB, the 16 byte IV used in generating the first keystream does not need to be unpredictable, it only needs to be unique for each execution of the encryption process[14]. According to NIST specification, the IV may be a counter[14].

Suppose the advertising beacons would keep an internal 16 byte counter that would get incremented on each encryption to ensure uniqueness. Due to the nature of BLE broadcasting, packets can drop, which means the counter cannot be incremented implicitly by the Observer, but has to be included in every packet sent by the beacons. Including the 16 byte IV in the advertising packet means 6 byte will be leftover for sending data and implementing other protection mechanisms. This reduces to the same previous problem CFB had, with the added complication of making sure the counter is never reused for

the same key. The beacons would need to have a persistence mechanism to store the value of the last counter used. Whenever they are restarted they would either resume broadcasting from the last counter value stored or new encryption keys would need to be distributed.

3.4 Counter Mode

AES-CTR is an IV-based encryption scheme that turns a block-cipher into a stream cipher. Similarly to OFB, it generates keystreams that are XORed with blocks of the plaintext to obtain a final cipher text. Unlike the previous modes, the value used to generate the keystream is a combination of a "nonce" and a "counter"[12].

This is a normal run when encrypting a message consisting of 36 bytes with CTR:

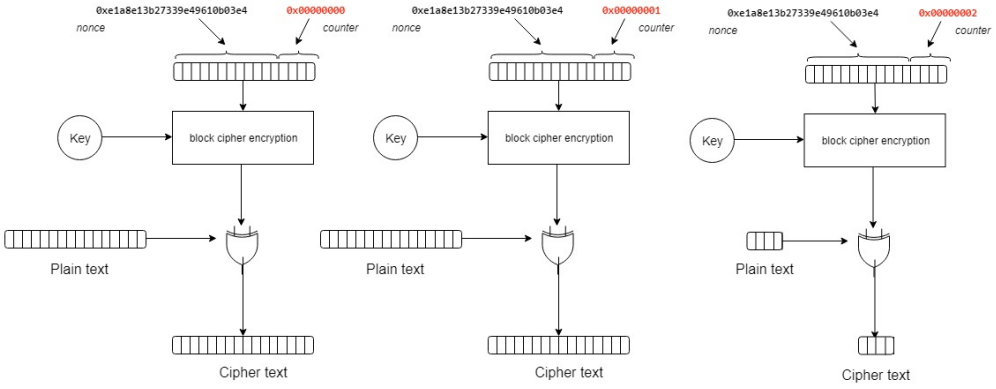

Figure 7: Counter Mode Encryption

The process begins with generating a random 12 byte nonce: `0xe1a8e13b27339e49610b03e4`. The 4 byte counter is initialized to 0 and the initial plaintext is split into 3 blocks that are encrypted independently of each other. At each step, the keystream is generated from the encryption key and a CounterBlock composed of the nonce and an incrementing counter. The last block consisting of only 4 bytes is not padded and is encrypted to a 4 byte cipher text.

3.4.1 CTR Nonce

Generating a nonce is an elegant way to ensure uniqueness of counter blocks across all messages encrypted under a given key. This nonce is concatenated to a counter of variable length to form a CounterBlock. The counter is any function that guarantees a long, non-repeating sequence. Typically a simple increment-by-one function is enough: the counter starts at 0 for block 0 and is incremented by one for each subsequent block. The combination of nonce-counter can be seen as a one-time pad: once "burnt", it can never be reused with the same key without compromising confidentiality[14].

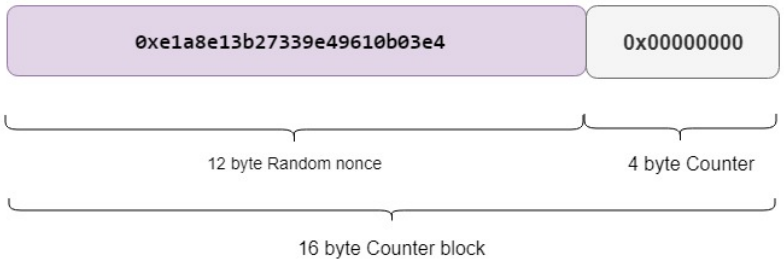

Figure 8: CounterBlock structure

3.4.2 CTR Advantages

Counter mode has several advantages that make it a good choice for encrypting small packets:

