UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

Studio e analisi del protocollo CBOR Object Signing and Encryption (COSE) per applicazioni Internet of Things

Study and analysis of the CBOR Object Signing and Encryption (COSE) protocol for Internet of Things applications

Candidato:
Tommaso Fava

Relatore:
Ing. Paola Pierleoni

Correlatore:
Dott.ssa Luisiana Sabbatini

Anno Accademico 2021-2022

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

Studio e analisi del protocollo CBOR Object Signing and Encryption (COSE) per applicazioni Internet of Things

Study and analysis of the CBOR Object Signing and Encryption (COSE) protocol for Internet of Things applications

Candidato:
Tommaso Fava

Relatore:
Ing. Paola Pierleoni

Correlatore:
Dott.ssa Luisiana Sabbatini

Anno Accademico 2021-2022

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica
Via Brecce Bianche – 60131 Ancona (AN), Italy

Ai miei nonni
Duilio, Ersilia, Bruno e Liliana

Ringraziamenti

Alla fine di questo percorso, pieno di soddisfazioni e anche di qualche difficoltà, ci tengo a ringraziare tutti quelli che hanno lasciato un loro contributo.

Dalla mia famiglia, che ha fornito un supporto prezioso e costante nei momenti più impegnativi, fino ai docenti ed ai colleghi, inclusi quelli fantastici conosciuti in Erasmus, che hanno acceso la mia curiosità e alimentato passione e aspirazioni. Infine menziono me stesso, perché la mia tenace intraprendenza è stata la bussola che mi ha condotto a nuove ed interessanti esperienze.

Jesi, Ottobre 2022

Tommaso Fava

Sommario

L'Internet of Things (IoT) ha avuto negli ultimi anni uno sviluppo decisamente rapido e una delle sue applicazioni più interessanti riguarda lo sviluppo dello Structural Health Monitoring (SHM), grazie al quale è possibile tenere sotto controllo l'integrità delle infrastrutture. Di vitale importanza alla luce degli avvenimenti nella storia recente del nostro Paese, questa tecnologia oggi è ormai semplice da implementare grazie ai dispositivi wireless. Rispettando il vincolo di avere uno scambio dati compatto e veloce e partendo dall'opportunità di ottimizzare la comunicazione attuale attraverso l'uso del formato di serializzazione CBOR (Concise Binary Object Notation), ideato per messaggi di piccola dimensione, e del protocollo MQTT (Message Queuing Telemetry Transport), questo elaborato si concentra sull'idea di garantire la sicurezza durante la trasmissione. In particolare, una volta evidenziate le vulnerabilità di rete e presentato possibili scenari di attacco, viene esaminato il protocollo COSE (CBOR Object Signing and Encryption), responsabile insieme di crittografia e codifica, e si fornisce una sua implementazione in linguaggio Python per la specifica applicazione. Diverse configurazioni vengono valutate in base a dimensione del pacchetto e latenza delle operazioni prima di scegliere l'algoritmo AES-GCM (Advanced Encryption Standard - Galois Counter Mode) con chiave a 256 bit e poi si definisce l'overhead introdotto rispetto al caso non cifrato. Infine, si valutano i vantaggi del binario sul testuale confrontando la soluzione prodotta con un'implementazione identica del protocollo JOSE (JSON Object Signing and Encryption), simile al rivale ma basato sul JSON (JavaScript Object Notation).

Indice

1 Introduzione **1**

2 Formati dati e protocolli di comunicazione **3**
2.1 Serializzazione dati . . . 3
2.1.1 Testuale . . . 4
2.1.2 Binaria . . . 5
2.1.3 Criticità . . . 5
2.2 Formato CBOR . . . 5
2.2.1 Modello di dati . . . 6
2.2.2 Specifiche di codifica . . . 7
2.2.3 Criticità . . . 9
2.3 Protocollo MQTT . . . 9
2.3.1 Architettura . . . 11
2.3.2 Criticità . . . 12
2.4 Sicurezza della comunicazione . . . 13
2.4.1 Scenari di attacco . . . 15
2.4.2 Soluzioni . . . 19
2.5 Protocollo COSE . . . 19
2.5.1 Struttura . . . 20
2.5.2 Chiavi . . . 22
2.6 Algoritmo AES-GCM . . . 23

3 Implementazione del protocollo COSE **27**
3.1 Librerie utilizzate . . . 27
3.2 Firmware . . . 28
3.2.1 Publisher . . . 28
3.2.2 Subscriber . . . 30
3.3 Setup sperimentale . . . 32

4 Risultati **33**
4.1 Confronto tra algoritmi . . . 33
4.2 Overhead introdotto . . . 35
4.3 Confronto tra protocolli COSE e JOSE . . . 35

5 Conclusioni **38**

Elenco delle figure

1.1 Architettura di un sistema di Early Earthquake Warning. 1
1.2 Prototipo di sensore e sua installazione. 2

2.1 Serializzazione e deserializzazione. 3
2.2 Confronto tra XML e JSON. 4
2.3 Rielaborazione dati presentati in [1]. Confronto su 2000 campioni. . 5
2.4 Header di un elemento CBOR. 7
2.5 Rappresentazione binaria dell'intero 12. 7
2.6 Esempio di codifica da JSON a CBOR. 8
2.7 Posizione di MQTT nei livelli ISO-OSI. 10
2.8 Struttura del pacchetto MQTT. 10
2.9 Esempio di infrastruttura MQTT. 11
2.10 Esempio di struttura del topic. 12
2.11 Messaggi scambiati nei vari livelli di QoS. 13
2.12 Protocolli di rete più utilizzati. 14
2.13 Differenti tipi di penetration test. 15
2.14 Risultati della ricerca su Shodan al 21/02/2022. 16
2.15 Primo scenario di attacco. 17
2.16 Secondo scenario di attacco. 17
2.17 Messaggio catturato con Wireshark. 18
2.18 Cattura del pacchetto CONNECT con Wireshark. 18
2.19 (a) Filtro per la modifica e (b) risultato della cattura. 19
2.20 Soluzione adottata. 20
2.21 Tipologie di messaggio COSE, da [2]. 21
2.22 Struttura base di un messaggio COSE. 21
2.23 Struttura di un messaggio COSE_Encrypt0. 22
2.24 Crittografia simmetrica. 24
2.25 Struttura dell'algoritmo AES con chiave a 128 bit. 24
2.26 Funzionamento della modalità operativa ECB. 25
2.27 Struttura della modalità operativa GCM con 2 blocchi. 26

4.1 Latenza con diverse configurazioni di COSE. 34
4.2 Dimensioni di pacchetto iniziale, serializzato e cifrato con A256GCM. 35
4.3 Latenza con JOSE e COSE in configurazione A256GCM. 36
4.4 Pacchetto con JOSE e COSE in configurazione A256GCM. 37

5.1 Cattura Wireshark di pacchetto a) non cifrato e b) cifrato. 39
5.2 Tentativo di decodifica del pacchetto. 39

Elenco delle tabelle

2.1 Principali tipologie di dati. 8
2.2 Informazioni aggiuntive alla codifica. 8
2.3 Corrispondenza tra CBOR e JSON. 9
2.4 Codici di connessione MQTT. 16
2.5 Parametri comuni dell'header di un messaggio COSE. 21
2.6 Parametri comuni di una chiave COSE. 22

5.1 Confronto delle dimensioni di pacchetti con 4000 campioni. 38

Capitolo 1

Introduzione

Il monitoraggio della salute strutturale ha l'obiettivo di fornire, durante ogni momento, informazioni in tempo reale sulle condizioni della struttura in esame [3]. In pratica, l'utilizzo di sensori accelerometri consente di rilevare movimenti anomali, aprendo la strada a interventi di manutenzione o previsione della vita residua. In una zona sismica come quella del Centro Italia, diventa allora quasi banale pensare di replicare tale approccio per cercare di raccogliere quanti più dati possibili in caso di terremoto. Ciò è realizzabile conoscendo che le onde sismiche, prodotte con il rilascio parziale dell'energia accumulate nelle rocce, sono di due tipi, primarie (P) e secondarie (S), con le seconde più lente e catastrofiche delle prime. Nonostante non si possano fare delle previsioni, è però fattibile sfruttare la differente velocità di propagazione (quasi 4 km/s) per generare una allerta prima dell'arrivo delle onde S. Pensata già dopo il terremoto in Messico del 1985 ma concettualizzata in seguito in Giappone e California, questa procedura è nota come Early Earthquake Warning (EEW) e può funzionare perchè le onde elettromagnetiche con cui viene veicolata l'informazione viaggiano molto più veloci di quelle sismiche. Grazie ai sistemi di SHM, il compito che avevano le sole stazioni di controllo oggi può essere affidato ad una densa rete di dispositivi sparsi sul territorio, consentendo una raccolta veloce e precisa.

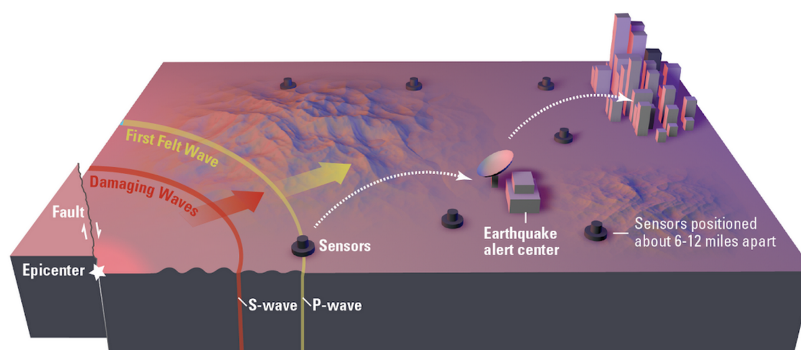

Figura 1.1: Architettura di un sistema di Early Earthquake Warning.

Tuttavia, raggiungere una capillare disposizione dei sensori non è affatto semplice, dati gli elevati costi della strumentazione necessaria o la difficoltà di gestire la comunicazione attraverso le consuete reti di telecomunicazione. In questa ottica,

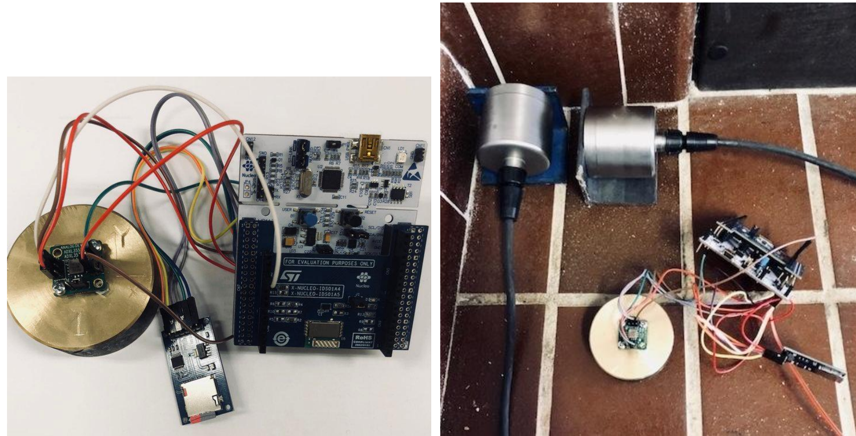

Figura 1.2: Prototipo di sensore e sua installazione.

lo sviluppo delle Wireless Sensor Network (WSN) e dei protocolli IoT assume un ruolo fondamentale: oggi è infatti possibile assemblare componenti economici per costruire dispositivi adeguati e non c'è alcun bisogno di collegamenti via cavo. Il gruppo di ricerca che supervisiona questa tesi ha già intrapreso la via, predisponendo e testando [4] il prototipo mostrato in Figura 1.2. Comparato ad uno cablato ad alte prestazioni in uso al Dipartimento di Ingegneria Civile, Edile e Architettura dell'Università Politecnica delle Marche, il prodotto offre risultati promettenti con solo un decimo della spesa. Ad ogni modo, nel corso del presente elaborato, tale apparecchio e in generale tutto l'hardware verrà ignorato, ricorrendo a simulazioni quando necessario. Ciò perchè qui l'attenzione è rivolta soltanto alla comunicazione, riprendendo un'analisi [1] sull'abbattimento della latenza per sistemi economici e con poche risorse energetiche e computazionali che ha individuato in MQTT [5] un valido sostituto del SeedLink, il protocollo principale per lo scambio di rilevazioni sismiche. Sfortunatamente, questo non supporta la trasmissione dei pacchetti nel formato standard miniSEED ed è perciò necessario ricorrere alla serializzazione. La scelta è ricaduta allora sul CBOR, un derivato del JSON che serializza in binario prediligendo la compattezza alla leggibilità umana. Tenendo fermi questi risultati, la novità introdotta con la presente trattazione riguarda la sicurezza nello scambio dei messaggi. I benefici che il mondo IoT genera devono sempre tenere conto del moltiplicarsi degli attacchi informatici, redditizi e piuttosto facili data la poca attenzione che gli utenti vi prestano. Specialmente per informazioni che impattano sul benessere comune come quelle in esame, è vitale che la comunicazione sia protetta così da evitare che utenti malintenzionati possano accedervi, modificarle e addirittura lanciare allarmi inesistenti [6].

Capitolo 2

Formati dati e protocolli di comunicazione

Questo Capitolo presenta in ordine tutti i concetti fondamentali alla comprensione del tema trattato. Oltre alla descrizione di CBOR, MQTT e COSE si accenna alla serializzazione dei dati e si approfondiscono le possibili vulnerabilità di rete.

2.1 Serializzazione dati

Nei sistemi informatici la serializzazione è un processo utilizzato per la memorizzazione o la trasmissione in rete dell'informazione e consiste nella conversione di un oggetto in un flusso di byte. Lo scopo principale della serializzazione è quello di salvare lo stato di un oggetto per consentirne successivamente la ricreazione, cosa che avviene con il processo inverso chiamato deserializzazione.

