Secure protocol buffers for Bluetooth Low-Energy communication with wearable devices

Miguel C. Francisco Samih Eisa Miguel L. Pardal
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa – Lisbon, Portugal
miguel.c.francisco@tecnico.ulisboa.pt samih.eisa@inesc-id.pt miguel.pardal@tecnico.ulisboa.pt

***Abstract*—Wearable devices are further connecting people to the world, extending the reach of smartphones and the Internet. New applications are possible such as activity and location tracking that allows health monitoring and increased access to health services. Bluetooth Low-Energy (BLE) is a pivotal technology for this vision, as it allows power-efficient network connections to smartphones and to service infrastructure. However, there are design flaws and implementation vulnerabilities in BLE that affect the most widely used chipsets and operating systems.**

In this paper, we present POSE, an end-to-end security layer, that can mitigate attacks on BLE pairing and link-layer communications. POSE uses protocol buffers for efficient message data serialization/deserialization and, on top of them, provides message confidentiality and authenticity, including message freshness. POSE was implemented and its processing time, packet overhead, and CPU usage were evaluated. The results show that POSE is an efficient solution for secure communication with wearables and other constrained devices, especially when they already use protocol buffers.

***Index Terms*—Internet of Things, Bluetooth Low-Energy, Security, Privacy, Protocol Buffers.**

I. Introduction

Many people enjoy wearing gadgets such as smartwatches or fitness trackers to keep track of their physical activities. These *wearable devices* have great potential for health monitoring and can also provide increased access to health services. Most of these devices connect to the Internet through the smartphone. Bluetooth [9] is widely used for communication between devices and smartphones, and through them, to the back-end service infrastructure. Bluetooth Low-Energy [8] is an energy-efficient variation of the classic Bluetooth short-range communication technology, designed mostly for constrained devices [15], like wearables and other Internet Of Things (IoT) devices [1].

There are important security problems in BLE, namely design flaws and implementation vulnerabilities. These affect different versions of the standard [34] that, in turn, affect the most widely used operating systems, such as Android and iOS [17]. Security in BLE is provided by the *pairing protocols* performed at the end of the connecting process, where keys are exchanged for data encryption in future communications. Such pairing protocols have also shown design flaws, regarding Android-based devices, making the communications vulnerable to *Downgrade* attacks [34], which exploit Android mishandling errors to cancel the exchange of keys and leave the communication in plain-text, without any warning to the user. Even when the pairing is robust, BLE contains the inherent limitation of providing only *link security*, i.e., the protection only applies to the device-to-device connection. In many use cases, the messages have to use intermediary devices to reach the final destination. For example, a message from a wearable device may need to go through a smartphone before reaching a remote server. All of these problems and vulnerabilities are detailed in Section II.

978-1-7281-8326-8/20/$31.00 ©2021 IEEE

The existing solutions to provide end-to-end security, like TLS [24], can be used, but require significant resources and complex code that will likely exclude many IoT devices. We detail this discussion in Section III.

The pairing limitations/vulnerabilities and the need for lightweight end-to-end communication opens up a space for solutions especially suited to the use of BLE by wearables and other resource-constrained devices. COSE (CBOR Object Signing and Encryption) [28] is such a solution to obtain object security for end-to-end communications. It uses CBOR [3] for efficient data encoding of objects. The implementation of this standard in constrained devices by Tjäder [32] is detailed in Section III-B. The COSE solution is suited to solve the problem that we mentioned, but there is an additional challenge. The devices and applications would need to use CBOR directly, or, otherwise convert to it from other formats. The challenge here is that CBOR is not a very popular format, and there are already many applications that use alternative formats. In constrained devices, this format conversion takes up time and memory, and is a waste of power, with no benefit other than the ability to communicate.

In our work, we developed a new solution that delivers the benefits of a COSE-like approach but relies on a more popular format. We chose *protocol buffers* (abbreviated as *Protobuf*) because it has been increasingly used in applications with strict performance requirements, since its public release in 2008 by Google. In public library repositories, like Maven Central and PyPI, Protobuf libraries are an order of magnitude more popular than CBOR libraries. We propose POSE (Protocol buffer Object Signing and Encryption). It is inspired by the COSE specification, but it has two key differences:

1) it uses Protobuf rather than CBOR;
2) we hardened the secure messages and solved a message freshness vulnerability in COSE.

POSE relies on efficient and popular encoding language to exchange secure messages on pairing-less BLE communications. We implemented our solution and tested it in a

location certification system that relies on Protobuf for its data messaging [26], [14], [6].

A. Use case: Location Certification System

The example system is called SurePresence, and it is part of a health use case, where a patient is able to verify his presence towards a medical appointment, using a wearable device to communicate with a kiosk. This example application is represented in Figure 1.