- Parallelization and Random-access[11]: The plaintext block that corresponds to a specific cipher block can be recovered independently from the other blocks, as long as its corresponding counter value is known: this allows for parallelization of both encryption and decryption. This is useful when encrypting small packets that may drop and may need to be decrypted out of order[13].
- Arbitrary length messages[13]: The plaintext blocks don't need to be padded and the length of the resulting cipher text is the same as the initial plaintext[14]. This means no bits are wasted in the advertising packet.
- Provable security: the safety of CTR mode is well researched and understood, providing the CounterBlock is never reused[13]
- Simplicity: Encryption and decryption are identical, simplifying the implementation[15]

3.4.3 CTR Disadvantages

The following objections were raised against CTR mode; these have to be carefully considered when deciding on an implementation:

- Malleability: CTR mode offers no guarantees of message integrity or authentication. Ciphertexts are malleable: flipping a bit of the ciphertext will flip the corresponding bit of the plaintext[16]. This makes it possible for an attacker to alter encrypted sensor data such that it decrypts to a packet that has a correct structure and tag, but wrong data. It is crucial that a Message Authentication Code is included along with the packet to prevent such attacks.[13]
- Stateful encryption: The CounterBlock used for encrypting a block has to be known to both the Beacons and the Observer. However, it is possible to keep a part of the CounterBlock implicit (the nonce) and only include the counter explicitely in the packet[[14]]. This is the basic principle that the implementation is based on and will be detailed below
- Sensitivity to usage errors: CTR mode allows users control over the counter; great care has to be taken so that the CounterBlock is not reused when restarting the beacons

After carefully considering the options offered by AES encryption, Counter Mode is clearly the most suitable mode, assuming it is implemented correctly such that the CounterBlock is not reused. Including a 4 byte counter in the packet, for example, leaves 18 free bytes to be used for sensor data and other protections.

4 Related work

The problem of adding security mechanisms to BLE broadcast traffic was studied by several researchers at Google[17]. In order to mitigate tracking and spoofing threats to beacons, they introduce a scheme composed of "cloud-based Ephemeral Identifiers" which allows only authorized parties to identify the beacon broadcasts. EIDs generated cryptographically can only link back to the beacon by authorized users, while to the other users it will appear to be a random ID. This is done through an online trusted third party that acts as a global resolver. This is an efficient way to augment BLE broadcasting with security mechanisms, while handling packet size limitations.

TLS is a cryptographic protocol used to provide communications security over a network. It had several iterations, each meant to increase security and patch various flaws and vulnerabilities. TLS version 1.0 used CBC mode that functioned similarly to RC4: every packet was treated as part of one continuous message. This meant that the ciphertext of the previous packet was used as the IV of the next packet, which made the IV predictable and lead to BEAST attack[18]. TLS version 1.2 changed to use AES-GCM in order to prevent this attack. Instead of the entire IV being included explicitely in the packet, AES-GCM splits the "IV" into a fixed per-connection value and an explicit nonce that is unique per record. This idea of splitting the IV into explicit and implicit parts is crucial in designing the protocol for encrypting beacon traffic[19].

5 Implementation

5.1 Protocol

When encrypting a large message, AES Counter mode dictates that it be divided into blocks of 16 bytes each, which are then encrypted with unique CounterBlocks. The same principle of splitting a long message into blocks and encrypting them with incrementing counters can be applied to beacon traffic. We can think of the "plaintext" as all the data that is sent in an "advertising period" - the period the device is on and broadcasting. In this context, the "plaintext" is split into "message blocks" - all the individual advertising packets.

This is possible due to the Random-access property of Counter Mode: packets can be encrypted independently of each other. However, there is still the bigger issue of communicating the CounterBlock to the Observer. We want to avoid wasting 16 bytes to explicitly communicate it in every packet. A solution is to split the counterblock into a fixed, per-"advertising period" 12 byte nonce, and a 4 byte incrementing counter, that is sent explicitly in each packet. This is inspired by the design of TLS version 1.2 discussed in Section 4 [20]. The current CounterBlock is first transmitted when the advertising period starts and then re-synced every minute between the beacons and gateway. For clarity these terms are defined in the table:

- ***Advertising period***: the period of time from when the device is turned on until it is turned off or the maximum counter value is reached.
- ***CounterBlock***: 16 byte value used along with the encryption key to create a CTR keystream; composed of nonce and counter; has to always be unique per key; is synced periodically between beacons and Observers.
- ***Nonce***: Random 12 byte value; generated fresh at the start of the advertising period; stored both on the beacons and the Observer
- ***Counter***: 4 byte incrementing value that is initialized at 0 at the beginning of the advertising period; when its maximum value: $2^{32} - 1$ is reached, it is reset and a new advertising period starts

There are two types of packets being broadcast:

Sensor Data Packet: Contains the encrypted sensor data and any additional protection mechanisms (18 bytes) plus the counter used during encryption (4 bytes) - resulting in a total length of 22 bytes.

Sync Packet: Broadcasts the entire CounterBlock periodically in order to sync the beacons with the Observer. Contains a 2 byte header that is the ASCII representation of "SY", which is encrypted using the 16 byte CounterBlock inside the packet. The CounterBlock does not need to be encrypted and can be sent "in the clear", according to NIST specification[14]. Once the Observer receives a Sync packet, it tries to decrypt the 2 byte header using the CounterBlock included in the packet and the encryption key of the beacon that sent it. If this header correctly decrypts to the one the Observer expects, the CounterBlock is accepted as correct and the nonce is synced.

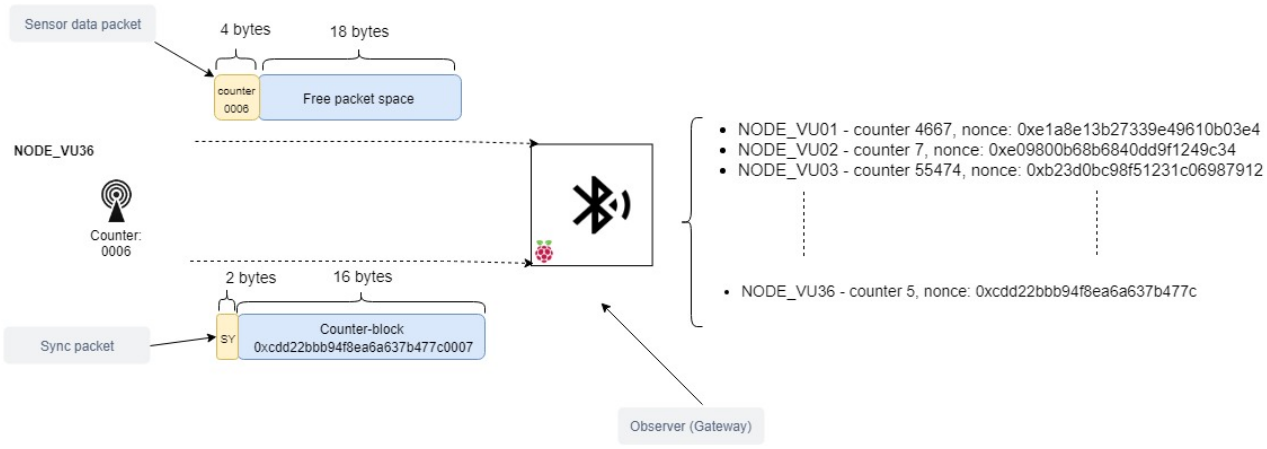

Figure 9: Sync and Sensor data packets

5.2 HMAC

There are three desirable properties when encrypting data:

1. ***Confidentiality***: Information is not disclosed to unauthorized parties. In this case, sensor data is encrypted and only parties that have the encryption key should be able to read it.

2. ***Integrity***: Information cannot be modified by unauthorized parties. The Observer needs to be able to detect when the Sensor Data packet or the Sync packet have been tempered with and reject the received data.

3. ***Authenticity***: The message was sent by an authorized party. When the Observer receives a packet, it is able to detect whether it's been produced using a legitimate encryption key.

Although Counter mode ensures Confidentiality, it makes no guarantees on Integrity and Authentication. As mentioned in section 3, it is a malleable encryption scheme [21]: if the attacker knows the plaintext, they can manipulate the ciphertext in such a way that it decrypts to a message of their chosing[21].

A popular design paradigm for authentication encryption is "generic composition" [22]: a symmetric encryption scheme that only offers confidentiality, such as CTR, is combined with a message authentication scheme, such as HMAC.