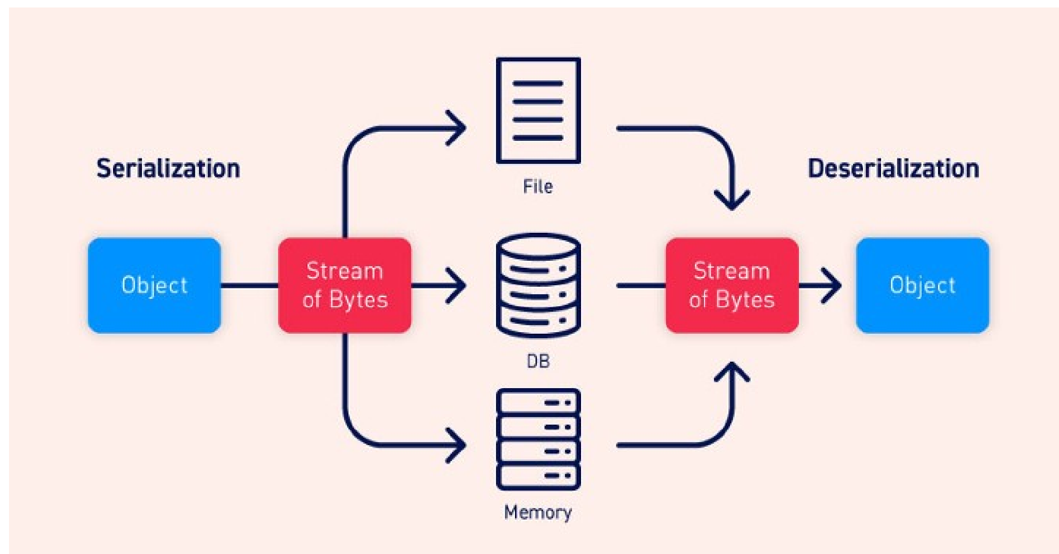

Figura 2.1: Serializzazione e deserializzazione.

Oltre a questo, serializzare i dati è importante anche per altri motivi:

- **Compatibilità:** la serializzazione permette la rappresentazione dei dati in un formato indipendente dall'architettura hardware o software che deve compiere l'elaborazione. Questo significa che l'informazione risulta comprensibile e ricostruibile da tutti.

- **Gestione risorse:** spesso, soprattutto in applicazioni IoT, si ha a che fare con dispositivi che hanno limitate risorse di memoria o rete. Grazie alla serializzazione può essere possibile ridurre la dimensione del payload del messaggio, aspetto che assicura anche il calo di tempo di trasferimento e rischio di perdita dell'informazione.

Esistono due tipi di serializzazione, basata su testo oppure binaria. La prima ha il vantaggio di mantenere la leggibilità umana dell'informazione garantendo allo stesso tempo una struttura compatta ed efficace. Differentemente, la seconda predilige la riduzione della dimensione del flusso dati accettando di contro la perdita di leggibilità.

2.1.1 Testuale

Sono due i principali formati di serializzazione testuale: XML e JSON.

XML
Extensible Markup Language è un formato di testo standardizzato nel 1998 per utilizzi web ma successivamente utilizzato in molteplici contesti grazie alla sua semplicità, flessibilità ed estensibilità. Utilizza elementi ed attributi per organizzare i dati in una struttura gerarchica ad albero.

JSON
JavaScript Object Notation è un formato di interscambio dati leggero e facilmente leggibile che utilizza una struttura composta di coppie chiave-valore. Generalmente, per rappresentare la stessa informazione il JSON richiede meno caratteri dell'XML (vedi Figura 2.2) e per questo motivo è diventato il formato principale per l'utilizzo in ambito web.

XML

```
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40></age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```

JSON

```
{ “empinfo” :
    {
        “employees” : [
        {
            “name” : “James Kirk”,
            “age” : 40,
        },
        {
            “name” : “Jean-Luc Picard”,
            “age” : 45,
        },
        {
            “name” : “Wesley Crusher”,
            “age” : 27,
        }
                    ]
    }
}
```

Figura 2.2: Confronto tra XML e JSON.

2.1.2 Binaria

Di formati di rappresentazione binaria standardizzati ne esistono molti, alcuni pensati per applicazioni specifiche e altri di carattere generale. Tra i più interessanti si trova il CBOR, un formato liberamente basato sul JSON che come questo permette la trasmissione di coppie chiave-valore, ma in maniera più concisa. Definito con la RFC 8949 [7], rappresenta la base della presente trattazione e merita perciò di essere approfondito con dettaglio nella Sezione 2.2. Prima di procedere si tenga a mente che questa serializzazione non è sempre necessariamente migliore delle altre. Ogni caso specifico richiede la sua analisi e la Figura 2.3 spiega perché in [1] si è fatta questa scelta.

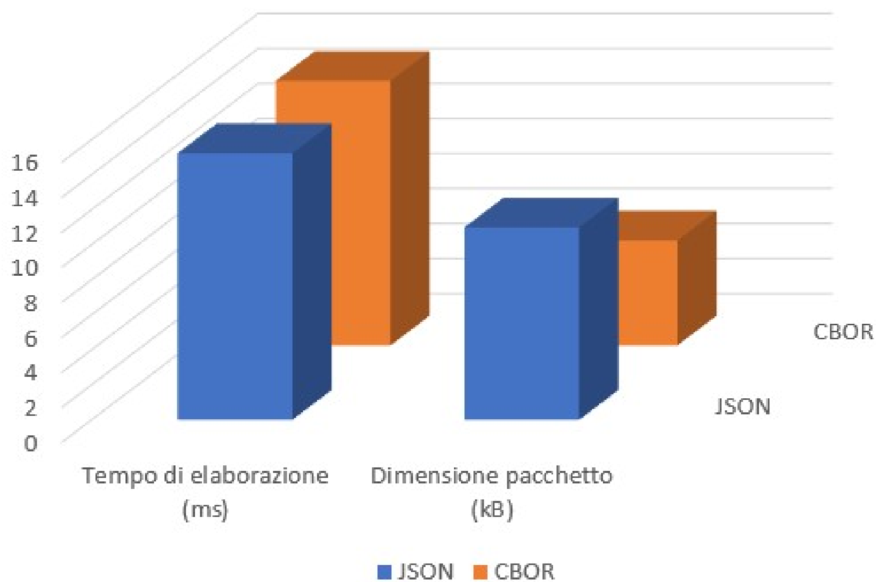

Figura 2.3: Rielaborazione dati presentati in [1]. Confronto su 2000 campioni.

2.1.3 Criticità

La serializzazione può far insorgere problemi di sicurezza perché quando viene eseguita agisce su tutti i membri degli oggetti, inclusi quelli privati. Ciò mina il paradigma dell'incapsulamento dei dati e mette a nudo le strutture interne, lasciando la possibilità di accedere a dati normalmente protetti partendo dal flusso di byte. Per tale ragione spesso gli sviluppatori utilizzano meccanismi di cifratura.

2.2 Formato CBOR

Il Concise Binary Object Representation è un formato dati proposto già nel 2013 dalla Internet Engineering Task Force (IETF) con lo scopo di migliorare le prestazioni

del JSON, mantenendo al contempo la possibilità di trasmettere coppie chiave-valore. Tra gli obiettivi progettuali, in ordine di importanza, ci sono:

1. La rappresentazione deve essere in grado di codificare in modo univoco i più comuni formati dati utilizzati negli standard Internet;
2. Il codice di un codificatore o decodificatore deve essere compatto per supportare sistemi con memoria, potenza del processore e set di istruzioni molto limitati (esempio: nodi di classe 1 definiti in [8]);
3. Deve essere possibile decodificare i dati senza uno schema descrittivo. Come il JSON, il formato deve dunque essere auto-descrittivo;
4. La serializzazione deve essere ragionevolmente compatta, ma la compattezza dei dati è secondaria rispetto a quella del codice di codificatore e decodificatore. Per ragionevole si prende il JSON come limite superiore;
5. Il formato deve essere implementabile in nodi con risorse limitate ma anche in applicazioni che richiedono la codifica di grandi volumi di dati;
6. Il formato deve supportare tutti i tipi di dato JSON così che sia possibile convertire da e verso JSON;
7. Il formato deve essere estensibile consentendo però la decodifica anche a decodificatori precedenti.

2.2.1 Modello di dati

CBOR definisce un suo generico modello di dati che comprende l'insieme di tutti gli elementi rappresentabili. Di base sono presenti i seguenti tipi:

- interi
- valori semplici
- numeri a virgola mobile
- stringhe di byte
- stringhe di testo Unicode
- array
- mappe di coppie chiave-valore
- tag

Questo modello può essere esteso registrando valori semplici e tag. I primi sono identificati da un intero compreso tra 0 e 255 che non rappresenta un numero bensì un tipo (esempio: *false, true, null*), gli altri sono invece elementi composti da identificativo e contenuto che forniscono informazioni aggiuntive per un determinato dato (esempio: stringa che rappresenta una data). E' possibile trovare tutte le loro definizioni in un registro IANA (Internet Assigned Numbers Authority).

2.2.2 Specifiche di codifica

Un elemento CBOR viene codificato in (o decodificato da) una stringa di byte che contiene dati formattati secondo le specifiche illustrate in seguito. Si noti che un codificatore deve produrre solo stringhe ben formate, mentre un decodificatore non deve decodificare un input non ben formato. I requisiti prevedono che ogni stringa abbia un byte iniziale che funge da header e descrive l'elemento codificato.

Figura 2.4: Header di un elemento CBOR.

In Figura 2.4 è riportata la struttura degli 8 bit iniziali, con i primi 3 a identificare una tipologia di dato tra le 8 mostrate in Tabella 2.1 e i restanti 5 che definiscono l'argomento come in Tabella 2.2. La comprensione del suo contenuto dipende però dallo specifico tipo. Per i tipi 0, 1 o 7 se il numero è minore di 24 questo coincide direttamente con l'argomento, che altrimenti indica il numero di byte successivi necessari alla rappresentazione. Nei tipi 2 e 3 si indica la lunghezza in byte delle stringhe, nel tipo 4 il numero di elementi che costituisce l'array mentre nel tipo 5 si parla del numero di coppie della mappa. Diversamente, il tipo 6 ha come informazione aggiuntiva il numero di tag. Un esempio di tipo 0 è proposto in Figura 2.5.

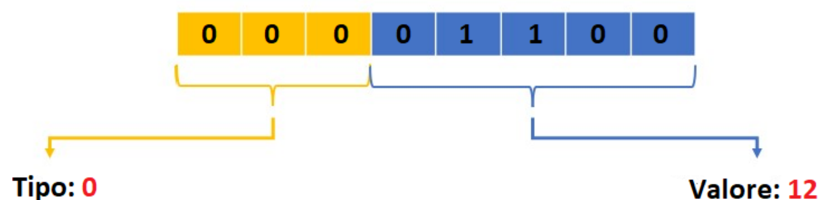

Figura 2.5: Rappresentazione binaria dell'intero 12.

Tipo	Descrizione	Codifica
0	Intero senza segno	000
1	Intero negativo	001
2	Stringa di byte	010
3	Stringa di testo	011
4	Array di elementi	100
5	Mappa di coppie chiave-valore	101
6	Tag	110
7	Numero in virgola mobile e dati semplici speciali	111

Tabella 2.1: Principali tipologie di dati.

Contenuto	Significato
< 24	Argomento contenuto nei 5 bit di informazioni aggiuntive
24	Argomento contenuto in 1 byte successivo
25	Argomento contenuto in 2 byte successivi
26	Argomento contenuto in 4 byte successivi
27	Argomento contenuto in 8 byte successivi
28, 29, 30	Valori riservati a sviluppi futuri
31	Valore indeterminato

Tabella 2.2: Informazioni aggiuntive alla codifica.

Per quel che riguarda il passaggio da e verso JSON, nella definizione del formato CBOR [7] viene proposta una possibile corrispondenza tra i loro tipi, qui sintetizzata in Tabella 2.3. In Figura 2.6 viene invece mostrato un esempio di codifica di un oggetto JSON in CBOR esadecimale. Come previsto, il byte di header codificato A2 corrisponde al binario 101 (tipo 5) 00010 (due coppie). Il motivo per cui si preferisce la rappresentazione esadecimale è la sua compattezza, infatti ogni carattere corrisponde a quattro cifre binarie. Per prendere mano con la conversione è possibile utilizzare il sito interattivo *cbor.me* [9].