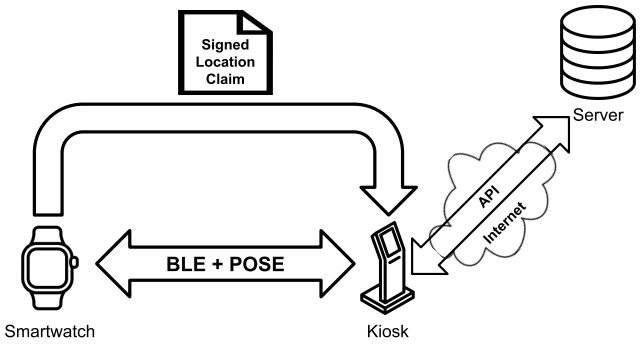

Fig. 1. POSE deployment in the wearable to kiosk communication in a location certification system.

The core concept of the location certification systems is the *Location Proof* [27]. The idea of having the users of the system working as witnesses for other users and certifying their location claims is also a method used by multiple systems to prevent *location spoofing*. In this case, the witness is the kiosk at the medical office. In APPLAUS [36] and CREPUSCOLO [4], the communication between both roles is made in a direct way, meaning that there is no intermediary in the communication, making it faster. Both systems make efforts to minimize collusion attacks but cannot eliminate the attack in all cases. This is true even in more recent systems, such as PASPORT [16]. Recently, there has been an effort to provide better tools for using location proofs, such as the SureThing framework [6]. In all of these systems, it is essential to have proximity to witnesses and, on top of that, to have secure communication with them, including authentication and integrity protection. This is where POSE fits in. Since it natively supports Protobuf, no data format conversions are necessary, and the wearable and kiosk shown in Figure 1 can communicate and mitigate all the security risks of using BLE.

B. Contributions

The main contributions of the paper are the following:

- architecture and implementation of POSE;
- evaluation of POSE with an application developed with actual wearable and other constrained devices;
- performance assessment of POSE.

C. Outline

The remainder of this paper is structured as follows: details on BLE attacks that we want to mitigate are in Section II. Alternatives to communication security are presented in Section III. POSE is specified in Section IV; the implementation and use in a real-world use case is detailed in Section V; the evaluation results are shown and discussed in Section VI; and the conclusion is presented in Section VII.

II. Bluetooth Low-Energy Attacks

There are four different BLE pairing mechanisms to be selected at the end of the connecting process [22]. The "Numeric Comparison" is the most secure method but requires both devices to have input/output capabilities, which is an issue for most IoT devices. The remaining three pairing protocols have been shown to be insecure [18], leaving communications vulnerable to *Eavesdropping* and *Man-In-The-Middle* (MITM) attacks. Pallavi and Narayanan [19] have shown the feasibility of such types of attacks on devices without any input/output (I/O) capabilities. The only possible pairing mechanism, in this case, is "Just Works" which is the most simple pairing protocol, allowing the capture and manipulation of data, as well as command injection. This pairing mechanism generates the Long Term Key (LTK) from a Term Key (TK) always set to `0x00`, which causes the messages to be exchanged in plaintext and without authentication from any of the two devices. An application-based authentication phase after the pairing phase is necessary to ensure protection against MITM attacks [13].

Zhang et al. [34] showed that such pairing protocols also present four specific design flaws in Android, the most widely used operating system for mobile and for interactions with wearables and other IoT devices [2]. These design flaws are serious security problems as they make the BLE exchanged messages vulnerable to *Man-In-The-Middle* and *Downgrade* attacks, as well as *Information Disclosure* from the connected devices. The four Android BLE design flaws are:

- An Android application cannot enforce a specific pairing protocol even if it knows the capable pairing protocols of the peer BLE device;
- Android applications cannot cancel insecure pairing processes until conclusion or remove suspicious pairing connections;
- Android mishandles pairing errors, without notifying the application or the user, being vulnerable to rogue devices attempting MITM attacks;
- There are no mechanisms to obtain the negotiated pairing protocol on time or even start a new secure pairing process with the same peer BLE device.

Zhang et al. also showed [35] how to explore such flaws with practical attacks using fake devices. These attacks include *Downgrade* and injection of false data on the vulnerable device and *Denial-of-Service* (DoS). The security flaws are shown to be extended to other major OSes including Windows, Linux, and iOS [17]. The same authors present countermeasures for all security issues which include the pairing error handling at the application level and enforcing a specific pairing method and notifying the application in time about it. However, the countermeasures carry a usability trade-off for users.

Wu et al. [33] explore the vulnerabilities of the BLE link-layer to spoofing attacks by designing an attack (BLESA) where a rogue device pretends to be a previously-paired server device. This attack showcases the security flaws in the reconnection process of two previously paired devices. The rogue device rejects the authentication requests and can feed spoofed data to the client device. The authors also present mitigation techniques including fixing implementation bugs in the BLE stack and adjusting the security level of the connection based on the attributes access requirements, before sending the reading request.

All of these attacks show that BLE security is not good enough and there is a need for additional security solutions.

III. Alternatives for Communication Security

We analyze alternatives for secure communication when using BLE. There are transport and application-level solutions.