The Hash-based Message Authentication Code is a specific type of authentication code that uses a cryptographic hash function along with a secret key to simultaneously verify the integrity and authenticity of a message[23]. Although any hash function can be used, the cryptographic strength of the HMAC relies on the strength of the hash function, as well as the size and quality of the key. After an inner and outer key are derived from the encryption key, the algorithm goes through two passes of computation: the first will derive a hash of the inner key and the message, and the second a hash of the outer key and the inner hash result[23].

After including a 4 byte counter in the packet there are 18 bytes left to budget. The most conservative option, SHA1, will still have an output of 20 bytes that cannot fit in the packet. However, according to [24], a well-known practice with MAC is to truncate the output and include only the truncated part in the sent packet. There are advantages to this: less information about the hash available to the attacker, as well as disadvantages: fewer bytes that have to be predicted. RFC2104 recommends that the hash output is truncated to not less than half of the entire length, to match the birthday attack bound. In addition, it specifies that it should also be not less than 80 bits, which is an acceptable lower bound on the number of bits that need to be predicted by the attacker[24].

There are several composition schemes possible[22]:

- ***Encrypt-and-MAC***:
  - Encrypt the plaintext, and append a MAC of the plaintext.
  - No integrity check on the ciphertext; this makes chosen-ciphertext attacks possible
  - Since CTR is malleable, the ciphertext could be altered, but on decryption this would be obvious
  - Could theoretically reveal information about the plaintext
- ***MAC-then-Encrypt***:
  - A MAC of the plaintext is produced, then the plaintext and MAC are concatenated and encrypted together.
  - Offers integrity of the plaintext, but not of the ciphertext
  - Has not been proven to be strongly unforgeable
- ***Encrypt-then-MAC***:
  - Encrypt the plaintext, calculate a MAC of the resulting ciphertext and concatenate them as the final message.
  - Provides integrity check of the plaintext, as well as the ciphertext
  - Because the MAC is produced on the encrypted text, it does not provide any structural information on the plaintext
  - For malleable cipher schemes, the MAC can be used to filter out any invalid ciphertext
  - Is considered to be the only method which achieves the highest degree of security, providing the MAC used is strongly unforgeable.
  - It is mandatory that distinct keys are used for encryption and the keyed hash[24].

The most suitable composition scheme, and the one that will be used in the system is Encrypt-than-MAC. A SHA1 with a truncated output of 10 bytes will be used to generate an HMAC from the encrypted sensor data. The counter will then be appended to the concatenated ciphertext-MAC .

5.3 Replay Attack Prevention

A replay attack occurs when an actor eavesdrops on private communication, and then fraudulently delays or resends it to misdirect the receiver [25]. Using devices such as Ubertooth or nRF52-DK it is trivial to capture a message and replay it later so that it looks like it's coming from a whitelisted bluetooth address. Such a replayed message would appear to be genuine on the receiving end: the MAC is correct, the message is encrypted and decrypts to the correct plaintext[25].

A common method to prevent such attacks is to include a timestamp along with the message[26]. Senders and receivers are synchronized using a common clock. Any messages are only accepted if they appear to have been generated within an expected timerange[26]. This reduces the window of opportunity for an attacker to eavesdrop.

There are several difficulties with implementing such a prevention mechanism in the Beacons-Observer system. Suppose the beacons would syncronize their clock with the Observer. The packets would need to contain a 2-4 byte Unix timestamp, and this would consume precious packet space. Another possibility is to use the one byte value of the time since power-on. This can potentially be done, as shown in this PoC [27]. However, this introduces more complications: any power-off power-on cycle resets the birthday of the sensor and the advertisement timestamps would conflict with the observer's recorded birthdate.

A better solution to prevent replay attacks is to take advantage of the builtin packet counter. Since the counter is an incrementing value, it will never repeat or decrease until a new advertising period starts. If the Observer "remembers" the last counter value it has seen per beacon, it can reject any packets that contain a value equal or lower to the last one.