```
                              A2                 # map(2)
                                 64              # text(4)
                                    6E6F6D65     # "nome"
                                 67              # text(7)
{                                   546F6D6D61736F # "Tommaso"
  "nome" : "Tommaso",            64              # text(4)
  "anni" : 22                       616E6E69     # "anni"
}                                16              # unsigned(22)
```

Figura 2.6: Esempio di codifica da JSON a CBOR.

CBOR	JSON
Numero intero o a virgola mobile	JSON number
Stringa di testo o byte	JSON string
Array	JSON array
Mappa	JSON object

Tabella 2.3: Corrispondenza tra CBOR e JSON.

2.2.3 Criticità

Un applicazione che si interfaccia alla rete presenta spesso vulnerabilità legate ai dati ricevuti in input, soprattutto se il *parsing* risulta particolarmente complesso. Nella definizione del formato CBOR si è chiaramente scelto di ridurre questa complessità allo scopo di limitare possibili problemi, ma ciò non è sufficiente. Non esiste un modo certo per evitare dati che possano causare errori (*overflow*, *underflow*, ecc...) in modo più o meno intenzionale, ma è possibile rendere abbastanza robusto il decodificatore imponendo che questo accetti soltanto input ben formati, come introdotto nella Sezione 2.2.2. In generale un decodificatore CBOR deve quindi assumere tutti gli ingressi come potenzialmente pericolosi, anche se provenienti da canali sicuri e crittografati. Se dunque non è sufficiente cifrare la comunicazione, questa pratica può portare grandi benefici dal momento che rende possibile assicurare la confidenzialità e l'integrità dell'informazione. La decodifica è infatti l'ultimo step prima dell'applicazione e arriva dopo che il pacchetto ha effettuato un percorso più o meno lungo attraversando la rete e con essa anche le sue possibili minacce. Dovendo prendere decisioni in base a quello che riceve, risulta evidente come l'applicazione debba elaborare dati validi e non manomessi durante la trasmissione. Questo aspetto è di fondamentale importanza per la trattazione e viene approfondito nella Sezione 2.5.

2.3 Protocollo MQTT

Dopo aver discusso della codifica è bene dedicare spazio al protocollo scelto per la trasmissione dell'informazione. Il Message Queuing Telemetry Transport [5] nasce nel 1999 con l'idea di minimizzare l'utilizzo di banda e risorse computazionali ed energetiche, garantendo al contempo affidabilità nella consegna dei pacchetti grazie all'utilizzo del TCP/IP nei livelli inferiori dello stack protocollare. Divenuto nel 2013 standard OASIS (Organization for the Advancement of Structured Information Standards), negli ultimi anni ha avuto una crescita enorme perché l'essere un protocollo leggero e semplice da implementare lo rende perfetto per applicazioni di Internet of Things, specie per dispositivi con risorse limitate o connessi a reti non completamente stabili [10]. Per quanto riguarda le porte, di default utilizza la 1883 ma se si appoggia sui protocolli TLS/SSL per instaurare una comunicazione sicura allora lo si trova alla porta 8883.

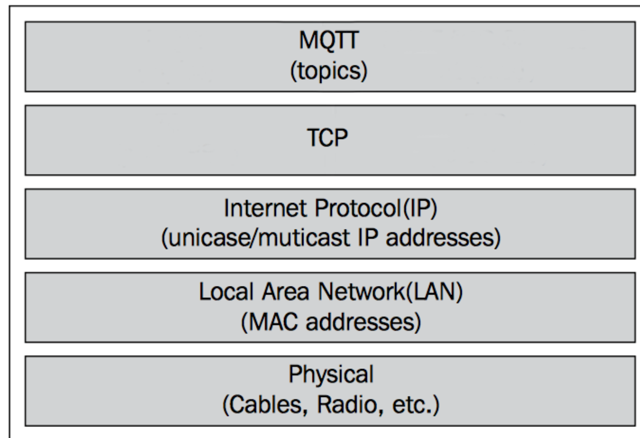

Figura 2.7: Posizione di MQTT nei livelli ISO-OSI.

Il motivo per cui questo è un protocollo particolarmente leggero risiede nella struttura del pacchetto inviato, che prevede un header composto da parte fissa (2 byte) e in alcuni casi da un'altra variabile oltre ad un payload contenente al massimo 256MB di dati, come mostrato in Figura 2.8. Addirittura questo pacchetto può essere ulteriormente ridotto utilizzando una variante chiamata MQTT-SN (MQTT for Sensor Networks), un protocollo pensato per lavorare esclusivamente su reti wireless che sfrutta UDP al posto di TCP. In generale negli anni sono state standardizzate diverse versioni di MQTT (v3.1, v3.1.1, v5, ecc...) e molte sono anche le implementazioni, ma tutte mantengono il modello *publish/subscribe* approfondito più avanti nella Sezione 2.3.1.

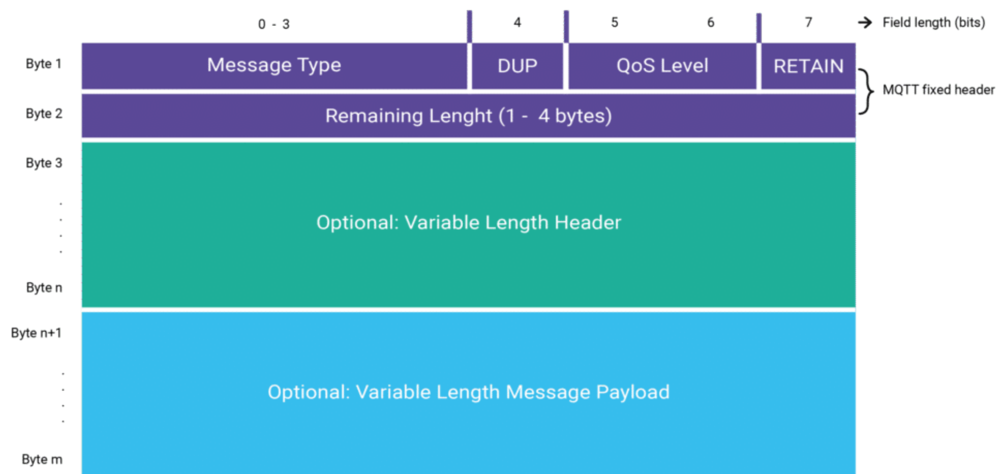

Figura 2.8: Struttura del pacchetto MQTT.

2.3.1 Architettura

L'architettura del protocollo MQTT prevede 3 figure fondamentali:

1. **Publisher:** dispositivo che invia i dati
2. **Broker:** dispositivo che gestisce la distribuzione dell'informazione
3. **Subscriber:** dispositivo che riceve i dati

dove Publisher e Subscriber sono anche detti client in funzione del fatto che stabiliscono una connessione verso il Broker, che si comporta da server. Il modello è quello rappresentato in Figura 2.9 ma per comprendere come avvenga lo scambio dei messaggi bisogna prima definire il topic. Si tratta di una stringa che rappresenta il percorso nel quale viene indirizzato il messaggio pubblicato dal Publisher, che può essere ricevuto soltanto da un Subscriber sottoscritto a quel particolare topic. Proprio come mostrato in Figura 2.10, utilizzando il carattere '/' il topic può essere organizzato in maniera gerarchica offrendo la possibilità di suddividere i dati in livelli diversi. Inoltre, esistono due cosiddette *wildcards* attraverso le quali il Subscriber può decidere di iscriversi a più di un solo topic. La principale, rappresentata da '#', permette di accedere a tutti i topic che stanno sotto un determinato livello. Quando un client trasmette un messaggio, questo viaggia verso il server che conoscendo la lista di chi vuole riceverlo si occupa di inoltrarlo. In questo modo client diversi non hanno alcun bisogno di conoscersi o essere sincronizzati e ciò assume una grande importanza per quel che riguarda la scalabilità. In aggiunta, se richiesto dal Publisher settando un flag (RETAIN), il Broker ha la possibilità di mantenere al suo interno messaggi che devono essere letti da Subscriber disconnessi in quel particolare momento. Tale funzione è ottima per un utilizzo su rete instabile, ma deve essere gestita con attenzione perché se non previsto un tempo massimo il messaggio resta indefinitamente conservato alterando le prestazioni del dispositivo.

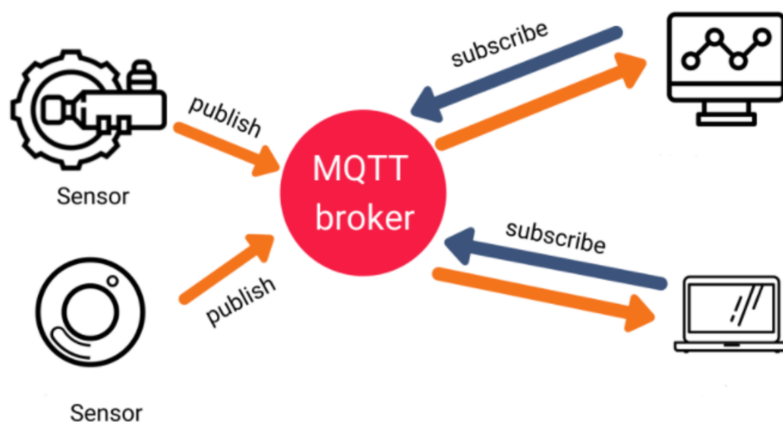

Figura 2.9: Esempio di infrastruttura MQTT.

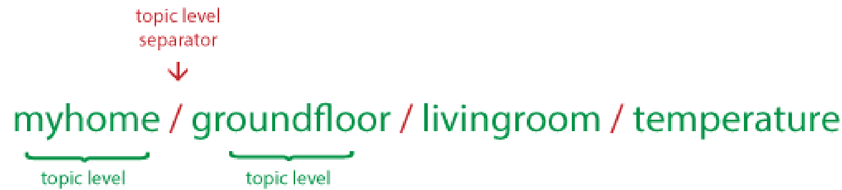

Figura 2.10: Esempio di struttura del topic.

MQTT definisce tre livelli di Quality of Service (QoS) per la distribuzione dei messaggi:

- **QoS = 0:** il messaggio viene inviato al massimo una volta e non è prevista la garanzia della ricezione. (*At most once*)
- **QoS = 1:** il messaggio viene inviato più volte ad intervalli regolari finché non viene ricevuto. Può essere ricevuto più volte. (*At least once*)
- **QoS = 2:** il messaggio viene inviato e ricevuto una sola volta. (*Exactly once*)

Come si nota dalla Figura 2.11, ogni livello di QoS prevede lo scambio di un diverso numero di pacchetti. Se il QoS = 0 prevede soltanto la pubblicazione (PUBLISH), già con QoS = 1 per ogni messaggio pubblicato è necessario scambiare due pacchetti. Questo perché il client attende un acknowledgment del Broker (PUBACK) per cancellare l'informazione dalla coda di uscita, e se non la riceve attiva il flag DUP (Duplicate) e invia più volte lo stesso messaggio finché non arriva conferma. La procedura rallenta ancora con QoS = 2, che richiede 4 pacchetti scambiati. In questo caso quando il server riceve il messaggio non risponde ad entrambi i client contemporaneamente. Prima di inoltrare il messaggio al Subscriber chiede infatti al Publisher (PUBREC) se può procedere, e dopo aver ricevuto conferma (PUBREL) avvisa dell'avvenuta consegna (PUBCOMP) permettendo al mittente di liberare la coda. Evidentemente, la scelta di garantire una migliore affidabilità della comunicazione ha un grande impatto su ritardo end-to-end [11] e sul consumo energetico [12], e ciò impone una attenta valutazione soprattutto nel caso di applicazioni su dispositivi dalle risorse limitate. Inoltre, in [13] viene anche spiegato come attraverso la trasmissione di messaggi con QoS alto sia possibile lanciare attacchi DoS (Denial of Service) [14] verso l'infrastruttura di rete. Questo ultimo aspetto verrà ripreso nella Sezione 2.4.

2.3.2 Criticità

Sebbene siano molti i pregi di questo protocollo, MQTT non è esente da difetti. In [13] è presente la lista di quelli principali, che include:

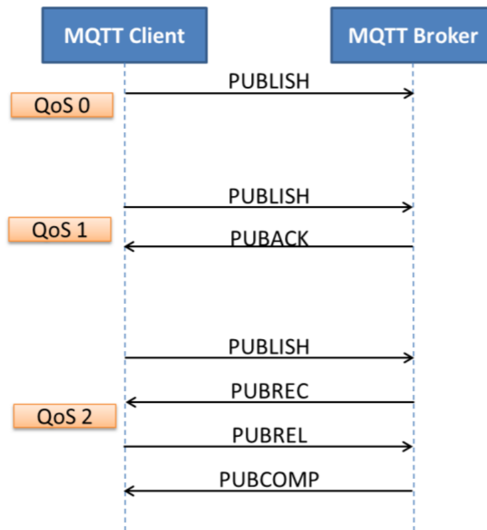

Figura 2.11: Messaggi scambiati nei vari livelli di QoS.

1. Ogni client può partecipare a qualunque topic. Non è prevista una tecnica di verifica che permetta al Broker di controllare gli accessi.
2. Se un Subscriber si dimentica di raccogliere il messaggio, questo resta nel Broker per un tempo indeterminato riducendo le sue performance.
3. Non è possibile dare priorità ad alcuni messaggi.
4. Ordinare e inviare messaggi che sono stati persi è difficile.

Fortunatamente esistono contromisure per rimediare ad alcuni di essi. Può essere implementato un controllo sugli utenti attraverso username e password oppure inserito un *timestamp* per gestire il tempo massimo in cui il Broker deve mantenere un determinato messaggio. Le soluzioni sono molte ma spesso le lacune non vengono colmate per mancanza di consapevolezza. Nella Sezione 2.4 si indaga il comportamento di MQTT dal punto di vista della sicurezza al fine di dimostrare quanto possa essere facile attaccare un protocollo che giorno dopo giorno gestisce sempre più dispositivi connessi.

2.4 Sicurezza della comunicazione

L'Internet of Things è la nuova frontiera della comunicazione [15] e secondo una previsione della IDC [16] entro il 2025 ci saranno 79ZB di dati generati da miliardi di dispositivi connessi. Per gli analisti di McKinsey&Company [17] ogni secondo ne vengono collegati 127 e questi numeri vanno di pari passo con la crescita degli attacchi informatici. In [12] è riportato che in media un dispositivo viene attaccato già dopo 5 minuti, mentre Kaspersky rivela che nella sola prima metà del 2021 sono state

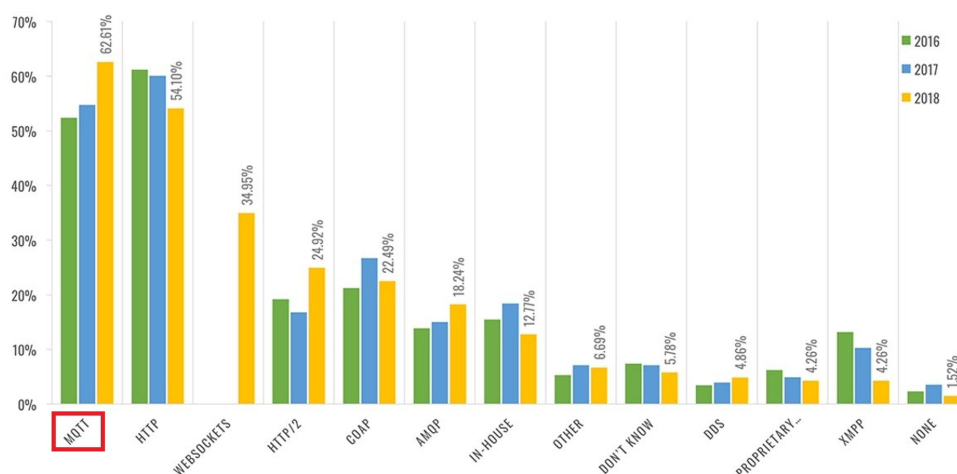

Figura 2.12: Protocolli di rete più utilizzati.

1.51 miliardi le violazione accertate, un numero doppio rispetto all'anno precedente. L'obiettivo più comune è quello di lanciare attacchi DoS [14] o prendere possesso di informazioni confidenziali, mentre le cause, così come le conseguenze, sono di ogni tipo. Principalmente va sottolineato come il crescente interesse degli utenti verso le nuove tecnologie non sia accompagnato dalla giusta consapevolezza riguardo il tema della sicurezza. Molte persone credono di non essere abbastanza importanti da essere attaccate e tendono a sottovalutare il pericolo. Oltre a ciò, in molti casi anche aziende e sviluppatori non prestano la dovuta attenzione alla tematica. Secondo gli autori di [18] la sicurezza è infatti il frutto del compromesso tra livello di protezione e grado di usabilità, che spesso prevale sul primo. Va però aggiunto infine che anche con le migliori intenzioni non è possibile progettare un sistema sicuro al 100%, la sua robustezza è sufficiente solo finché un attaccante non trova il modo di entrare. In seguito si indagano le vulnerabilità del protocollo MQTT, che oltre ad essere la scelta per la nostra applicazione è anche il più utilizzato negli ultimi anni, come si può notare in Figura 2.12. Prima di procedere bisogna discutere dei motivi per cui esistano tali vulnerabilità. In [19] si evidenziano tre ragioni fondamentali:

1. Utilizzo su dispositivi con risorse troppo limitate da supportare adeguati meccanismi di sicurezza.
2. Mancanza di consapevolezza che porta gli sviluppatori a preferire altre funzionalità e gli utenti a non agire correttamente.
3. Difficoltà di gestione di un vasto numero di oggetti diversi connessi in rete.

Tra questi il primo è sicuramente quello principale. Nella Sezione 2.3 è stato spiegato come uno dei vantaggi principali di MQTT sia quello di richiedere minime risorse computazionali permettendo così l'utilizzo dei dispositivi meno potenti [8], ma questa configurazione porta con sé delle pesanti conseguenze. Sui nodi più semplici

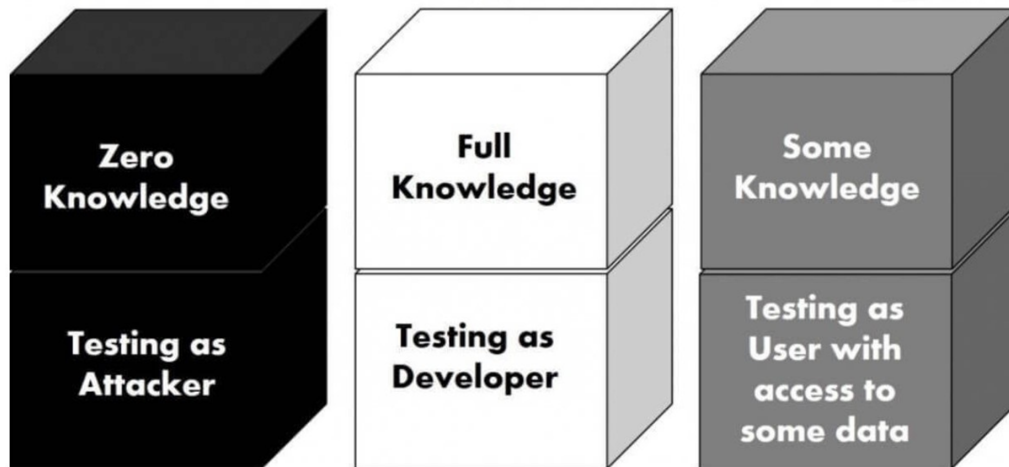

Figura 2.13: Differenti tipi di penetration test.

può infatti non essere possibile, per questioni di elaborazione [12], implementare il protocollo crittografico TLS (Transport Layer Security) [20], una soluzione che consente la creazione di un canale sicuro tra client e server per applicazioni costruite sopra TCP/IP, proprio come quelle che usano MQTT. Questo protocollo si occupa di oscurare il payload e impedire che la sua rappresentazione in chiaro (*plaintext*) incentivi gli attacchi di tipo MITM (Man-In-The-Middle) [21], dove un utente malevolo intercetta e in alcuni casi modifica l'informazione. Volendo indagare le vulnerabilità nel modo più generale possibile, nella Sezione 2.4.1 ci si concentra su implementazioni di MQTT senza TLS per mostrare i possibili scenari di attacco. L'utilizzo sopra TLS, nonostante non sia nei nostri scopi, viene presentato per completezza nella Sezione 2.4.2.

2.4.1 Scenari di attacco

Questa Sezione riprende l'analisi effettuata in [13, 19] per discutere di come sia possibile lanciare un attacco verso il protocollo MQTT. L'esperimento fatto consiste in un test di penetrazione *black box*, dove si assume di non conoscere nulla del sistema da attaccare: nessuna informazione sull'infrastruttura di rete, sui meccanismi di difesa o sui canali di comunicazione. Questa metodologia assicura la massima generalità e impone la raccolta di alcune informazioni. Ciò è stato fatto utilizzando Shodan [22], un motore di ricerca per l'Internet of Things che indicizza i dispositivi connessi alla rete. Inserendo nell'area di ricerca la stringa 'port:1883 MQTT'; è infatti possibile individuare Broker MQTT che utilizzano la porta 1883, quella di default per protocolli non basati sulla sicurezza del TLS. Questi sono i più semplici da attaccare e purtroppo (o per fortuna, per un hacker malevolo) rappresentano il 99.78% del totale al momento di questa stesura (vedi Figura 2.14).

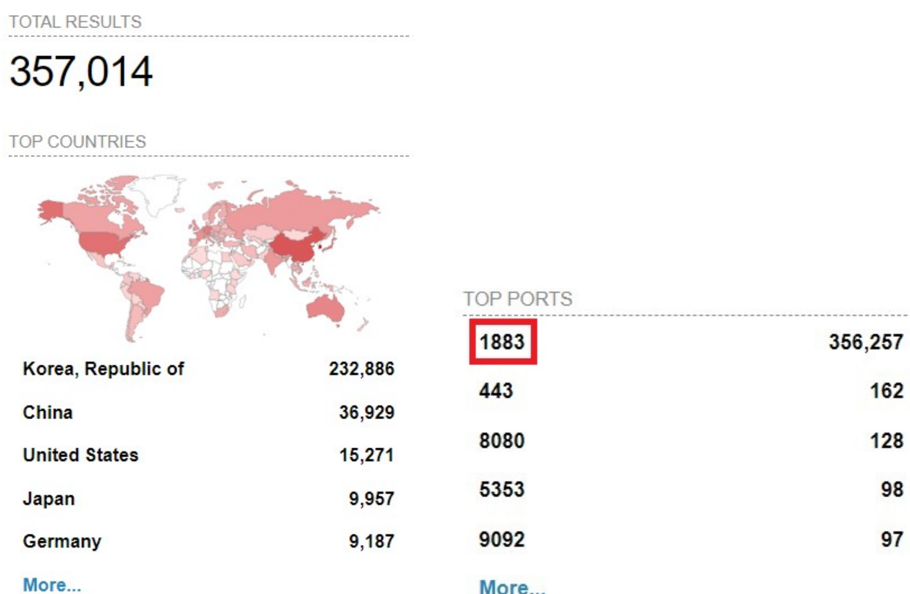

Figura 2.14: Risultati della ricerca su Shodan al 21/02/2022.

Nell'elenco dei risultati viene riportato anche il *connection code*, un parametro molto importante che ci segnala se il Broker è provvisto oppure no di una procedura di autenticazione e autorizzazione. Questi codici sono parte dell'header del pacchetto CONNACK, inviato dal server al client in risposta alla richiesta di connessione (CONNECT), e possono assumere i valori riportati in Tabella 2.4. La grande maggioranza riporta il codice 0, indice della possibilità di diventare Publisher o Subscriber senza alcun controllo. Detto ciò, vedremo che in alcuni casi sarà comunque agevole superare il problema della mancanza di credenziali semplicemente intercettando il traffico. Si affrontano finalmente ora i possibili scenari di attacco, differenti in base al fatto che ci si trovi sulla stessa rete della vittima (local network) oppure su un'altra (public network).

Scenario 1 - possibile su tutte le reti

In questo primo scenario di attacco si sceglie un Broker con codice 0 e si fa la sottoscrizione a tutti i topic attraverso la *wildcard* '#'. Diventa così possibile ascoltare

Codice	Descrizione
0	Connessione accettata
1	Connessione rifiutata per versione protocollo non disponibile
2	Connessione rifiutata per client non abilitato
3	Connessione rifiutata per server non disponibile
4	Connessione rifiutata per credenziali errate
5	Connessione rifiutata per mancanza di autorizzazione
6-255	Riservato a sviluppi futuri

Tabella 2.4: Codici di connessione MQTT.

tutte le conversazioni e venire in possesso di alcune informazioni confidenziali o utili per fasi di attacco successive.

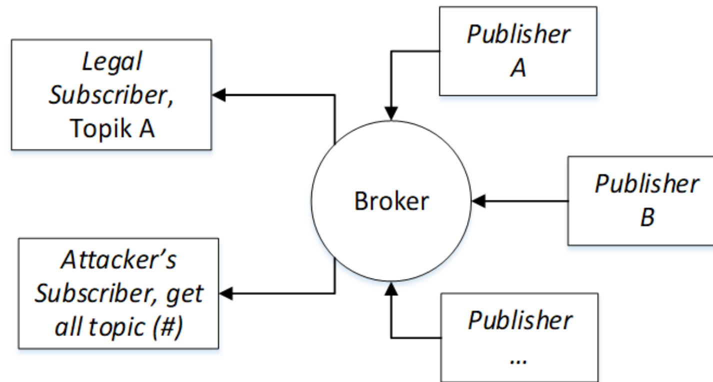

Figura 2.15: Primo scenario di attacco.

Scenario 2 - possibile su tutte le reti

Come il precedente, anche questo scenario si applica ad architetture senza richiesta di autenticazione. Consiste nella pubblicazione di messaggi volti soprattutto a stremare le risorse computazionali per creare disservizi di rete (attacco DoS) ed è fattibile senza particolare ingegno semplicemente utilizzando i QoS più alti [13], come introdotto in precedenza. Una alternativa può essere invece quella di prendere il controllo di infrastrutture critiche (esempio: illuminazione stradale) inserendo soltanto una iniziale fase di ascolto (scenario 1) in cui analizzare i pacchetti e trovare la corretta sintassi richiesta dal messaggio.

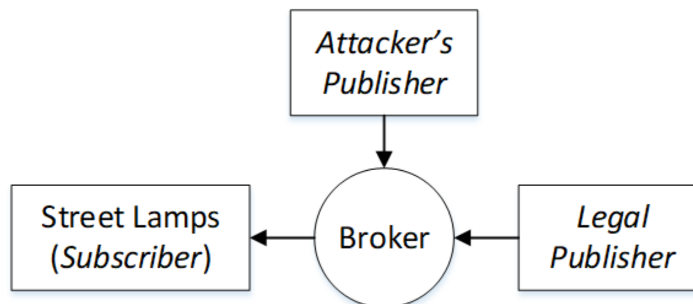

Figura 2.16: Secondo scenario di attacco.

Scenario 3 - attaccante e vittima sulla stessa rete

Il terzo e ultimo scenario richiede la connessione alla stessa rete, una condizione più stringente attraverso la quale è però possibile analizzare il traffico di rete (*sniffing*) e portare a compimento un attacco anche su Broker che implementano autenticazione. Grazie allo *sniffing* è inoltre possibile applicare tale scenario quando la porta utilizzata non è quella di default (condizione di *port obscurity*), perché il suo numero viene inserito nell'header dei pacchetti. In [19] vengono usati i software Wireshark ed

Ettercap per dimostrare come sia fattibile minare la confidenzialità, l'autenticità e l'integrità della comunicazione.

- **Confidenzialità**

  La confidenzialità del messaggio MQTT è assolutamente un problema perché di default il protocollo non prevede alcuna cifratura. Catturando con Wireshark un pacchetto è possibile leggere topic e messaggio come in Figura 2.17.