A. Transport-level security

TLS [24] can be used above the BLE link-layer, but it requires significant resources to keep session context that excludes many constrained devices [7]. DTLS [25] is also not a direct option, even though there are optimized versions for IoT [5], [21]. However, they are not widely available and consume significant power resources [31]. Overall, the use of session-based security communication that assumes 1-to-1 communication, between the same client and server, with a strict correspondence is not possible to assure in many cases when constrained devices are involved, because there is the need to keep the same security context. There are also other uses that require the protection to be verified later, for example, when the results are cached.

B. Application-level security

Another possibility is to use an application-layer protocol to obtain all necessary security guarantees. There can be custom solutions, but each application is at risk of re-implementing cryptographic protocols that are not reviewed enough. The ideal would be to rely on a standard that has been defined, revised, and implemented by a significant community effort.

Tjäder [32] has shown how to guarantee message confidentiality in BLE exchanges by protecting those communications with COSE [28]. COSE specifies how to process signatures, encryption, and Message Authentication Codes (MAC) computations for CBOR (Concise Binary Object Representation) message-encoding format. This standard is based on JOSE (Javascript Object Signing and Encryption) [11] a similar standard that uses JSON (JavaScript Object Notation) as the underlying format. CBOR is more efficient for data transport than JSON, as the former is binary-based, and the latter is text-based.

COSE has been used in RESTful Environments to protect CoAP messages exchanged between constrained devices, through OSCORE [29]. The OSCORE (Object Security for Constrained RESTful Environments) specification describes how to protect the payload and the metadata header fields of CoAP messages, using COSE to ensure confidentiality and integrity. However, OSCORE still uses the CBOR data format.

COSE does not address *message freshness*, making all messages vulnerable to replay attacks, unless the application itself adds some robust repeated message detection.

In summary, COSE ensures object security by providing confidentiality, authenticity, and integrity of the exchanged messages, but it does not assure freshness and it is coupled with the CBOR data format.

IV. POSE Design

We now present POSE (Protocol buffer Object Signing and Encryption) an application-layer protocol on top of pairing-less BLE communications, providing all security guarantees while increasing the usability of BLE, thanks to the absence of any pairing process. Fundamentally, POSE follows the COSE specification. However, it takes full advantage of Protobuf as its underlying data encoding format to represent the exchanged messages over BLE communications. Protobuf provides a convenient way to structure data in a compact way and then use `Protoc`, a protocol buffers code generator, to easily write and read those structured data in the most popular programming languages [12].

A. Choice of data format

CBOR [3] is a standardized data format similar to JSON, whose goals include small message size and extensibility to binary fields. Unlike JSON, it is not a universal data format among all types of devices, which causes constant conversion from other formats, which is not practical, especially for resource-constrained devices. Moreover, Jenkov [10] has shown in practice that CBOR messages with five different fields have a significantly smaller read and write throughput than equivalent Protobuf messages. Protobuf, originally proposed by Google, has already been introduced in the world of heterogeneous constrained devices as a lightweight and interoperable alternative to standardized message encoding schemes like CBOR, JSON, or BSON [20]. This data format has surpassed CBOR in popularity over the years, showing a much higher number of usages in multiple code repositories, including Maven Central[1] (3589 and 446, respectively) and PyPI[2] (13301 and 37, respectively). The "universality" of the data format is especially important in limited devices because they may not have the means to convert from and to other formats, at least, not without wasting precious energy. POSE opts to use Protobuf to leverage its popularity, and compare favorably to COSE in this regard.

B. Message Types

In the context of a single recipient one-way BLE communications, POSE specifies three message types, each offering different security guarantees. These messages depend on the knowledge of who is the recipient and on an implicit symmetric key/asymmetric key pair, previously established. The

[1]https://search.maven.org/

[2]https://pypi.org/

TABLE I
POSE MESSAGE TYPES.

POSE Message Type	Semantics
POSE_Sign1	Signed data object
POSE_Mac0	Mac data object
POSE_Encrypt0	Encrypted data object

```
message POSE_Sign1{
    bytes protected = 1;
    HeaderMap unprotected = 2;
    bytes payload = 3;
    bytes signature = 4;
}
```

Listing 1: POSE_Sign1 Protocol Buffer definition.

different POSE message types can be seen in Table I and they follow the same grammar and present the same fields as all the remaining COSE objects, as detailed in Section IV-C.

1) Signed data object: The `POSE_Sign1` message type object ensures the non-repudiation of the transmitted data, by one signer only. It makes use of the same signature algorithms as used in the COSE standard, which is the Edwards-curve Digital Signature Algorithm (EdDSA) and the Elliptic Curve Digital Signature Algorithm (ECDSA). The protected bucket and payload are both signed, where the latter can even include another POSE message type, allowing the creation of sealed objects. The `POSE_Sign1` protocol buffer definition can be seen in Listing 1.