The final protocol is shown in Figure 10:

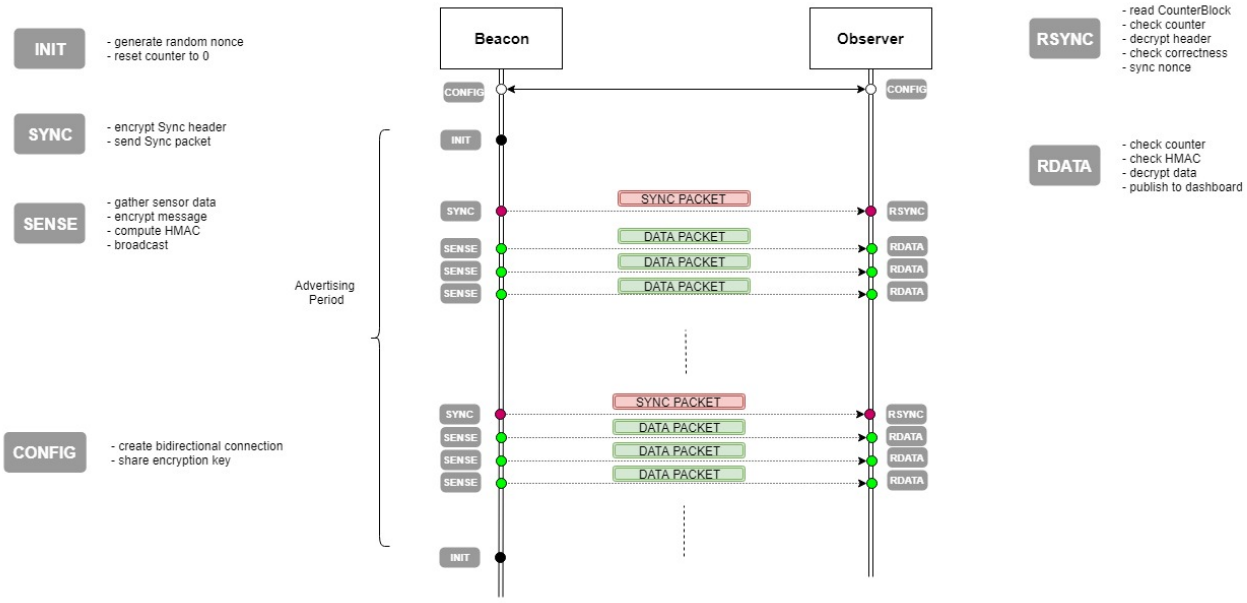

Figure 10: Protocol

6 Evaluation

6.1 Sensor Data Packet

Using this protocol, it is possible to safely advertise a maximum of 8 bytes per data packet. Out of the 22 bytes available, the truncated HMAC occupies 10 bytes, and the counter 4. Every Sensor Data packet relies on a one byte tag that will specify to the Observer what kind of data it contains. There are 7 usable bytes left for actual sensor data.

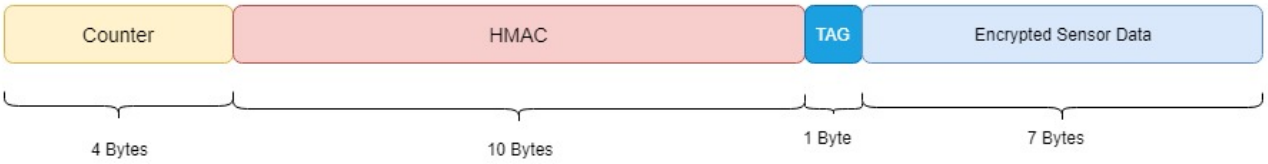

Figure 11: Sensor Data Packet

This is sufficient to advertise packets that contain air quality readings (4 bytes), atmospheric data (6 bytes), acceleration measurements(4 bytes), as well as battery and noise levels (4 bytes). The only other bundle that can't fit into this size is the color readings data (8 bytes). The Nordic BH1745 sensor treats a color channel value as 2 bytes, however, the resolution can be decreased to one byte without compromising too much information.[26]

Sensor data bundle	Data	Size (B)	Fits in the packet
Air quality readings	CO2, TVOC	4	yes
Atmospheric data	temp, hum, press	6	yes
Color readings	red, green, blue, clear	8	no (can reduce res)
Acceleration measurements	x, y, z	6	yes
Misc	noise, battery	4	yes

According to the documentation of the embedded TLS library of the firmware [28], using a 4 byte counter means you can encrypt "at most 2^{96} messages of up to 2^{32} blocks each with the same key". The 2^{96} figure comes from using a 12 byte nonce. This translates to being able to encrypt 2^{32} individual packets before needing to regenerate the nonce and reset the counter to 0. There are 2^{96} advertising cycles that can use the same key before it needs to be changed. Of course, the counter can be decreased to 2 bytes, in order to increase the available space for sensor data. In this case, the nonce will have to change every 2^{16} packets sent. The table below shows the relation between the counter size and the number of packets or advertising periods the key can be used for, under the assumption of a packet sent per second.