  ```
  MQ Telemetry Transport Protocol
    Publish Message
      0011 0000 = Header Flags: 0x30 (Publish Message)
      Msg Len: 21
      Topic: outTopic
      Message: hello world
  ```

  Figura 2.17: Messaggio catturato con Wireshark.

- **Autenticità**

  Se il Broker prevede un meccanismo di autenticazione tramite username e password, l'attaccante non può agire da Publisher o Subscriber finché non conosce le credenziali (in questo caso il codice di connessione sarà 4 oppure 5). Nel caso di questo scenario è però molto semplice venirne in possesso perchè username e password vengono trasmessi in chiaro nei pacchetti CONNECT, inviati periodicamente al termine del timer *KeepAlive* contenuto nell'header del messaggio. Intercettando il traffico si ha la situazione in Figura 2.18.

  ```
  MQ Telemetry Transport Protocol
    Connect Command
      0001 0000 = Header Flags: 0x10 (Connect Command)
      Msg Len: 42
      Protocol Name: MQTT
      Version: 4
      1100 0010 = Connect Flags: 0xc2
      Keep Alive: 15
      Client ID: ESP8266Client-3f03
      User Name: ipul
      Password: ipul
  ```

  Figura 2.18: Cattura del pacchetto CONNECT con Wireshark.

- **Integrità**

  Un altro possibile attacco è quello che ha l'obiettivo di modificare i dati in transito e per metterlo in atto è sufficiente creare un filtro come in Figura 2.19a. In Figura 2.19b viene invece mostrato il risultato ottenuto. In particolare, se il messaggio intercettato contenesse un link (magari un download di aggiornamento del firmware) sarebbe possibile cambiarlo per installare un software malevolo

che ci restituisca il controllo dei dispositivi. Questa situazione corrisponde alla creazione di una rete conosciuta come *Botnet* [23].