2) Mac data object: The `POSE_Mac0` message type object ensures integrity and authenticity of both the payload and the protected bucket by generating a tag with a Message Authentication Code (MAC). It makes use of the same algorithms as COSE, which can either be a block cipher algorithm, like AES-MAC, or a hash algorithm, like HMAC. The Protobuf definition of the `POSE_Mac0` message type can be seen in Listing 2.

3) Encrypted data object: The `POSE_Encrypt0` message type object is the one that offers the most security guarantees and is the type in which we focused on in our implementation (Section V). This type ensures message confidentiality and integrity and message freshness. The Protobuf definition of the `POSE_Encrypt0` message type can be seen in Listing 3.

C. Message Format

POSE protocol supports the same primitive types as defined in the COSE specification, like Booleans, Integers, Byte strings, etc. Each field in a POSE object message can be a primitive type or Protobuf-defined message, depending on the type of the message, however, it always starts with the three fields: protected header parameters, unprotected header parameters, and the content of the message. Both protected and unprotected headers are 'label'-'value' maps. Each label is a well-defined Integer, while a value is a primitive type or another POSE message. The protected bucket contains parameters about the current layer and information about the used cryptographic algorithms, allowing flexibility on the chosen algorithms. All this information is protected since it is used in the cryptographic computation, where is either signed, hashed, or used as associated data in the encryption, as explained in Section V-A. The unprotected field is similar but not protected, meaning that it may not be authentic when it reaches the recipient. Both fields are serialized into a byte string. The content of the message field can be either plain-text, cipher-text, or another POSE message.

```
message POSE_Mac0{
    bytes protected = 1;
    HeaderMap unprotected = 2;
    bytes payload = 3;
    bytes tag = 4;
}
```

Listing 2: POSE_Mac0 Protocol Buffer definition.

```
message POSE_Encrypt0{
    bytes protected = 1;
    HeaderMap unprotected = 2;
    bytes ciphertext = 3;
}
```

Listing 3: POSE_Encrypt0 Protocol Buffer definition.

D. Message Confidentiality and Integrity

The different security guarantees are provided by the different message types, detailed in Section IV-B. To ensure both the confidentiality and integrity of the transmitted data, the `POSE_Encrypt0` message must be used. It uses the Authenticated Encryption with Associated Data (AEAD) as the form of encryption and the Advanced Encryption Standard-Galois/Counter Mode (AES-GCM) as the mode of operation for the encryption computation of the payload.

AES-GCM takes full advantage of pipelining and processing techniques unlike the remaining modes of operations of block ciphers, making it fit for constrained devices with inexpensive resources and lower rate applications of the IoT world [30], although other modes may offer better security-space tradeoffs. Other cryptographic algorithms can be used thanks to the specific format of POSE messages, detailed in Section IV-C, allowing exchange information about the algorithms used by the sender and that should be used by the recipient. This promotes flexibility on the encryption algorithms and opens a door for the usage of other more lightweight algorithms.

AEAD ensures the authenticity of the associated data used in encryption along with the integrity and confidentiality of the cipher-text. The encryption makes use of a pre-shared 128-bit symmetric key. We leave a possible handshake for a key agreement for future work. This mode of operation requires a *nonce* which is also used to protect against replay attacks, as shown in Section IV-E.

```
message Enc_Structure{
    string context = 1;
    bytes protected = 2;
    HeaderMap unprotected = 3;
    POSE_Encrypt0 body = 4;
}
```

Listing 4: Enc_Structure Protocol Buffer definition.

E. Message Freshness

As previously described in Section III, the COSE standard does not protect from replay attacks nor addresses message freshness. POSE solves the first problem by adding the nonce used in the AEAD encryption to the protected field of the exchanged message. This nonce is verified and remembered by the message validator, in the receiving end. The nonce persistence time in memory is application and machine resources-dependent.

The message freshness problem is addressed at the application level. Our location certification system used to implement POSE issues location proofs with daily granularity timestamps, which means it does not accept or validate messages containing location proofs from past days, hence guaranteeing location proof freshness.

V. POSE Implementation

We implemented POSE in a two-component example application. The first component is an Android client application written in Java and deployed on a quad-core HUAWEI Watch 2 running Android Wear OS. The second component is a kiosk application written in Python 3 deployed in a Raspberry Pi 4 equipped with a touchable screen.

In Section V-A we detail how to build the `POSE_Encrypt0` message object according to format and fields previously explained. In Section V-B we show how POSE is used in an example application.

A. Security processing

The first step to produce a `POSE_Encrypt0` message object is to generate the associated data that will be used in the cryptographic computation. For that purpose, we need to create a structure that will include all the protected fields that require to be authentic when they reach the recipient and then create a consistent byte stream from it, so that it can be computed. That new structure is the `Enc_Structure` object and its Protobuf definition is shown in Listing 4, as defined by the POSE protocol. The subsequent sections describe how to process the security protection and verification.