CTR Size	Packets	Adv. periods	Data bundles	Reset nonce period	Reset key period
8	2^{64}	2^{64}	2/5	∞	∞
4	2^{32}	2^{96}	4/5	50.000 days	∞
2	2^{16}	2^{128}	5/5	18 hours	∞

6.2 Sync Packet

The Sync packet serves as an easy way to keep the beacons and the Observer synchronized in regards to the nonce currently in use. This simplifies the process of restarting the beacons: on restart they will generate a random nonce which will then be synced with the Observer.

This packet does not need to contain an HMAC. If an attacker were to intercept and modify the packet, there are two possibilities:

- Header is modified: on decryption, it will not have the expected value "SY" and will be rejected by the Observer
- CounterBlock is modified: on decryption, the header will have a garbled value and will be rejected by the observer.

To mitigate Denial of Service attacks, the value of the counter is checked for correctness. A legitimate Sync packet will either have a counter value close to the last one received from the beacon or close to

0 in case the nonce is different than the current one. DoS attacks are still possible if an attacker sends multiple packets with counter value close to 0. Some form of blacklisting mechanism can be implemented on consecutive failed attempts. If protecting against DoS attacks is a strict requirement, the beacon could request a connection to the Observer in order to sync the nonce securely whenever it needs to be regenerated. This could lead to some complications if more than 5-6 beacons need to connect to the Observer at the same time.

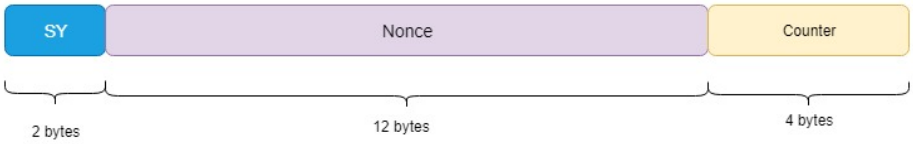

Figure 12: Sync Packet

6.3 Observer

In order for the system to function securely, the Observer is responsible for requesting connection with the beacons for configuration purposes. Espruino allows for a device to connect to 5-6 beacons at a time using *noble* node package. Once connected, it shares the encryption and HMAC key securely. In addition to generating and distributing keys, the Observer also has to keep track of the whitelisted bluetooth addresses and the necessary values per each beacon: encryption and HMAC keys, last known counter and the current nonce.

The configuration phase is computationally intensive due to the necessity of generating encryption keys and requesting connections to the beacons. After this phase ends, there is little overhead on the Observer beyond keeping track of the necessary values per beacon.

6.4 Attacks

6.4.1 Cryptographic Attacks

1. ***Known plaintext*** - not possible. An attacker cannot obtain a plaintext-ciphertext pair.
2. ***Chosen plaintext*** - possible. An attacker could potentially influence the environment, by increasing the "noise" levels for example. The beacons would then produce the ciphertext for the modified sensor data.
3. ***Chosen ciphertext*** - not possible. The attacker cannot request and obtain the plaintext of a ciphertext.[13]
4. ***Side-channel attacks*** - possible. Although the Espruino HMAC module prevents time-based attacks, other types of attacks are possible.
5. ***Replay attacks*** - not possible. The Observer keeps track of the last known counter seen from each beacon and rejects out of date packets.
6. ***Birthday attacks*** - possible. SHA1 is used as the underlying hash of HMAC and is known for having collisions[29]. It is truncated to 10 bytes, which might further weaken it.
7. ***Brute force*** - not possible. The encryption key has a size of 16 bytes, making it computationally secure against brute force attacks[13].

6.4.2 General Attacks

1. ***Eavesdropping*** - not possible. Sensor data is broadcast encrypted and only the Observer can decrypt and read it.

2. ***Man-in-the-middle*** - possible. Any of the usual attacks against Bluetooth Low Energy bonding-pairing process are still possible during configuration phase[1]. However, HMAC assures authentication and integrity of the message[24].