```
#owned.filter
if (ip.proto == TCP && tcp.dst == 1883 && ip.dst == 'IP Broker' &&
search(DATA.data, "outTopic")) {
  replace("outTopic", "outTopuc");
  msg("payload replaced\n");
}
```

```
Transmission Control Protocol, Src Port: 1883, Dst Port: 28830, Se
MQ Telemetry Transport Protocol
  Publish Message
    0011 0000 = Header Flags: 0x30 (Publish Message)
    Msg Len: 25
    Topic: outTopuc
    Message: hello world #31
```

Figura 2.19: (a) Filtro per la modifica e (b) risultato della cattura.

2.4.2 Soluzioni

Se i primi due scenari possono essere scongiurati semplicemente utilizzando meccanismi di autenticazione ormai presenti su tutte le implementazioni dei Broker, per il terzo questo non è sufficiente. Se si è connessi alla stessa rete, l'unico modo per impedire che chiunque acceda alle informazioni è la creazione di un canale sicuro tra i due punti. Questa funzionalità è solitamente assente nelle applicazioni MQTT ma può essere ottenuta inviando dati cifrati oppure utilizzando il protocollo TLS, scelta però non sempre possibile per via della capacità computazionali. Inoltre, tale opzione prevede un canale sicuro solo tra client e server e quindi al Broker è consentito di leggere tutti i messaggi. Alla luce di tutto ciò, la soluzione scelta e presentata nel Capitolo 3 consiste nel cifrare i pacchetti prima della loro trasmissione, come mostrato in Figura 2.20. Questo permette a tutti di intercettare la comunicazione, ma soltanto il ricevitore autorizzato possiede la chiave e può decifrare l'informazione. Qualora si volesse comunque procedere con TLS, si tenga a mente che la procedura si basa su un *handshake* in cui il server accoglie la richiesta di connessione del client, invia il suo certificato e insieme concordano una chiave simmetrica con cui cifrare la comunicazione attraverso l'algoritmo di Diffie-Hellman [20].

2.5 Protocollo COSE

Risulta dunque fondamentale trovare il modo di cifrare i dati e codificarli in CBOR prima di trasmetterli, tuttavia tali passaggi non possono essere eseguiti separatamente per i seguenti motivi:

Figura 2.20: Soluzione adottata.

- serializzare dopo la cifratura non è possibile
- cifrare dopo la serializzazione cambia il formato dei dati

Questo problema è risolto dal CBOR Object Signing and Encryption [2], un protocollo ad hoc presentato nel 2017 che si occupa di crittografia, codici MAC e firme. La struttura di un pacchetto COSE viene presentata nella Sezione 2.5.1, dove si tratta brevemente di come avvenga la cifratura e si tralasciano tutti gli aspetti non fondamentali per comprendere il lavoro svolto. La stessa logica è adottata anche in seguito, laddove l'unico altro argomento descritto è la gestione delle chiavi crittografiche presentata nella Sezione 2.5.2. Tutto il resto può essere approfondito direttamente in [2].

2.5.1 Struttura

Tutti i messaggi COSE sono codificati come CBOR array (tipo 5 da Tabella 2.1). L'insieme delle tipologie implementabili è raccolto in Figura 2.21, mentre nella 2.22 viene riportata la struttura base di un messaggio. Al fine di ridurre l'ammontare del codice necessario all'elaborazione, i primi tre oggetti presenti in ogni messaggio contengono le medesime informazioni, mentre i seguenti variano a seconda del tipo utilizzato (esempio: informazioni sulla firma). Questi campi fondamentali sono così suddivisi:

- **Protected header parameters:** parametri che devono essere crittograficamente protetti perché utili alla decifrazione. Se presenti vengono codificati in una stringa di byte CBOR (tipo 2).
- **Unprotected header parameters:** parametri che non necessitano di protezione. Se presenti vengono codificati in una mappa CBOR (tipo 5).
- **Message content:** questo campo contiene il payload del messaggio, che può essere *plaintext* o *ciphertext* a seconda del tipo scelto.

I più comuni parametri dell'header sono menzionati e descritti in Tabella 2.5. Questi valgono qualunque sia la struttura utilizzata, che nel caso in esame, cercando soltanto

CBOR Tag	cose-type	Data Item	Semantics
98	cose-sign	COSE_Sign	COSE Signed Data Object
18	cose-sign1	COSE_Sign1	COSE Single Signer Data Object
96	cose-encrypt	COSE_Encrypt	COSE Encrypted Data Object
16	cose-encrypt0	COSE_Encrypt0	COSE Single Recipient Encrypted Data Object
97	cose-mac	COSE_Mac	COSE MACed Data Object
17	cose-mac0	COSE_Mac0	COSE Mac w/o Recipients Object

Figura 2.21: Tipologie di messaggio COSE, da [2].

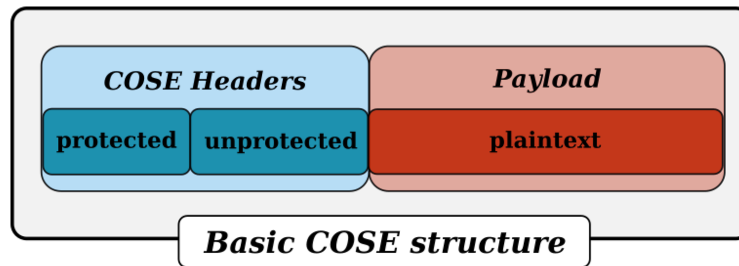

Figura 2.22: Struttura base di un messaggio COSE.

la cifratura, può essere *COSE_Encrypt* oppure *COSE_Encrypt0*. La prima si usa se ci sono più ricevitori o se deve essere scambiata la chiave ma per il progetto in esame è sufficiente la seconda. Dalla Figura 2.23 è possibile notare come il protocollo specifichi che oltre alla confidenzialità debba essere garantita anche l'autenticazione del contenuto. Tale decisione restringe la lista degli algoritmi leciti ad AES-GCM [24], AES-CCM [25] e ChaCha20/Poly1305 [26] e tra loro è stato scelto il primo con chiave a 256 bit sulla base dei risultati presentati successivamente nel Capitolo 4. I parametri devono essere adeguati di conseguenza:

- *alg* deve essere settato su 'A256GCM' (stabilito da registro IANA)
- *IV* deve essere un vettore lungo 12 byte (per raccomandazione del NIST)

Nome	Descrizione	Posizione
alg	Algoritmo crittografico utilizzato	Protected header
crit	Parametri critici dell'header	Protected header
content type	Tipologia di dato contenuto	Unprotected header
kid	Dati utili per derivare la chiave	Unprotected header
IV	Vettore di inizializzazione	Unprotected header
Partial IV	Parte del vettore di inizializzazione	Unprotected header

Tabella 2.5: Parametri comuni dell'header di un messaggio COSE.

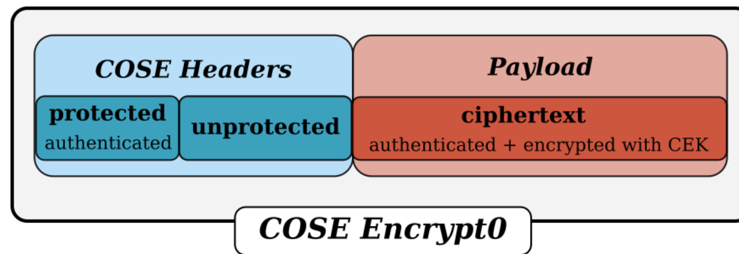

Figura 2.23: Struttura di un messaggio COSE_Encrypt0.

Una descrizione approfondita dell'algoritmo si trova nella Sezione 2.6, ma prima bisogna discutere di come il protocollo gestisca le chiavi.

2.5.2 Chiavi

Le informazioni relative alle chiavi necessarie per cifrare il messaggio sono contenute in una *COSE_Key*, che ha la struttura di una mappa CBOR e contiene alcuni tra i diversi parametri illustrati in Tabella 2.6. La scelta del particolare algoritmo impone i valori che questi possono assumere e se non c'è adeguata corrispondenza la computazione non deve avvenire. Nel caso in esame bisogna controllare che:

- *alg*, se presente, sia settato su 'A256GCM'
- *kty* sia presente e riporti la voce 'Symmetric'
- *key_ops*, se presente, includa 'encrypt' e 'decrypt'

Insieme a queste informazioni deve ovviamente esserci anche la chiave stessa. Esistono vari metodi per generarla e trasmetterla ma considerato che deve essere nota a entrambi i dispositivi si è scelto di utilizzarne una fissa comunicata a priori. Affidarsi alla stessa chiave per cifrare un gran numero di messaggi diversi impatta negativamente sul livello di sicurezza ma ciò non è stato ritenuto fondamentale per la presente trattazione.

Nome	Descrizione
kty	Famiglia di chiavi utilizzata
kid	Dati utili per derivare la chiave. Riprende quello nel messaggio
alg	Algoritmo crittografico utilizzato
key_ops	Set di operazioni consentite con la chiave
Base IV	Porzione base dell'IV. Va utilizzato insieme al Partial IV

Tabella 2.6: Parametri comuni di una chiave COSE.

2.6 Algoritmo AES-GCM

Tra gli scopi di questo elaborato c'è anche la volontà di permettere al lettore una comprensione senza eccessivi prerequisiti. Per tale motivo, procedere con la presentazione del codice implementato senza prima discutere dell'algoritmo crittografico utilizzato sarebbe impossibile. D'altro canto questa è una materia piuttosto complicata e sviscerarla nel dettaglio richiederebbe uno sforzo che esula dalle nostre intenzioni. Si è quindi deciso di evitare la discussione matematica e fornire una panoramica generale che parte dalla crittografia simmetrica e passa per l'algoritmo AES e la modalità operativa GCM.

Crittografia simmetrica
Per crittografia simmetrica si intende una tecnica di cifratura in cui la chiave d utilizzata per decifrare è la stessa di e usata per cifrare:

$$e = d = k \tag{2.1}$$

Questo rende l'algoritmo di cifratura molto performante e semplice da implementare, tuttavia presuppone che le due parti siano già in possesso della chiave. Se così non fosse esistono procedure per lo scambio di chiavi basate sulla crittografia asimmetrica come quella introdotta nella Sezione 2.4.2. Una volta in possesso delle chiavi la funzione E scelta prende il messaggio P e la chiave k per restituire il *ciphertext* C:

$$E(P, k) = C \tag{2.2}$$

Per risalire al *plaintext* basta applicare la funzione inversa:

$$E^{-1}(C, k) = P \tag{2.3}$$

Se l'attaccante intercetta la trasmissione non ha modo di risalire al messaggio in chiaro perchè non è in possesso della chiave. Esistono però molteplici possibili attacchi, tra cui quelli a forza bruta, e la sicurezza dell'algoritmo dipende soprattutto dalla dimensione utilizzata per la chiave. Oggi si ritiene che la lunghezza minima per garantire protezione con la crittografia simmetrica sia di 128 bit ma l'avvento dei computer quantistici e della loro capacità computazionale porta la soglia sempre più in alto.

Algoritmo AES
AES è un algoritmo di cifratura a blocchi a chiave simmetrica prodotto per sostituire il non più sicuro DES (Data Encryption Standard) e utilizzato come standard dal governo USA. Per cifratura a blocchi (*block cipher*) si intende che l'algoritmo funziona su un gruppo di bit di lunghezza finita (128 bit in questo caso) organizzati in un blocco. A differenza di quelli a flusso (*stream cipher*) che cifrano un bit alla volta, ogni elemento nel blocco viene criptato contemporaneamente. L'Advanced Encryption Standard è una rete a sostituzione e permutazione in cui semplici operazioni

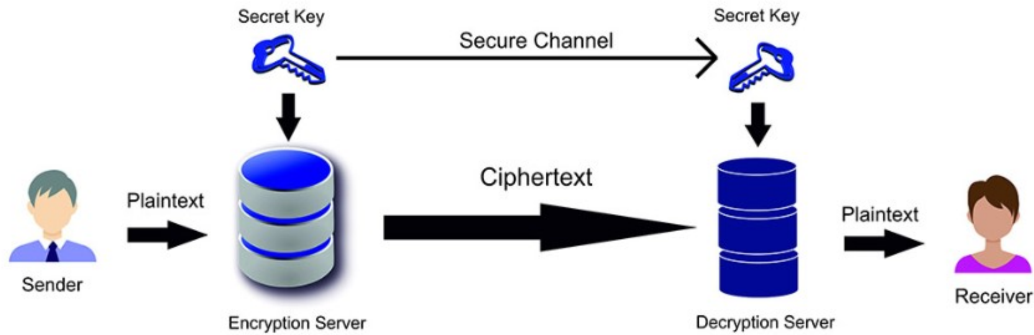

Figura 2.24: Crittografia simmetrica.

matematiche ripetute in sequenza permettono di aggiungere confusione e diffusione ai dati in ingresso. E' strutturato per lavorare con matrici 4x4 che vengono modificate in diversi round (10, 12 o 14 con chiavi rispettivamente a 128, 192 o 256 bit) composti ognuno di 4 passaggi:

1. **SubBytes:** sostituzione non lineare di tutti i byte che vengono rimpiazzati secondo una specifica tabella.
2. **ShiftRows:** spostamento dei byte di un certo numero di posizioni dipendente dalla riga di appartenenza.
3. **MixColumns:** combinazione dei byte con un'operazione lineare. I byte vengono trattati una colonna per volta.
4. **AddRoundKey:** ogni byte della tabella viene combinato con la chiave di sessione, calcolata da quella principale.

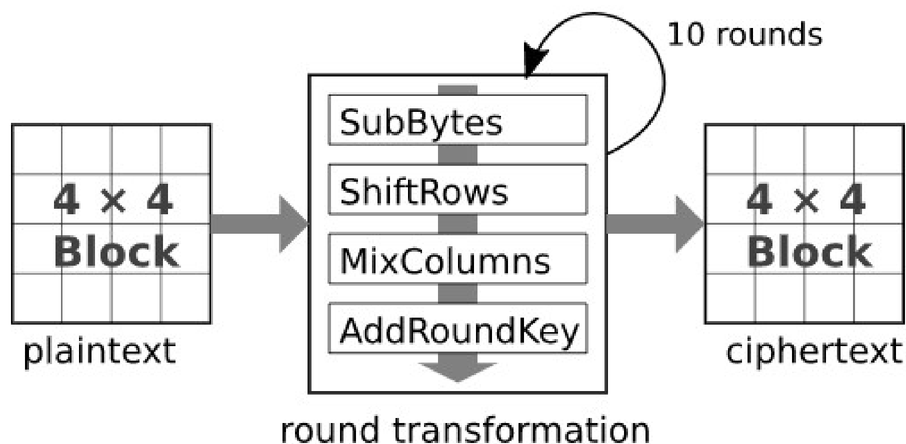

Figura 2.25: Struttura dell'algoritmo AES con chiave a 128 bit.

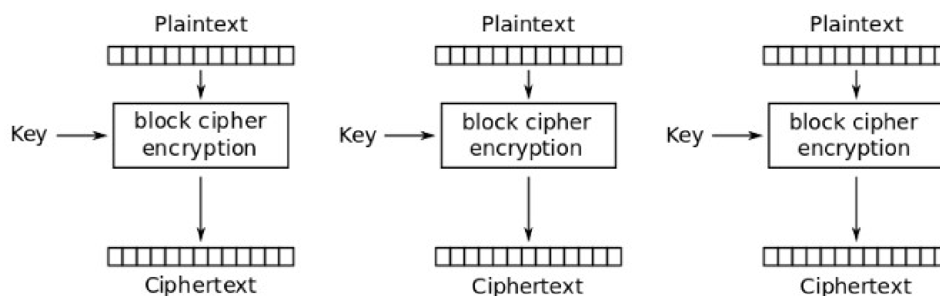

Figura 2.26: Funzionamento della modalità operativa ECB.

Modalità GCM

Un cifrario a blocchi è sufficiente soltanto per l'elaborazione di un singolo blocco di lunghezza finita. Quando si ha a che fare con dati più grandi di tale dimensione esistono delle modalità operative che gestiscono la procedura, con la più semplice mostrata in Figura 2.26. In tale modalità ogni blocco viene cifrato indipendentemente dagli altri e ciò impone il grande svantaggio di avere lo stesso *ciphertext* tutte le volte che si usino medesimi *plaintext* e chiave. Questo problema è stato risolto introducendo nuove modalità operative proprio come GCM, che a differenza di altre permette anche l'autenticazione del messaggio. Approfondire la sua struttura richiede impegnative basi matematiche, inclusa la teoria dei campi finiti di cui Galois è un esempio. Pertanto di seguito verranno forniti solo i dettagli essenziali. La Figura 2.27 spiega quale sia la sequenza di passi da compiere nel caso di due blocchi ma prima di procedere è bene fare chiarezza sulla notazione:

- **IV:** il vettore di inizializzazione è una sequenza binaria di lunghezza predefinita che viene utilizzata per dare il via all'elaborazione. Inserito in punti diversi in base alla modalità operativa scelta, è necessario affinché cifrando messaggi identici con la stessa chiave non si abbia lo stesso output.

- **Auth Data 1:** questo blocco rappresenta una parte del messaggio che deve essere autenticata ma non cifrata. E' il caso di informazioni quali indirizzo IP o porte di rete.

- **Auth Tag:** il tag rappresenta il codice MAC che deve essere confrontato con quello calcolato dal destinatario per valutare l'autenticità del messaggio ricevuto.

La procedura è la seguente:

1. Il vettore IV viene concatenato ad un contatore che parte da zero e il risultante viene cifrato con chiave k secondo l'algoritmo scelto. L'output viene messo da parte e utilizzato alla fine per generare il codice MAC.

2. L'input del passo precedente viene incrementato e di nuovo cifrato. Il *ciphertext* è ottenuto dall'operazione XOR tra il risultato della cifratura e il relativo *plaintext*. Questo passaggio viene ripetuto per ogni blocco.

3. La parte di dati da autenticare viene elaborata in una operazione detta moltiplicazione con chiave *H*. Lo XOR tra ciò che si ottiene e il primo blocco cifrato viene di nuovo moltiplicato e poi inviato ai blocchi successivi.

4. L'uscita dell'ultimo blocco subisce uno XOR con la concatenazione tra le lunghezze di testo cifrato e dati da autenticare. Il risultato viene moltiplicato e poi operato secondo XOR con l'output del passo 1 per ottenere il codice MAC.

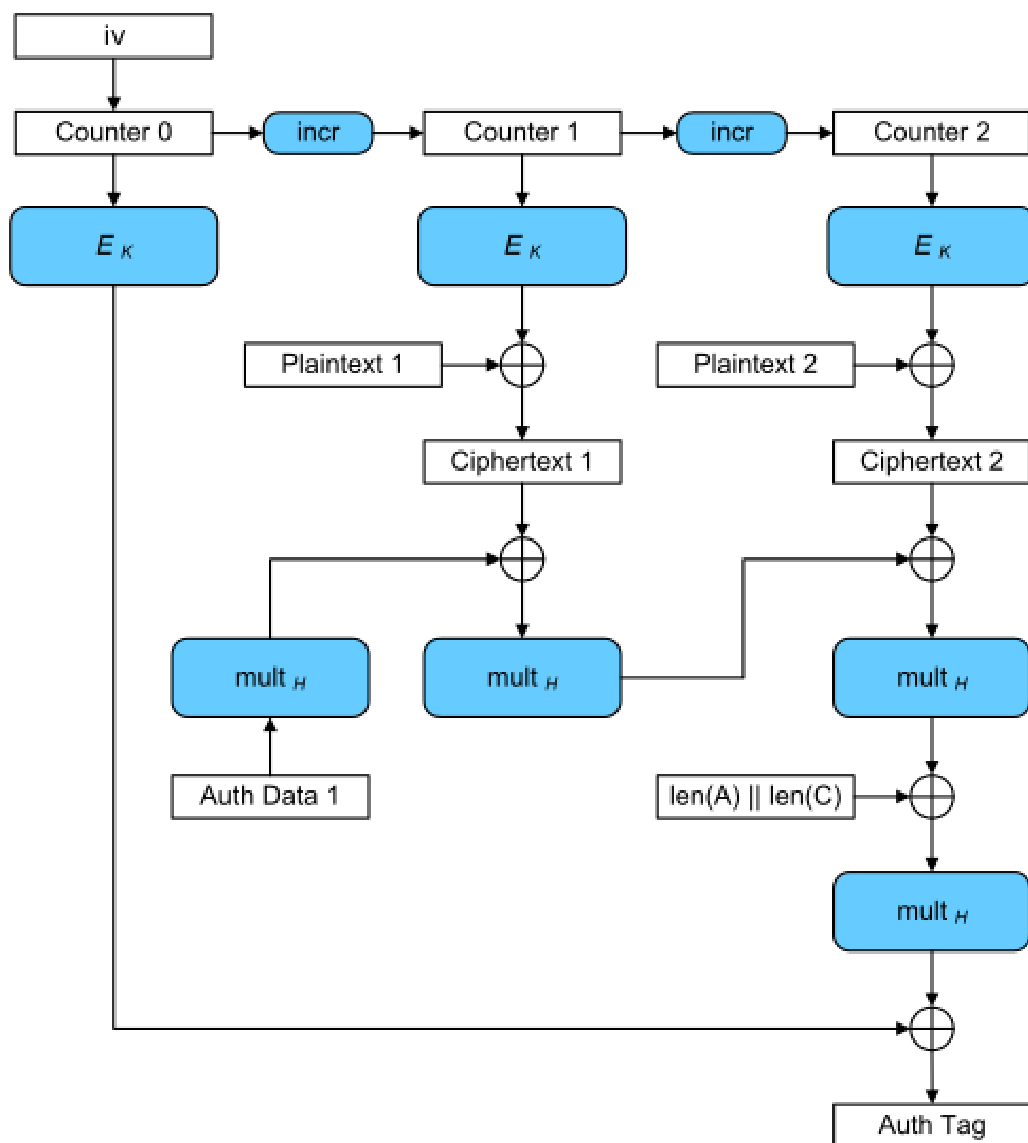

Figura 2.27: Struttura della modalità operativa GCM con 2 blocchi.

Capitolo 3

Implementazione del protocollo COSE

In questo Capitolo si discute nel dettaglio l'applicazione [27] sviluppata in Python. Chiamata *sequreCBOR* per evidenziarne le caratteristiche principali (l'inserimento della lettera *q* rimanda alla parola *earthquake* e dunque all'ambito di utilizzo), nel complesso il suo scopo è quello di simulare una architettura composta da un sensore (come quello presentato nel Capitolo 1) e da un centro operativo (anche un semplice computer), con i dati raccolti dall'accelerometro che vengono cifrati e codificati in CBOR prima di essere trasmessi attraverso il protocollo MQTT. Le operazioni principali sono svolte con l'ausilio delle librerie esterne presentate di seguito.

3.1 Librerie utilizzate

Come anticipato, la nostra implementazione si avvale di due pacchetti senza i quali con molta probabilità non sarebbe stato possibile raggiungere gli obiettivi prefissati. Questi si occupano rispettivamente di implementare il protocollo COSE e gestire i client in una infrastruttura MQTT e di seguito vengono presentati insieme alle loro funzionalità, così da poter procedere più velocemente con la discussione del codice affrontata poi nella Sezione 3.2.

pycose
La libreria *pycose* [28] consente l'utilizzo di tutto quanto specificato nella definizione del COSE [2] e nel nostro progetto viene applicata per la generazione della chiave e del messaggio oltre che nelle operazioni crittografiche e di serializzazione. Questo significa che ingloba al suo interno le librerie *cryptography* [29] e *cbor2* [30], responsabili rispettivamente di crittografia e formato CBOR in Python. Tra codificatore e decodificatore si usano:

- **key = CoseKey.from_dict(cose_key):** prende in ingresso un dizionario dove sono specificati i parametri della chiave e restituisce un oggetto *COSE SymmetricKey*.
- **msg = Enc0Message(phdr, payload):** dati header e payload restituisce un oggetto *COSE_Encrypt0*.
- **encoded = msg.encode():** implementa insieme cifratura secondo algoritmo scelto e serializzazione in formato CBOR.

- **decoded = CoseMessage.decode(encoded):** deserializza il messaggio ricevuto.
- **decrypted = decoded.decrypt():** decifra il messaggio decodificato.

paho.mqtt.python
Questa libreria [31] è un prodotto della Eclipse Foundation compatibile con le versioni 5.0, 3.1.1 e 3.1 del protocollo MQTT. Fornisce una classe client per permettere la connessione ad un qualsiasi Broker ed è dunque possibile pubblicare messaggi o sottoscriversi a topic per riceverli. Per Publisher e Subscriber sono necessari:

- **client = paho.mqtt.client.Client():** inizializzazione del client.
- **client.connect(broker_address):** connessione al server indicato.
- **client.publish(topic, data):** pubblicazione del messaggio nel topic specificato.
- **client.subscribe(topic):** sottoscrizione al topic.
- **client.disconnect(broker_address):** disconnessione dal server.

Oltre a quelli specificati possono essere passati come parametri di connessione e pubblicazione anche la porta cui connettersi o il livello di QoS, settati di default su 1883 e 0. Inoltre va detto che le funzionalità descritte non sono sufficienti a raggiungere lo scopo desiderato. Quando infatti un messaggio viene inviato o ricevuto, questo finisce in un buffer che viene letto e processato solo se sono state implementate le funzioni di *loop*, le quali lanciano poi dei messaggi di *callback* in base all'evento verificatosi (connessione, ricezione messaggio, ecc...). Non trattati qui per semplificare, sarà possibile comprendere meglio questi concetti procedendo con la lettura della Sezione 3.2.

3.2 Firmware

Viene ora analizzata la struttura del codice implementato, diviso in due diversi file da eseguire lato trasmettitore (*publisher.py*) o lato ricevitore (*subscriber.py*).

3.2.1 Publisher

Non trattando in alcun modo l'hardware necessario alla raccolta e al trasferimento dei dati, per una corretta simulazione si è reso necessario inserire come prima caratteristica la generazione di pacchetti coerenti con la realtà. Assegnata una particolare struttura per questioni di interoperabilità (deve essere semplice ricostruire un pacchetto miniSEED), ciò si ottiene attraverso una funzione che riceve come parametri il numero di campioni e la loro frequenza, riempie un vettore con interi casuali nel range $(0, 2^{32} - 1)$ e restituisce un JSON con le informazioni necessarie:

```
def packgen(numsamp, sps):
    ''' generates a random packet to simulate the measures '''
    tms = 1000*round(datetime.timestamp(datetime.now()))
    samples = []
    for _ in range(numsamp):
        sample = randint(0,pow(2,32)-1)
        samples.append(sample)
    packet= {"E":"enc: int, sps: {}".format(sps),"T":tms,"V":samples}
```

Tale pacchetto deve ora essere cifrato e codificato e in primo luogo serve avere una chiave. Questa viene generata una volta per tutte partendo da una stringa fissata di 256 bit e seguendo le linee guida di *pycose*:

```
#COSE KEY STRUCTURE
cose_key = {
    'KTY': 'SYMMETRIC', #key type
    'K': unhexlify('000102030405060708090a0b0c0d0e0f
                    000102030405060708090a0b0c0d0e0f')} #key = 256 bit
key = CoseKey.from_dict(cose_key)
```

Prima di mostrare il corpo principale del codice c'è un'altra parte da discutere. Si tratta di una funzione di *callback* eseguita quando il *loop* rileva un pacchetto CONNACK e che si occupa semplicemente di stampare a video alcuni dettagli:

```
def on_connect(client, userdata, flags, rc):
    ''' on_connect function, it is called after the connection '''
    print('Client connected with code: ' + str(rc))
    print('Publishing will start in 5sec, wait or press <CTRL+C> to
                                        stop!')
```

Il *main()* è responsabile delle invocazioni. I suoi compiti prevedono la raccolta degli input, la connessione del client al Broker online e gratuito situato in *test.mosquitto.org* [32] e la sistematica pubblicazione sul topic *data\sensor* di pacchetti precedentemente cifrati e codificati:

```
def main():
    ''' encrypts and encode the packet using COSE protocol and sends
                                        it via MQTT'''
    try:
        #COMMAND LINE ARGUMENT
        if len(sys.argv) < 3:
            print('Please select the number and the frequency of the
                                        samples!')
            sys.exit()
        numsamp = int(sys.argv[1]) #number of samples in a single
                                        packet
        sps = int(sys.argv[2]) #frequency of the samples
        delay = numsamp/sps #delay between packets
```

```
        #MQTT CONNECTION
        client = mqtt.Client() #random id to be sure that is unique
        client.on_connect = on_connect
        client.connect('test.mosquitto.org') #connection to broker
        client.loop_start()
        time.sleep(5) #5sec delay to allow the subscriber to connect
                                  without losing the first
                                  packets
        while True:
            time.sleep(delay)
            #COSE ENCRYPTO STRUCTURE
            msg = Enc0Message(
                phdr = {Algorithm: A256GCM, IV: os.urandom(12)}, #
                                        protected header
                                        with random IV = 12
                                         byte
                payload = dumps(packgen(numsamp,sps)).encode('utf-8'))
                                         #payload
            msg.key = key
            encoded = msg.encode() #encrypting and encoding in a
                                    single function
            #MQTT PUBLICATION
            client.publish('data\sensor', encoded) #topic, data
    except KeyboardInterrupt: #raised after <CTRL+C>
        print('Ending...')
        client.disconnect()
        client.loop_stop()
        sys.exit()
```

Osservando la struttura del *phdr* si noti come il vettore di inizializzazione venga ogni volta generato in modo random. Ciò ha il fine di evitare che più pacchetti vengano cifrati con stessa chiave e stesso IV ma nonostante una lunghezza di 96 bit, per via del paradosso del compleanno, questa probabilità non è poi così bassa e perciò il NIST raccomanda di cambiare chiave almeno dopo 2^{32} utilizzi.

3.2.2 Subscriber

Facendo uso di crittografia simmetrica, il ricevitore deve essere in possesso della stessa chiave con cui è stato cifrato il messaggio e pertanto quella parte di codice viene copiata senza modifiche:

```
#COSE KEY STRUCTURE
cose_key = {
    'KTY': 'SYMMETRIC', #key type
    'K': unhexlify('000102030405060708090a0b0c0d0e0f
                    000102030405060708090a0b0c0d0e0f')} #key = 256 bit
key = CoseKey.from_dict(cose_key)
```

Come in precedenza, anche in questo file sono state definite delle funzioni di *callback*. La prima, invocata dopo la connessione, è molto simile a quella già discussa ma responsabile anche dell'azione di sottoscrizione al topic desiderato:

```
def on_connect(client, userdata, flags, rc):
    ''' on_connect function, it is called after the connection '''
    print('Client connected with code: ' + str(rc))
    client.subscribe('data\sensor')
    print('Listening...visit "data.txt" to read messages or press <
                                    CTRL+C> to stop')
```

La seconda viene invece lanciata alla ricezione di un messaggio ed esegue la decodifica e la decifratura del pacchetto. Se l'operazione va a buon fine i dati vengono salvati sul file *data.txt*, altrimenti si ha a che fare con un possibile attacco e dunque si annota l'evento su *log.txt*:

```
def on_message(client, userdata, msg):
    ''' on_message function, it is called after the message is
                                    received '''
    try:
        received = msg.payload
        decoded = CoseMessage.decode(received) #decoding from CBOR
        decoded.key = key #key to decrypt
        decrypted = decoded.decrypt() #decrypting
        data = loads(decrypted.decode('utf-8'))
        #FILE
        with open('data.txt', 'a') as f1:
            f1.write(str(data) + '\n\n')
    except InvalidTag: #raised if the auth tag is not correct
        t = time.localtime()
        current_time = time.strftime("%H:%M:%S", t)
        with open('log.txt', 'a') as f2:
            f2.write('Attempted attack at {}\n'.format(current_time))
```

A questo punto il corpo principale non deve far altro che stabilire la connessione MQTT e dare il via al *loop*:

```
def main():
    ''' connects to the broker and calls the function to read and save
                                    data '''
    try:
        #MQTT CONNECTION
        client = mqtt.Client() #random id to be sure that is unique
        client.on_connect = on_connect
        client.on_message = on_message
        client.connect('test.mosquitto.org') #connection to broker
        client.loop_forever()
    except KeyboardInterrupt: #raised after <CTRL+C>
        print('Ending...')
        client.disconnect()
        sys.exit()
```

Differentemente dal Publisher ora si utilizza *loop_forever* al posto di *loop_start*. Questa scelta è consigliata quando il programma deve girare per un tempo indefinito (come nel nostro caso) ed inoltre consente riconnessioni automatiche.

3.3 Setup sperimentale

Come già introdotto nel Capitolo 2, l'implementazione del protocollo COSE può essere fatta utilizzando diversi algoritmi crittografici o dimensioni della chiave. In questo lavoro di tesi, la configurazione migliore è stata individuata effettuando test su dimensione del pacchetto e latenza delle operazioni crittografiche, misurando rispettivamente il tempo impiegato dalla funzione *encode()*, che include sia serializzazione che cifratura, e dall'insieme di *decode()* e *decrypt()*. Tali analisi sono state eseguite su una versione offline del firmware, con Publisher e Subscriber implementati nello stesso file e senza utilizzo del Broker, mentre al fine di avere esiti attendibili il confronto è stato fatto su 8 diversi numeri di campioni:

$$\text{campioni} = [100,250,500,1000,2000,4000,8000,16000]$$

ognuno riprodotto 100 volte e valutato con il valore medio delle misurazioni. La macchina su cui sono state eseguite le prove ha le seguenti caratteristiche: Intel Core i7-8565U CPU, 1.80 GHz, 8 GB RAM e sistema operativo Windows 11. Con delle piccole necessarie modifiche, lo stesso setup è stato poi utilizzato anche per i test volti ad indagare l'overhead introdotto dal protocollo COSE o per il confronto tra questo e il JOSE. Il codice utilizzato può essere visionato nel percorso *src/test* del progetto [27].

Capitolo 4

Risultati

Di seguito si riportano i risultati delle analisi effettuate con il setup presentato alla fine del Capitolo 3. Vengono discusse la scelta della struttura crittografica, l'overhead introdotto e infine si confrontano i protocolli COSE e JOSE in medesima configurazione.

4.1 Confronto tra algoritmi

La scelta di utilizzare l'algoritmo AES-GCM con chiave a 256 bit (A256GCM) ovviamente può e deve essere giustificata. Come visto nel Capitolo 2 sono tre i possibili algoritmi nel caso di crittografia simmetrica ma, mancando nella libreria *pycose* l'implementazione del ChaCha20/Poly1305, l'analisi ha tenuto conto di tutte le possibili configurazioni di AES-GCM e AES-CCM cercando la migliore soluzione in termini di latenza delle operazioni e dimensione del pacchetto generato. Tali configurazioni differiscono per la lunghezza della chiave (128, 192 o 256 bit) ma hanno tutte in comune un vettore IV di 12 byte perchè AES-GCM viene raccomandato così. Per AES-CCM si potrebbe invece avere qualsiasi lunghezza tra 7 e 13 byte e proprio tale parametro differenzia le versioni CCM16 e CCM64, entrambe comunque implementabili con 12 byte e perciò inserite nei test. Il risultato in Figura 4.1 mostra come i tempi, seppure in maniera non schiacciante e da 1000 campioni in poi, lascino preferire l'uso di GCM, che invece equivale all'altro guardando solo le dimensioni ottenute. Per dimostrare quest'ultima cosa non serve alcun grafico (che è comunque stato prodotto in fase di test per completezza) ma basta sottolineare come entrambe le modalità operative studiate generino il *ciphertext* alla stessa maniera, utilizzando la modalità *counter* (CTR) e senza espandere il *plaintext*. L'unica differenza consiste allora nella costruzione del tag però, anche qualora questo sia di grandezza diversa, in entrambi i casi si occupano al massimo 128 bit, troppo pochi per impattare in modo significativo su pacchetti di svariati kilobyte totali. Non ci sono inoltre sostanziali differenze tra le varie chiavi e pertanto è bene scegliere la versione a 256 bit così da avere la massima sicurezza verso attacchi a forza bruta. Al di là del risultato, si noti dai grafici la presenza di *outliers* che rendono l'andamento delle linee non perfettamente monotono. Ciò è probabilmente dovuto a Python o all'hardware utilizzato ma comunque trascurabile data la scala di riferimento.

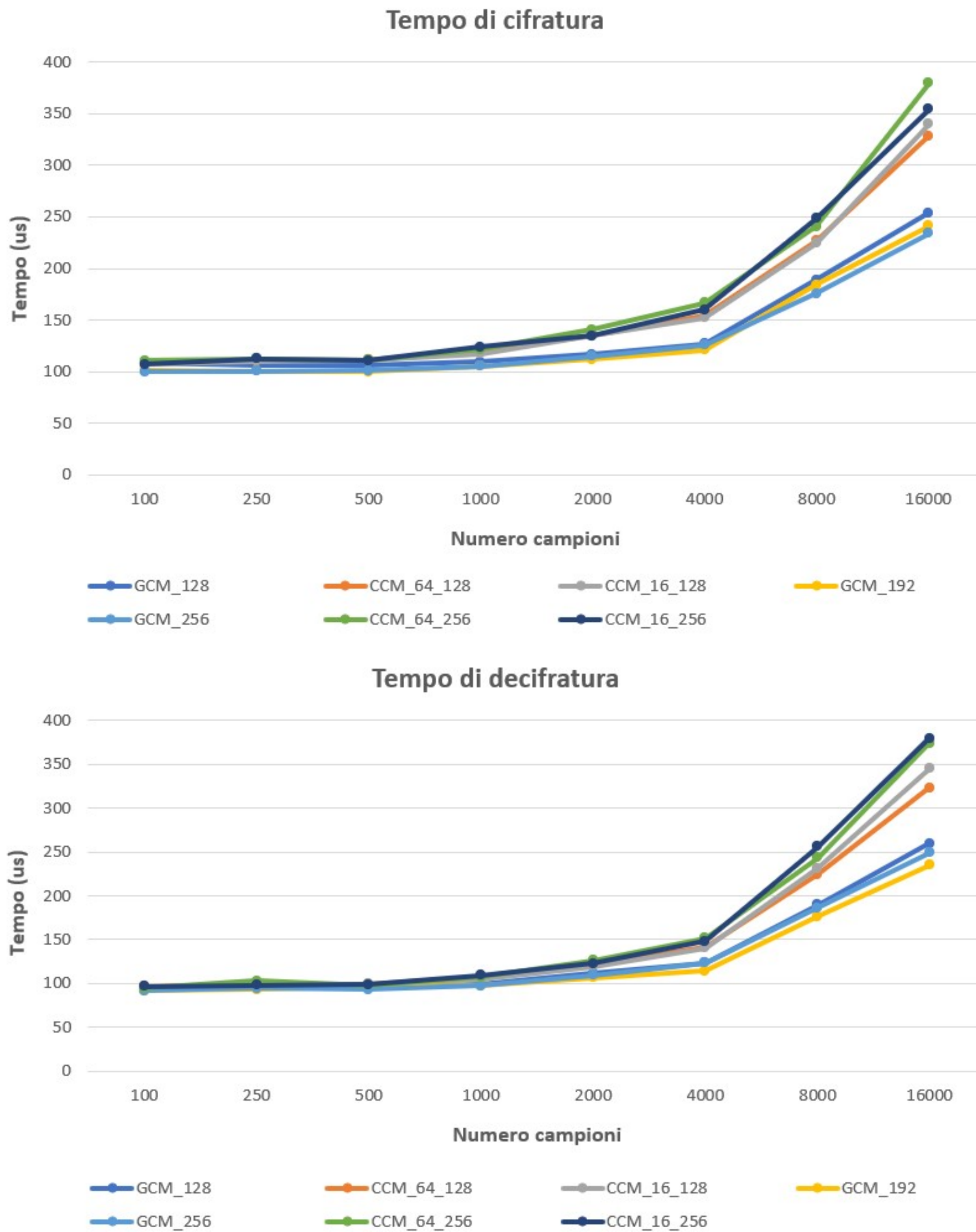

Figura 4.1: Latenza con diverse configurazioni di COSE.

4.2 Overhead introdotto

Se tra le varie implementazioni del protocollo COSE non c'è differenza di dimensione, questa diventa invece un parametro fondamentale per valutare l'impatto dell'introduzione della cifratura. La Figura 4.2 mostra il confronto tra pacchetto standard JSON, serializzato CBOR e cifrato secondo la configurazione scelta, e il risultato evidenzia (sempre da 1000 campioni in poi) come il vantaggio di passare al binario venga perso una volta considerato l'overhead prodotto dall'algoritmo crittografico. Come analizzato anche in [33], c'è dunque un notevole trade-off tra sicurezza e dimensione del pacchetto. In generale, comunque, questo effetto non è così negativo perchè, a patto di accettare risorse e tempi necessari per le operazioni, si ha la stessa dimensione iniziale una volta garantita anche la sicurezza della trasmissione.

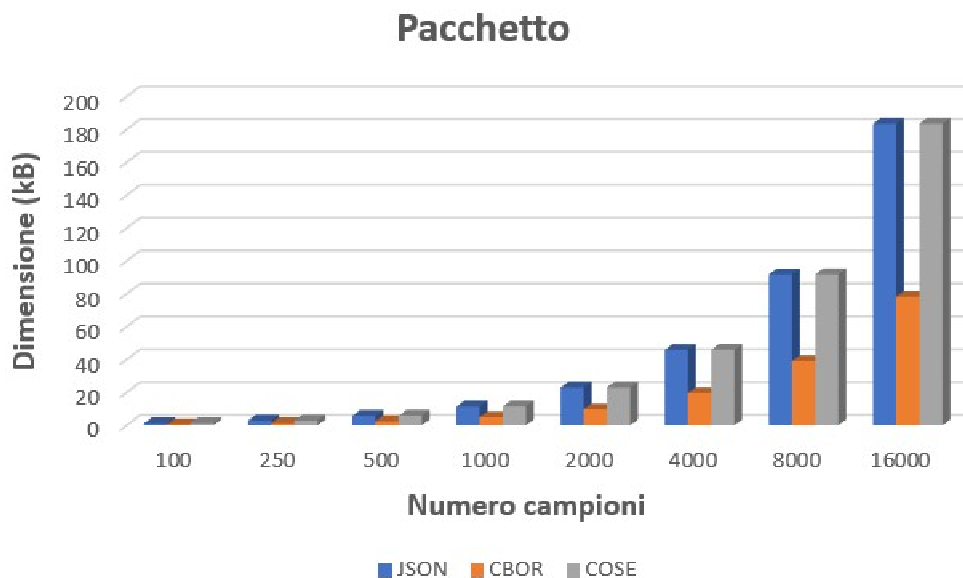

Figura 4.2: Dimensioni di pacchetto iniziale, serializzato e cifrato con A256GCM.

4.3 Confronto tra protocolli COSE e JOSE

In [1] era stato dimostrato come il formato CBOR fosse una soluzione migliore del JSON e per tale ragione il presente elaborato si è focalizzato soltanto su studio e implementazione del protocollo COSE. Ora questa idea può essere ulteriormente consolidata osservando i grafici proposti nelle Figure 4.3 e 4.4. Qui tutti i parametri considerati dimostrano il vantaggio, che in accordo con [1] diventa più evidente con un numero crescente di campioni, del protocollo adottato nei confronti del JOSE, testato introducendo delle semplici modifiche al codice già scritto e utilizzando la libreria *python-jose* [34]. Prima del confronto sono state analizzate anche le sue

prestazioni con chiavi di diversa dimensione e nemmeno questa volta l'indagine ha evidenziato differenze tali da preferire lunghezze inferiori di 256 bit.

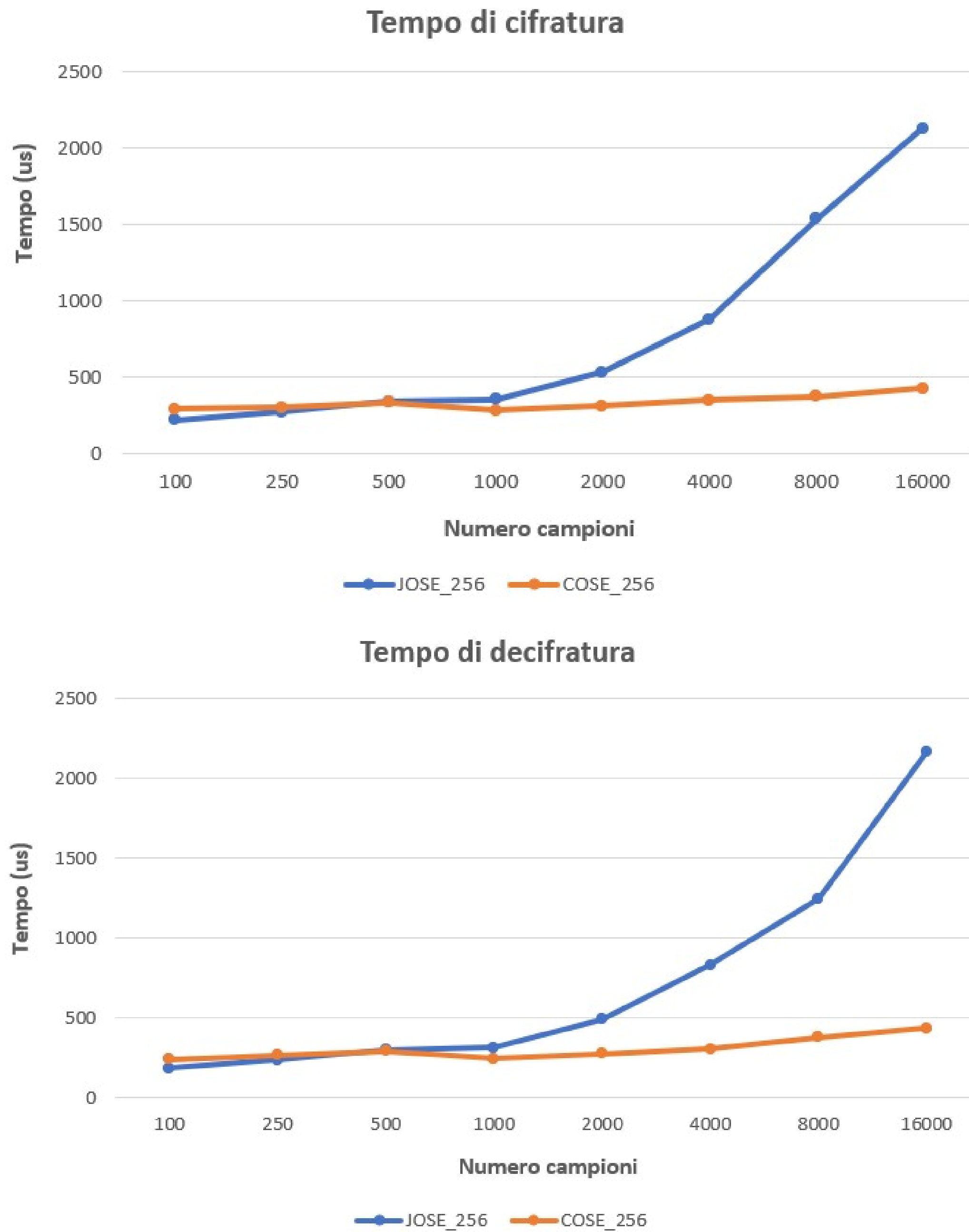

Figura 4.3: Latenza con JOSE e COSE in configurazione A256GCM.

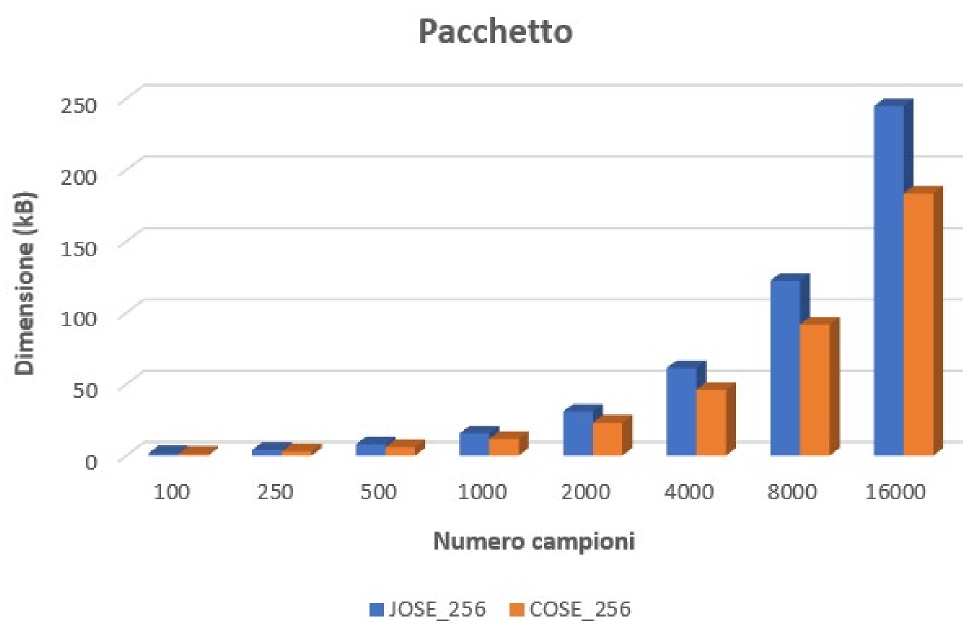

Figura 4.4: Pacchetto con JOSE e COSE in configurazione A256GCM.

Capitolo 5

Conclusioni

Il lavoro completato nasce dall'esigenza di dare protezione ai dati trasmessi lungo la rete e, come spiegato nel Capitolo 2, la soluzione migliore è quella di cifrare i pacchetti all'interno dello stesso dispositivo che si occupa poi di inoltrarli. Il firmware sviluppato e proposto nel Capitolo 3 simula l'architettura utilizzata e sfrutta per la parte crittografica l'algoritmo AES in modalità GCM e con chiave a 256 bit, scelti sulla base di una analisi che ha tenuto conto di latenza e dimensioni prodotte con diverse implementazioni. Questa configurazione è stata prima confrontata con una semplice serializzazione CBOR per valutare l'overhead introdotto dalla cifratura e poi con una medesima implementazione del protocollo JOSE. La Tabella 5.1 offre una sintesi di entrambi i risultati e mostra che, malgrado un chiaro ed inevitabile trade-off tra sicurezza e dimensione (a 4000 campioni il COSE vale il 134% in più del solo CBOR), l'impiego del COSE consente di produrre un pacchetto sicuro pesante quanto quello iniziale. Sempre con riferimento ad un pacchetto di 4000 campioni, usando il JOSE al posto del COSE, si avrebbero invece un aumento di dimensione del 33% circa e operazioni più lente di centinaia di microsecondi (una media di circa il 160% tra cifratura e decifratura) sull'hardware utilizzato.

Tipologia	Formato	Dimensione (kB)
Standard	JSON	45.92
Cifrato	JOSE (JSON)	61.31
Serializzato	CBOR	19.57
Serializzato e cifrato	COSE (CBOR)	45.96

Tabella 5.1: Confronto delle dimensioni di pacchetti con 4000 campioni.

Una volta verificato che, grazie al passaggio al binario, la crittografia non incide in maniera pesante sulle prestazioni, anche la sicurezza ottenuta può essere evidenziata. Come in Figura 5.1, per fare ciò basta aprire il software Wireshark e avviare una cattura di rete durante la trasmissione dello stesso pacchetto, prima in chiaro e poi protetto. Seppure dall'immagine non sia ancora chiara la differenza, cercando di decodificare entrambi i payload (campo *Message*), con un banale script Python o anche con il sito *cbor.me*, si ottiene quanto illustrato in Figura 5.2. Se dal primo pacchetto viene restituita tutta l'informazione contenuta, nel secondo non c'è nulla che un malintenzionato possa sfruttare a suo vantaggio. Ricordando che la situazione

qui considerata è molto semplice da riprodurre vista l'assenza di autenticazione al Broker, il presente risultato corrisponde proprio all'obiettivo desiderato.

Figura 5.1: Cattura Wireshark di pacchetto a) non cifrato e b) cifrato.

Figura 5.2: Tentativo di decodifica del pacchetto.

Alla fine, resta principalmente un'importante criticità legata all'utilizzo di una chiave fissa. Tale aspetto può essere ripreso e approfondito per uno sviluppo futuro, ma si tenga sempre bene in considerazione che velocità e leggerezza sono proprietà imprescindibili per un utilizzo in questo ambito, dove la differenza tra ricevere ed elaborare dati più o meno velocemente può salvare delle vite umane.

Bibliografia

[1] Zacchilli A. Studio e implementazione di protocolli real time per il monitoraggio sismico e struttrale. Master's thesis, Università Politecnica delle Marche, 2021.

[2] Schaad J. and Cellars A. RFC 8152 CBOR Object Signing and Encryption (COSE), 2017. URL: https://www.rfc-editor.org/rfc/rfc8152.html.

[3] Balageas D., Fritzen C.-P., and Güemes A. *Structural Health Monitoring*, volume 90. John Wiley & Sons, 2010.

[4] Valenti S., Conti M., Pierleoni P., Zappelli L., Belli A., Gara F., Carbonari S., and Regni M. A Low Cost Wireless Sensor Node for Building Monitoring. 2018.

[5] MQTT: The Standard for IoT Messaging. URL: https://mqtt.org/.

[6] Seismological Society of America. Tests Reveal Cybersecurity Vulnerabilities of Common Seismological Equipment, February 2021.

[7] Bormann C. and Hoffman P. RFC 8949 Concise Binary Object Representation (CBOR), 2020. URL: https://www.rfc-editor.org/rfc/rfc8949.html.

[8] Bormann C., Ersue M., and Keranen A. RFC 7228 Terminology for Constrained-Node Networks, 2014. URL: https://www.rfc-editor.org/info/rfc7228.

[9] CBOR playground. URL: https://cbor.me/.

[10] Soni D. and Makwana A. A survey on MQTT: A protocol of Internet of Things (IoT). April 2017.

[11] Toldinas J., Lozinskis B., Baranauskas E., and Dobrovolskis A. MQTT Quality of Service versus Energy Consumption. In *2019 23rd International Conference Electronics*, pages 1–4, June 2019. doi:10.1109/ELECTRONICS.2019.8765692.

[12] Prantl T., Iffländer L., Herrnleben S., Engel S., Kounev S., and Krupitzer C. Performance Impact Analysis of Securing MQTT Using TLS. pages 241–248, New York, NY, USA, April 2021. Association for Computing Machinery. URL: https://doi.org/10.1145/3427921.3450253.

[13] Harsha M.S., Bhavani .B.M., and Kundhavai K.R. . Analysis of vulnerabilities in MQTT security using Shodan API and implementation of its countermeasures via authentication and ACLs. In *2018 International Conference on Advances*

in Computing, Communications and Informatics (ICACCI), pages 2244–2250, September 2018. doi:10.1109/ICACCI.2018.8554472.

[14] Mahjabin T., Xiao Y., Sun G., and Jiang W. A survey of distributed denial-of-service attack, prevention, and mitigation techniques. In *International Journal of Distributed Sensor Networks*, volume 13, 2017. doi:10.1177/1550147717741463.

[15] Hassija V., Chamola V., Saxena V., Jain D., Goyal P., and Sikdar B. A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures. In *IEEE Access*, volume 7, pages 82721–82743, 2019. doi:10.1109/ACCESS.2019.2924045.

[16] IDC. IDC FutureScape: Worldwide IoT 2020 Predictions. URL: https://www.idc.com/research/viewtoc.jsp?containerId=US45591819.

[17] McKinsey&Company. What's new with the Internet of Things? URL: https://www.mckinsey.com/industries/semiconductors/our-insights/whats-new-with-the-internet-of-things.

[18] Anthraper J.J. and Kotak J. Security, Privacy and Forensic Concern of MQTT Protocol. Rochester, NY, March 2019. Social Science Research Network. URL: https://papers.ssrn.com/abstract=3355193.

[19] Syaiful A., Budi R., and Bagus H. Attack scenarios and security analysis of MQTT communication protocol in IoT system. In *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pages 1–6, September 2017. doi:10.1109/EECSI.2017.8239179.

[20] Rescorla E. Rfc 8446 The Transport Layer Security (tls) Protocol Version 1.3, 2018. URL: https://www.rfc-editor.org/info/rfc8446.

[21] Conti M., Dragoni N., and Lesyk V. A Survey of Man In The Middle Attacks. *IEEE Communications Surveys Tutorials*, 18(3):2027–2051, 2016. doi:10.1109/COMST.2016.2548426.

[22] Shodan Search Engine. URL: https://www.shodan.io/.

[23] Feily M., Shahrestani A., and Ramadass S. A Survey of Botnet and Botnet Detection. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 268–273, 2009. doi:10.1109/SECURWARE.2009.48.

[24] Dworkin M. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, November 2007. URL: https://csrc.nist.gov/publications/detail/sp/800-38d/final, doi:10.6028/NIST.SP.800-38D.

[25] Whisting D., Housley R., and Ferguson N. Counter with CBC-MAC (CCM), September 2003. URL: https://datatracker.ietf.org/doc/rfc3610/.

[26] Nir Y. and Langley A. ChaCha20 and Poly1305 for IETF Protocols, May 2015. URL: https://datatracker.ietf.org/doc/rfc7539/.

[27] Fava T. sequreCBOR. URL: https://github.com/ingtommi/sequreCBOR.

[28] Claeys T. pycose. URL: https://github.com/TimothyClaeys/pycose.

[29] Python Cryptographic Authority. cryptography. URL: https://github.com/pyca/cryptography.

[30] Grönholm A. cbor2. URL: https://github.com/agronholm/cbor2.

[31] Eclipse Foundation. Eclipse Paho MQTT Python Client. URL: https://github.com/eclipse/paho.mqtt.python.

[32] Mosquitto. URL: https://test.mosquitto.org/.

[33] Afzal S., De Biase LCC, Fedrecheski G., Pereira WT, and Zuffo MK. Analysis of Web-Based IoT through Heterogeneous Networks: Swarm Computing over LoRaWAN. 2022. doi:10.3390/s22020664.

[34] Davis M. python-jose. URL: https://github.com/mpdavis/python-jose.