1) Payload protection: We describe the steps required to securely protect any payload using the `POSE_Encrypt0`, from the generation of the associated data to the content of the final message.

i Create the `Enc_Structure` message object and fill it with the appropriate fields such as the context string and the protected attributes. The context string is a well-defined string that identifies the type of the payload.

```
{
    context: "Encrypt0"
    protected: map: {5: E090434C972...}
    unprotected : {}
    body {  //POSE_Encrypt0
        protected: map: {1: 10}
        unprotected {}
        ciphertext: 1FDF435D9C2C80B...
    }
}
```

Listing 5: Enc_Structure implementation.

ii Serialize the created `Enc_Structure` object into byte stream using protocol buffers encoding to create the associated data.

iii Call the encryption algorithm with the previously loaded encryption key K, the plain-text of the `POSE_Encrypt0` message object, and the associated data (AD). Then include the result in the cipher-text field of the `POSE_Encrypt0`.

2) Payload verification: Similar to the previous section, we describe the steps required to securely verify the contents of any received `POSE_Encrypt0` message.

i Create an object of `Enc_Structure` message similar to the one received but without the cipher-text.

ii Serialize the created `Enc_Structure` object into byte stream using protocol buffers encoding to create the associated data.

iii Call the decryption algorithm with the previously loaded decryption key K, the cipher-text of the `POSE_Encrypt0` object from the received `Enc_Structure` message, and the associated data (AD).

B. POSE in SurePresence

The example application, SurePresence, allows a patient to prove his presence at a medical clinic. The implemented process of this medical appointment use case is represented in Figure 1

The smartwatch to kiosk exchange was achieved through one-way BLE communication that allows the patient, using a running app on his smartwatch, to exchange messages with a kiosk device at the medical clinic to prove his presence. The patient acts as a location prover and the kiosk as a witness to endorse the presence of the patient at the clinic. The security needs to be guaranteed by successfully applying the POSE protocol to provide the desired BLE application security level. The exchanged BLE packet between the smartwatch and the kiosk device uses our implementation of the `Enc_Structure`, which can be seen in Listing 5.

The cipher-text is an encrypted location claim. The patient generates a location claim with his app, containing the user, location, and time information. Then the patient signs the claim to guarantee the non-repudiation of his claim. This discards the necessity to use the `POSE_Sign1` message object, previously explained. The smartwatch advertises the

BLE packet with the `Enc_Structure` object to the kiosk device. Without POSE, the signed location claim would be in plain-text and, therefore, an attacker could read the user-sensitive information. However, with POSE implemented, we guarantee the authenticity and confidentiality of the location claim generated by the patient. The kiosk device on the other end follows the decryption methodology, previously explained, and signs the location claim creating a location endorsement to be submitted to a location service provider as part of the location certification system.

VI. Evaluation

This Section presents a comparison between the developed POSE protocol and the COSE standard on the same example application. As mentioned before, POSE was developed mainly for constrained devices and IoT applications. Thus, we evaluated POSE with respect to: time required to create and serialize/deserialize the exchanged messages, packet overhead, and performance metrics including the CPU usage. The comparison baseline was COSE.

This evaluation will help us answer the following questions:

1) Is POSE faster than COSE in serializing/deserializing the exchanged BLE messages?
2) How large is the packet overhead of POSE in the exchanged messages compared to COSE and BLE alone?
3) How significant is the CPU usage of POSE when compared to COSE?

The creation and serialization of the exchanged messages includes the following operations:

- Creation of the `Enc_Structure` object with the required protected and unprotected fields
- Generation of associated data based on the Protobuf/CBOR serialization of the `Enc_Structure` to bytes
- Encryption of the payload using the AEAD algorithm
- Creation of the `POSE_Encrypt0` object inside the `Enc_Structure`.

The deserialization of the exchanged messages includes similar operations, with a decryption computation instead of an encryption one.

A. Processing time

The processing time measures the required time for the serialization and deserialization of the 150 messages that have been exchanged between the smartwatch and the kiosk devices during the experiments using POSE and the standard COSE protocol. The results are presented in Table VI-A.

The obtained results show high processing time and mostly similar values in both protocols with slightly lower processing time in COSE serialization.

B. Packet Overhead

Packet overhead measures the additional data that POSE requires to securely exchange payloads between the prover device (smartwatch) and the witness (kiosk). In this experiment, we compared POSE packet overhead against the COSE overhead and also against the BLE configuration with no security features (without pairing) as well as BLE pairing with authenticated LE Secure Connections [23].

TABLE II
Time required to process the POSE and COSE messages.