3. ***Denial of Service*** - possible. When the Observer receives a Sync packet it checks wether the counter has a value close to the last known one for the beacon in order to mitigate DoS attacks. It will not attempt decryption or HMAC verification if counter has a suspicious value. However, if the counter has value close to 0, the Observer has no chance but to accept it as correct at any point. DoS is still possible, but can be further mitigated by implementing a blacklisting mechanism.

4. ***Fuzzing attacks*** - possible. There are potential bugs in the Observer code that can be exploited to obtain further information.

Finally, there are other attacks possible under the assumption that the attacker can gain physical access to the beacons. An attacker could dump the Espruino memory and read any hard-coded values or keys. This would also make any of the known-chosen attacks possible by having access to both the hardware and firmware. By using the nRF52-DK from Nordic it is also possible to update the firmware of the beacons without needing to authenticate. These attacks are possible and important to note, but are out of the scope of this project.

7 Conclusion and Future work

This paper looks at several aspects of designing a cryptography protocol for advertising traffic. The design choices were dictated by requirements to enhance system security without compromising the efficiency of advertisements. Currently the system implements AES-Counter encryption with a 4 byte counter, which allows for the same key to be used for encrypting 2^{32} packets for 2^{96} advertising cycles. Under this configuration, the sensor data is sent as five advertising rounds for the five data bundles: air quality, atmospheric readings, color, acceleration and a miscellaneous bundle for noise and battery level. HMAC is added to the packet for message integrity and authentication. The Observer has a check in place to detect replay attacks and reject the packets based on the strictly increasing property of the counter. The configuration phase is not currently implemented and the encryption and HMAC keys are hardcoded. This phase would need to be added before being used in a production environment. It can be implemented by having the Observer request connection to a few beacons at a time. Once connected, the Observer can write the encryption and HMAC key characteristics using the secure link. More information on how to do this can be found in the Espruino documentation[30].

There are many aspects that need to be carefully considered when applying cryptography to a system. Making design choices under strict limitations, such as small packet size, can lead to unsafe compromises. It is important to analyse what is worth compromising in favor of better performance. For example, the decision to use SHA1 as the underlying hash of HMAC was made because it has the smallest output available: 20 bytes. It is not a safe choice, due to SHA1 having known collisions. A more secure choice would be to limit the size of the counter to 2 bytes in order to increase the size available in the packet and use SHA224 instead. This hash algorithm does not have known collisions, but has an output of 28 bytes, with a truncated output of 14 bytes[31]. Under this configuration, the nonce to be regenerated more often, and less sensor data would fit in the packet as a bundle.

When considering efficiency, there is some overhead on the system in the configuration phase, which is the most computationally expensive. The Observer has the difficult task of coordinating with the beacons in order to agree on the encryption and HMAC keys on every reboot. A potential improvement to this configuration phase is to implement as a swarm protocol. The Observer would send initial signals to a few

initial beacons which would then transmit the information further, thus reducing the computational costs of the configuration phase.

Whether adding encryption to such a system is worth the additional complication is dependent on how sensitive the data is and what it is used for. Currently, it is simply read and then published to a dashboard using MQTTS. However, if the received data was used to make control decisions in the system, adding encryption would become a crucial requirement. This is an important factor to keep in mind when adding future functionality. Finally, a formal verification of the protocol would be valuable in deciding the correctness of the system.

References

[1] J. C. B. J. M. T. H. Angela M. Lonzetta, Peter Cope, "Security Vulnerabilities in Bluetooth Technology as Used in IoT," *Journal of Sensor and Actuator Networks*, 2018.

[2] Bluetooth, *BLUETOOTH SPECIFICATION Version 4.2*, 2014.

[3] B. S. M. G. B. D. Ali Nikoukar, Mansour Abboud, "Empirical Analysis and Modeling of Bluetooth Low-Energy (BLE) Advertisement Channels," *Proceedings of 17th IEEE MEDHOCNET*, 2018.

[4] T. Instruments. (2016) BLE Stack User Guide for Bluetooth 4.2. [Online]. Available: http://dev.ti.com/tirex/content/simplelink_cc2640r2_sdk_1_40_00_45/docs/blestack/ble_user_guide/html/ble-stack-3.x/gap.html

[5] D. R. Surthineni. Ashok, "International Journal of Advanced Research in Computer Science and Software Engineering," *International Journal of Advanced Research in Computer Science and Software Engineering*, 2013.