	Serialization (ms)		Deserialization (ms)		Total Time (ms)
	avg.	std.	avg.	std.	sum
COSE	144.66	15.188	257.79	2.102	402.45
POSE	164.85	24.301	257.09	2.544	421.94

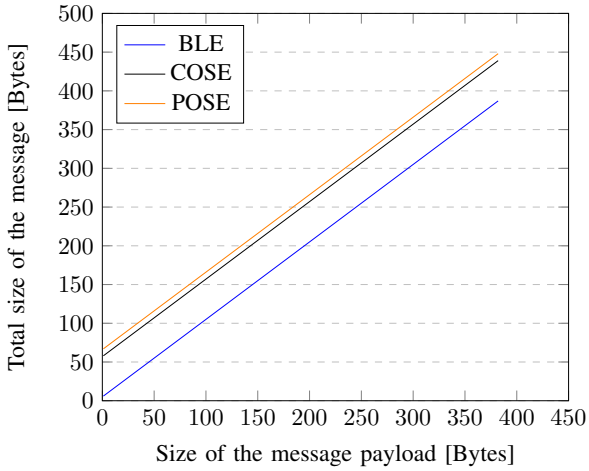

Fig. 2. Packet overhead of exchanged messages in multiple protocols.

Figure VI-B illustrates the obtained results. For simplicity, we merged the overhead results of the two BLE levels into just BLE, as both showed similar results in the experiments. The results were captured on the kiosk device using the Wireshark tool[3].

As shown in Figure VI-B, packet overhead generated by POSE protocol stays linear as the size of the application payload increases, showing around 66 bytes overhead. This is justified due to the mandatory added protected and unprotected fields of the message types in the POSE protocol. The generated packet overhead by the COSE protocol presents the same behavior, showing around 57 bytes overhead, which is slightly less than POSE. On the other hand, the packet overhead of both BLE modes is residual, showing around 5 bytes overhead. However, the increasing packet overhead of POSE is justified by the increasing security guarantees and the added message freshness feature. Overall, the packet overhead difference between the two protocols can be seen as the trade-off between efficiency and protection against replay attacks.

C. CPU Usage

The CPU usage metric is used to quantify how the smartwatch of the prover and the kiosk processor have been utilized

[3]https://www.wireshark.org/

TABLE III
CPU USAGE MEASUREMENTS ON THE SERIALIZATION OF MESSAGES

	CPU cores used	CPU execution time (%)	Wall Clock Duration (s)	CPU duration (s)
COSE	2	76.01	16.25	14.16
POSE	2	80.53	17.13	14.97

when creating and serializing/deserializing data packets. High CPU usage may indicate that the protocol has high demands for processing power. The experiment here is similar to the processing time experiment, described in section VI-A. We collected measurements of 150 messages. These messages were generated on the wearable device (smartwatch) and read through BLE on the kiosk device. The reading cycle period for each message was 4 seconds, to allow for the smartwatch to restart Bluetooth, since each interaction needs to start from the radio activation. During this experiment, we obtained CPU usage measurements on both constrained devices: CPU usage of the main thread responsible for the data payload on the smartwatch and the CPU usage on the receiving kiosk device. We divided the evaluation into the following sections based on the main processes of the protocol.

1) Serialization: The results obtained for the CPU usage using the Android Profiler tool[4] are shown in Table III.

The example application for the experiment ran using two cores. The information about the frequency of the CPU cores was not available for this kind of equipment. The CPU execution time gives us information about the percentage of time the CPU was used to execute the scheduled job when compared to the main thread. This means that 76.01% of the total CPU execution time of the main thread of the application was serializing the COSE messages. Regarding this metric, the results for both protocols were similar. The Wall Clock Duration is the total elapsed real-time of the serialization, while the CPU duration is the total time the serialization process consumed CPU resources. The Wall Clock Duration includes the time the main thread responsible for the experiment was idling. Overall, the CPU usage of both protocols is very similar, consuming similar resources, with close execution times. We leave for future work a more thorough analysis of the CPU cores used behavior.

2) Deserialization: The total waiting time of the deserialization part was 10 min (4s x 150 samples); while the total deserialization time was 21 seconds (21699 ms) for all messages. We obtained the CPU usage during the whole experiment using the uptime[5] Linux command, with a polling rate of 1 second. Therefore, the more meaningful and more representative measurement is the "Last 1 Minute" one, which can be seen in Table VI-C2.

The reference value for CPU full utilization is 4.0 since the Raspberry Pi 4 is quad-core. The average CPU usage values measured for the last 1 minute for both protocols are similar, justified by the overlapping confidence intervals. The CPU utilization in both COSE and POSE (0.310 and 0.463, respectively) is low and shows the CPU bottleneck is not a problem for POSE, regarding constrained devices.

[4]https://developer.android.com/studio/profile/android-profiler

[5]https://man7.org/linux/man-pages/man1/uptime.1.html

TABLE IV
CPU USAGE ON THE DESERIALIZATION OF THE POSE AND COSE MESSAGES.

	Load average on Deserialization (%)					
	Last 1 Minute		Last 5 Minutes		Last 15 Minutes	
	avg.	std.	avg.	std.	avg.	std.
COSE	0.310	0.1254	0.416	0.2534	0.306	0.1753
POSE	0.463	0.4685	0.384	0.2938	0.296	0.1418

D. Discussion

The obtained results of the packet overhead were surprising considering Protocol Buffer messages are more efficient and lightweight than most data formats. Although, the 9 bytes difference of POSE when compared to COSE is explained by the bytes required to identify each field of the Protobuf message. We consider a trade-off between these 9 bytes overhead and protection against replay attacks since COSE does not address message freshness.

Both protocols showed a low CPU usage which was expected regarding the complexity of the exchanged messages. Although, the similarity of the results between both protocols on both sides was surprising. Serialization and deserialization of Protobuf messages should be much more efficient, which leads us to think that the processing power of the CPU is much more important rather the data format. The total time required to process POSE messages is too high and can jeopardize the scalability of our protocol.

VII. CONCLUSION

In this paper, we presented POSE, an application-layer security protocol that can be used over Bluetooth Low Energy (BLE) communications on wearables and other constrained devices, and mitigate its security risks. It follows an approach very similar to COSE, but replaces CBOR with the much more popular Protobuf data-encoding format, and also adds message freshness protection. POSE was tested with an example application for location certification that allows users to prove their locations and presence with their devices. The results show the feasibility and efficiency of POSE protocol on top of pairing-less BLE connections with similar processing times as COSE on both serialization and deserialization and just a small increase of packet overhead when compared to COSE. Our results also show efficient CPU usage.

POSE shows great promise to increase the communication security for many other mobile and IoT applications, with wearables and other constrained devices, especially the ones that already use Protobuf for its messages.

ACKNOWLEDGEMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference

UIDB/50021/2020 (INESC-ID) and through the project with reference PTDC/CCI-COM/31440/2017 (SureThing).

REFERENCES

[1] Architecting the Internet of Things. Springer-Verlag GmbH (2011), https://www.ebook.de/de/product/14076594/architecting_the_internet_of_things.html

[2] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., Ayyash, M.: Internet of things: A survey on enabling technologies, protocols, and applications. IEEE Communications Surveys Tutorials **17**(4), 2347–2376 (2015)

[3] Bormann, C., Hoffman, P.E.: Concise Binary Object Representation (CBOR). RFC 8949 (Dec 2020). https://doi.org/10.17487/RFC8949, https://rfc-editor.org/rfc/rfc8949.txt

[4] Canlar, E.S., Conti, M., Crispo, B., Di Pietro, R.: Crepuscolo: A collusion resistant privacy preserving location verification system. In: 2013 International Conference on Risks and Security of Internet and Systems (CRiSIS). pp. 1–9 (Oct 2013). https://doi.org/10.1109/CRiSIS.2013.6766357

[5] Capossele, A., Cervo, V., De Cicco, G., Petrioli, C.: Security as a coap resource: an optimized dtls implementation for the iot. In: 2015 IEEE international conference on communications (ICC). pp. 549–554. IEEE (2015)

[6] Ferreira, J., Pardal, M.L.: Witness-based location proofs for mobile devices. In: 17th IEEE International Symposium on Network Computing and Applications (NCA) (Nov 2018)

[7] Gerez, A.H., Kamaraj, K., Nofal, R., Liu, Y., Dezfouli, B.: Energy and processing demand analysis of tls protocol in internet of things applications. In: 2018 IEEE International Workshop on Signal Processing Systems (SiPS). pp. 312–317. IEEE (2018)

[8] Gomez, C., Oller, J., Paradells, J.: Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. Sensors **12**(9), 11734–11753 (2012)

[9] Haartsen, J.C.: The bluetooth radio system. IEEE personal communications **7**(1), 28–36 (2000)

[10] Jenkov, J.: Rion vs. json vs. protobuf vs. messagepack vs. cbor (Sep 2019), http://tutorials.jenkov.com/rion/rion-performance-benchmarks.html

[11] Jones, M., Bradley, J., Sakimura, N.: JSON Web Signature (JWS). Tech. Rep. 7515 (May 2015). https://doi.org/10.17487/RFC7515, https://rfc-editor.org/rfc/rfc7515.txt

[12] Kaur, G., Fuad, M.M.: An evaluation of protocol buffer. In: Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon). pp. 459–462 (March 2010). https://doi.org/10.1109/SECON.2010.5453828

[13] Lounis, K., Zulkernine, M.: Bluetooth low energy makes "just works" not work. In: 2019 3rd Cyber Security in Networking Conference (CSNet). IEEE (oct 2019). https://doi.org/10.1109/csnet47905.2019.9108931

[14] Maia, G.A., Claro, R.L., Pardal, M.L.: Cross city: Wi-fi location proofs for smart tourism. In: Grieco, L.A., Boggia, G., Piro, G., Jararweh, Y., Campolo, C. (eds.) Ad-Hoc, Mobile, and Wireless Networks. pp. 241–253. Springer International Publishing, Cham (2020)

[15] Nieminen, J., Savolainen, T., Isomaki, M., Patil, B., Shelby, Z., Gomez, C.: Ipv6 over bluetooth(r) low energy. RFC 7668, RFC Editor (10 2015), https://www.rfc-editor.org/rfc/rfc7668.txt