[6] T. Instruments. (2016) BLE Beacons-Application Report. [Online]. Available: http://www.ti.com/lit/an/swra475a/swra475a.pdf

[7] F. M. Pavel Kriz and T. Kozel, "Improving Indoor Localization Using Bluetooth Low Energy Beacons," *Hindawi Publishing Corporation Mobile Information Systems*, 2016.

[8] N. Semiconductor. (2019) Nordic Thingy:52 Documentation. [Online]. Available: https://infocenter.nordicsemi.com/index.jsp

[9] Auth0. (2017) JavaScript for Microcontrollers and IoT: Espruino and the ESP8266. [Online]. Available: https://auth0.com/blog/javascript-for-microcontrollers-and-iot-part-4/

[10] P. Ltd. (2017) BLE Advertising with Node.js/Python/C/Android. [Online]. Available: https://www.espruino.com/BLE+Advertising/

[11] V. Gao. (2016) Debugging Bluetooth with an Android App . [Online]. Available: https://www.bluetooth.com/blog/debugging-bluetooth-with-an-android-app/

[12] D. B. A. B. B. S. V. Pachovski, "Modes of operation of the AES algorithm," *The 10th Conference for Informatics and Information Technology*, 2013.

[13] P. Rogaway, "Evaluation of Some Blockcipher Modes of Operation," *University of California, Davis*, 2011.

[14] N. I. of Standards and Technology, *Recommendation for Block Cipher Modes of Operation*, 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

[15] Wikipedia. (2019) Block cipher mode of operation. [Online]. Available: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

[16] D. A. McGrew, "Counter Mode Security: Analysis and Recommendations," *Cisco Systems, Inc.*, 2002.

[17] M. Y. Avinatan Hassidim, Yossi Matias and A. Ziv, "Ephemeral Identifiers: Mitigating Tracking Spoofing Threats to BLE Beacons," *Google Inc.*, 2016.

[18] N. J. AlFardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," *Information Security Group Royal Holloway, University of London.*

[19] IETF. (2008) The Transport Layer Security (TLS) Protocol Version 1.2. [Online]. Available: https://tools.ietf.org/html/rfc5246

[20] F. Valsorda. (2016) TLS nonce-nse. [Online]. Available: https://blog.cloudflare.com/tls-nonce-nse/

[21] W. T. Liang Xian, "Advanced Encryption Standard (AES) in Counter Mode," *ECE 575 Course Project.*, 2004.

[22] C. N. Mihir Bellare, "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm," *Advances in Cryptology — ASIACRYPT.*, 2007.

[23] Wikipedia. (2019) HMAC. [Online]. Available: https://en.wikipedia.org/wiki/HMAC

[24] IETF. (1997) RFC2014 - HMAC: Keyed-Hashing for Message Authentication. [Online]. Available: https://tools.ietf.org/html/rfc2104

[25] P. F. Syverson, "A taxonomy of replay attacks." *Proceedings of the Computer Security Foundations Workshop*, 1994.

[26] R. B. Sreekanth Malladi, Jim Alves-Foss, "On Preventing Replay Attacks on Security Protocols." *Center for Secure and Dependable Systems Department of Computer Science University of Idaho.*

[27] E. Tsai. (2017) BLE Advertisement Replay Attack and Spoof Detection. [Online]. Available: https://os.mbed.com/users/electronichamsters/notebook/ble-advertisement-replay-attack--spoof-detection/

[28] A. Limited. (2015) tlsMbed API documentation. [Online]. Available: https://tls.mbed.org/api/aes_8h.html#a375c98cba4c5806d3a39c7d1e1e226da

[29] P. K. A. A. Y. M. Marc Stevens, Elie Bursztein, "The first collision for full SHA-1," *CWI Amsterdam, Google Research.*

[30] P. Ltd. (2017) BLE-Interfacing with a PC. [Online]. Available: https://www.espruino.com/Interfacing#bluetooth-le

[31] Wikipedia. (2019) Secure Hash Algorithms. [Online]. Available: https://en.wikipedia.org/wiki/Secure_Hash_Algorithms