[16] Nosouhi, M.R., Sood, K., Yu, S., Grobler, M., Zhang, J.: Pasport: A secure and private location proof generation and verification framework. IEEE Transactions on Computational Social Systems **7**(2), 293–307 (April 2020). https://doi.org/10.1109/TCSS.2019.2960534

[17] Novac, O.C., Novac, M., Gordan, C., Berczes, T., Bujdosó, G.: Comparative study of google android, apple ios and microsoft windows phone mobile operating systems. In: 2017 14th International Conference on Engineering of Modern Electric Systems (EMES). pp. 154–159. IEEE (2017)

[18] Padgette, J., Bahr, J., Batra, M., Holtmann, M., Smithbey, R., Chen, L., Scarfone, K.: Guide to bluetooth security. Tech. rep. (may 2017). https://doi.org/10.6028/nist.sp.800-121r2

[19] Pallavi, S., Narayanan, V.A.: An overview of practical attacks on ble based iot devices and their security. In: 2019 5th International Conference on Advanced Computing Communication Systems (ICACCS). pp. 694–698 (2019). https://doi.org/10.1109/ICACCS.2019.8728448

[20] Popic, S., Pezer, D., Mrazovac, B., Teslic, N.: Performance evaluation of using protocol buffers in the internet of things communication. In: 2016 International Conference on Smart Systems and Technologies (SST). IEEE (oct 2016). https://doi.org/10.1109/sst.2016.7765670

[21] Raza, S., Trabalza, D., Voigt, T.: 6lowpan compressed dtls for coap. In: 2012 IEEE 8th International Conference on Distributed Computing in Sensor Systems. pp. 287–289. IEEE (2012)

[22] Ren, K.: Bluetooth pairing part 4: Bluetooth low energy secure connections – numeric comparison (Jan 2017), https://www.bluetooth.com/blog/bluetooth-pairing-part-4/

[23] Ren, K.: Bluetooth pairing part 4: Bluetooth low energy secure connections – numeric comparison (Jan 2019), https://www.bluetooth.com/blog/bluetooth-pairing-part-4/

[24] Rescorla, E.: The transport layer security (tls) protocol version 1.3. RFC 8446, RFC Editor (08 2018), https://www.rfc-editor.org/rfc/rfc8446.txt

[25] Rescorla, E., Modadugu, N.: Datagram Transport Layer Security Version 1.2. Tech. Rep. 6347 (Jan 2012). https://doi.org/10.17487/RFC6347, https://rfc-editor.org/rfc/rfc6347.txt

[26] Santos, H.F.: Stop: Secure transport location proofs for vehicle inspections (2019)

[27] Saroiu, S., Wolman, A.: Enabling new mobile applications with location proofs. In: Proceedings of the 10th Workshop on Mobile Computing Systems and Applications. HotMobile '09, Association for Computing Machinery, New York, NY, USA (2009), https://doi.org/10.1145/1514411.1514414

[28] Schaad, J.: CBOR Object Signing and Encryption (COSE). Tech. Rep. 8152 (Jul 2017). https://doi.org/10.17487/RFC8152, https://rfc-editor.org/rfc/rfc8152.txt

[29] Selander, G., Mattsson, J.P., Palombini, F., Seitz, L.: Object Security for Constrained RESTful Environments (OSCORE). Tech. Rep. 8613 (Jul 2019). https://doi.org/10.17487/RFC8613, https://rfc-editor.org/rfc/rfc8613.txt

[30] Sung, B.Y., Kim, K.B., Shin, K.W.: An AES-GCM authenticated encryption crypto-core for IoT security. IEEE (jan 2018). https://doi.org/10.23919/elinfocom.2018.8330586

[31] Tiburski, R.T., Amaral, L.A., de Matos, E., de Azevedo, D.F., Hessel, F.: Evaluating the use of tls and dtls protocols in iot middleware systems applied to e-health. In: 2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC). pp. 480–485. IEEE (2017)

[32] Tjäder, H.: End-to-end Security Enhancement of an IoT Platform Using Object Security. Master's thesis, Linköping University, Information Coding (2017)

[33] Wu, J., Nan, Y., Kumar, V., Tian, D.J., Bianchi, A., Payer, M., Xu, D.: BLESA: Spoofing attacks against reconnections in bluetooth low energy. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association (Aug 2020), https://www.usenix.org/conference/woot20/presentation/wu

[34] Zhang, Y., Weng, J., Dey, R., Jin, Y., Lin, Z., Fu, X.: On the (in)security of bluetooth low energy one-way secure connections only mode. ArXiv **abs/1908.10497** (2019)

[35] Zhang, Y., Weng, J., Dey, R., Jin, Y., Lin, Z., Fu, X.: Breaking secure pairing of bluetooth low energy using downgrade attacks. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 37–54. USENIX Association (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/zhang-yue

[36] Zhu, Z., Cao, G.: Applaus: A privacy-preserving location proof updating system for location-based services. In: Proceedings IEEE INFOCOM. pp. 1889–1897 (April 2011). https://doi.org/10.1109/INFCOM.2011.5934991